Learn by doing: less theory, more results

jQuery 1.4 Plugin Development

Build powerful, interactive plugins to implement jQuery to its best

Beginner's Guide

Giulio Bai

[PACKT] PUBLISHING open source community experience distilled

jQuery 1.4 Plugin Development

Beginner's Guide

Build powerful, interactive plugins to implement jQuery to its best

Giulio Bai

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

jQuery 1.4 Plugin Development
Beginner's Guide

First published: October 2010

Production Reference: 1121010

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-849512-24-4

`www.packtpub.com`

Cover Image by Asher Wishkerman (`a.wishkerman@mpic.de`)

Credits

Author
Giulio Bai

Reviewers
Abel Mohler
Peter Guo Pei
Keith Wood

Acquisition Editor
Chaitanya Apte

Development Editor
Chaitanya Apte

Technical Editor
Hithesh Uchil

Indexer
Hemangini Bari

Editorial Team Leader
Aanchal Kumar

Project Team Leader
Lata Basantani

Project Coordinator
Shubhanjan Chatterjee

Proofreader
Chris Smith

Graphics
Nilesh Mohite

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the Author

Giulio Bai is a law student living in Modena, Italy who spends most of his time toying with stuff that doesn't have anything to do with law.

Even after trying to keep the list of his past achievements as short as possible, the number of projects he joined in (and invariably sunk short thereafter) makes it hard to narrow down his interests to programming and carousels alone.

It should be made clear that any claim of responsibility for those unfortunate ventures is wholeheartedly rejected—they never had the necessary potential to make it anyway.

I can't brag about this book with anybody if no credit for the beautiful JavaScript library jQuery is given to its author, John Resig.

Also, a bunch of thanks are randomly distributed to everybody I had any kind of contact with, in both real and virtual life, who have—no doubt—somehow helped me in writing this precious manuscript.

About the Reviewers

Abel Mohler is a freelance web developer and jQuery plugin author who works from his home near Asheville in the mountains of North Carolina. He is the author of popular jQuery plugins such as Mapbox and wTooltip. You can see a list of the plugins he has released at `http://wayfarerweb.com/jquery/plugins/`.

I'd like to thank those at Packt Publishing who reached out to me to work on a project as fun as this one, to the author for doing such a wonderful job with the material, and to Project Coordinator Shubhanjan Chatterjee for his patience and diligence in helping glue this project together into what it became.

I'd also like to thank those who helped me along the way to become a better developer, Brett Lytle of Lytleworks, who has the vision to find unique and simple solutions to any problem, Matt McCabe for his endless ideas and projects, and Mike Bykov of TigerTiger for helping to inspire me to grow my own technologies. Most of all, I'd like to thank my wife, Rebecca, for putting up with countless sleepless nights of studying, and pushing me to be a better man.

Peter Guo Pei is a Chinese Canadian website and software specialist. His expertise is mainly in the design of websites and applications and other computer software systems. He lives in the quiet town of Langley along the US-Canadian border with his lovely wife and two kids. He studied computer science in Fudan University China.

He has worked for various IT companies in China, USA, and Canada, including Sun Microsystems, Tandem, Wang, Kodak, and Motorola.

He loves to ride his bike.

I would like to thank my sweet wife Yan and my two lovely kids – my daughter Angel and son Jimmy. They have always been the sunshine of my life.

Keith Wood lives in Brisbane, Australia, where he is a Solutions Architect for Hyro Ltd.

He has been in the IT industry for over 20 years, working his way down from mainframes, through mini-computers, to PCs. He has used Delphi and JBuilder since their first release, contributing many OpenTools to the JBuilder community. He was also a frequent contributor of technical articles to *Delphi Informant*, *Delphi Developer's Journal*, *Hardcore Delphi*, and *The Delphi Magazine* magazines, and has written three books:

- *Delphi Developer's Guide to XML*, WordWare Publishing, 2001
- *Delphi Developer's Guide to XML*, 2nd Edition, BookSurge, 2003
- *Inside the JBuilder OpenTools API*, BookSurge, 2004

He did the initial development for log4d, a port of log4j to Delphi, and SAX for Pascal.

More recently, he has worked with jQuery for several years and has contributed many jQuery plugins—`http://keith-wood.name/index.html#jquery`—as well as developed with Marc Grabanski the Datepicker component that was incorporated into the jQuery UI project.

Mostly, he works with Java these days, but uses jQuery for any frontend work.

Table of Contents

Preface

jQuery is the most famous JavaScript library. If you use jQuery a lot, it may be a good idea to start packaging your code into plugins. A jQuery plugin is simply a way to put your code into a package, which makes it easier to maintain your code and use it across different projects. Although basic scripting is relatively straightforward, writing plugins can leave people scratching their heads.

With this exhaustive guide in hand, you can start building your own plugins in a matter of minutes! This book takes you beyond the basics of jQuery and enables you to take full advantage of jQuery's powerful plugin architecture to deliver highly interactive content to your website viewers.

This book contains all the information you need to successfully author your very own jQuery plugin with a particular focus on the practical aspect of design and development.

This book will also cover some details of real-life plugins and explain their functioning to gain a better understanding of the overall concept of plugin development and jQuery plugin architecture.

Different topics regarding plugin development are discussed, and you will learn how to develop many types of add-ons, ranging from media plugins (such as slideshows, video and audio controls, and so on) to various utilities (image pre-loading, handling cookies). You will also learn the use and applications of jQuery effects and animations (sliding, fading, and combined animations) to eventually demonstrate how all of these plugins can be merged and give birth to a new, more complex, and multipurpose script that comes in handy in a lot of situations.

What this book covers

Chapter 1, *What is jQuery About?*, covers what jQuery is and why we should use and prefer it over other libraries. Some basic concepts, as well as some history, are covered in this chapter that acts as an introduction to the real topic of the book.

Chapter 2, *Plugins Basics*, is our first real approach to jQuery plugins. It provides an in-depth description of jQuery's own plugin architecture, providing some examples and sample applications for some of the most popular plugins.

Chapter 3, *Our First jQuery Plugin*, as its name suggests, is about creating our first, working, and fantastic jQuery plugin! Step-by-step instructions are provided in order to guide even very beginners to the successful realization of their first plugin.

Chapter 4, *Media Plugins: Images Plugins*, discusses how images play a big role in today's Internet. Since we don't want to be left out, nor behind, in this chapter, we do our best to create a jQuery plugin that is very easy to use, customize, and at the same time, very effective and good looking. Besides, a gallery-like plugin will certainly enhance the user experience of our web pages!

Chapter 5, *Media Plugins: Audio Plugins*, shows us how, after images, sounds too can be used in a variety of different ways to hold the visitor's attention. Not only will we learn how to develop a jQuery-based audio player plugin, but we will also analyze the advantages and disadvantages of the HTML5 audio tag, compared to JavaScript solutions.

Chapter 6, *Media Plugins: Video Plugins*, presents a detailed guide to the creation of a video player plugin, and also offers some hints on how to better display video objects on a web page with the aid of JavaScript and/or HTML code.

Chapter 7, *Form Plugins*, shows a handful of different, but all extremely useful, plugins we can develop in order to improve our forms and offer an enhanced user experience on our website. A number of jQuery plugins are coded, step-by-step, and discussed to better understand what to use, how to use it, and in what circumstances.

Chapter 8, *User Interface Plugins*, offers many plugin examples and explains how the developer should tackle the problem, in such a way that the final result can be easily modified and integrated into an organized project.

Chapter 9, *User Interface Plugins: Tooltip Plugins*, explains that to get a fully working tooltip plugin, a series of preliminary steps is required. These include understanding mouse movement and events, positioning through CSS rules, and, last but not least, interaction with jQuery code to actually show and hide the tooltip element at our will.

Chapter 10, *User Interface Plugins: Menu and Navigation Plugins*, discusses how developing menu and navigation plugins with some additional effects to enhance their appearance and user experience is rather simple. The principles are explained in this chapter, as well as a number of different approaches that we might want to use to obtain a menu plugin.

Chapter 11, *Animation Plugins*, discusses how fun-to-activate and nice-to-look-at animation plugins play one of the most important roles when it comes to user interaction. Be it a moving image or a bouncing shape, they are always worth the time spent coding them and actually amuse the visitor. We will learn how to make things move, bounce, fade in and away—nothing more, nothing less.

Chapter 12, *Utility Plugins*, shows how creating utility plugins (which can be easily used thanks to jQuery's own internal structure and which allow for a very effective integration) is a big plus. If we need some kind of function or method to take care of some repetitive task, we could speed up the process with just a few lines of code.

Chapter 13, *Top jQuery Plugins*, is a selection of the top 10 plugins. It briefly shows how they are customized on a website, their uses, their advantages and disadvantages, as well as provides a basic documentation that readers can easily use and refer to when (and if) they decide to mess with any of the plugins discussed in this chapter.

Appendix A, *Tools, reference, and final recommendations*, contains some useful jQuery-related links and offline resources for further reference.

Who this book is for

This book is for anyone who wants to have a better understanding of the dynamics of jQuery when plugins come into play, as well as for those who are willing to push jQuery to its limits and develop awesome plugins to use on their websites. A little background information about JavaScript and jQuery cannot harm anyone, but even very beginners can have a chance to be introduced to the wonderful world of jQuery.

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions on how to complete a procedure or task, we use:

Time for action – heading

1. Action 1
2. Action 2
3. Action 3

Instructions often need some extra explanation so that they make sense, so they are followed with:

What just happened?

This heading explains the working of tasks or instructions that you have just completed.

You will also find some other learning aids in the book, including:

Pop quiz

These are short multiple choice questions intended to help you test your own understanding.

Have a go hero – heading

These set practical challenges and give you ideas for experimenting with what you have learned.

You will also find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
jQuery.fn.txtHover = function() {
  return this.each(function() {
    jQuery(this).hover(function() {
      jQuery(this).text("Mouse hovered");
    });
  });
};
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "To enable the Web Inspector, open **Preferences**, go to the **Advanced** tab, and select the **Show develop menu in the menu bar** item".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for this book

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
What is jQuery About?

With the ever increasing number of websites and an overall surge of web professionals trying to make the Web a more beautiful and usable place, JavaScript has become fairly popular amongst web designers and developers in an attempt to overcome HTML and CSS shortcomings.

But, as we all know, JavaScript is a rather obtrusive language. It often happens to mess things up and worsen what before was good markup if we don't pay close attention to using unobtrusive JavaScript solutions. These are the reasons why web designers (and web professionals in general) avoid plain JavaScript like a plague and limit its usage to short and simple parts of the coding process.

Instead, jQuery has been designed with the aim of making it easier to navigate a document, select **Document Object Model** *(***DOM***) elements, handle events, develop AJAX applications, and eventually smooth out any browser differences.*

In this chapter, we will cover the following:

- jQuery background
- A jQuery introduction
- How jQuery works
- Extending jQuery using plugins
- jQuery plugin basics
- A reading material reference

A little background

Short after being officially presented by John Resig at BarCamp NYC in January 2006, jQuery, though still "new", spread rather quickly. It has become, as of today, the most used JavaScript library and is in use at 20 percent of the 10,000 biggest websites, including Google, Digg, and WordPress.

The reason why jQuery was created is to be found in the lack of a JavaScript library providing its users with a simple and easy-to-use syntax. In fact, at the time of its announcement, jQuery was aiming to improve and simplify the use of selectors in JavaScript—a topic much overlooked by libraries such as Behaviour.

The library then rapidly gained community interest and, shortly after the first plugin had been developed, AJAX support and some new effects were added. Less than one year later, the first sponsored developer joined the team and, as of now, four years later, companies such as Nokia and Microsoft are actively supporting the open source library.

Its success, so huge and originating a fast growing movement, which has undoubtedly contributed to promoting the library, has definitely helped jQuery to constantly improve the quality of both its features and code. This has made it more and more popular over time, especially amongst ASP.NET developers, as a 2009 survey points out (`http://codeclimber.net.nz/archive/2009/06/22/ajax-survey-2009-jquery-and-ms-ajax-are-almost-tied.aspx`).

The code, free and dual licensed under the MIT License and the GNU General Public License (GPL) Version 2, proves its suitability to the purpose by being extremely lightweight and cross-browser, supporting a variety of web applications and taking relatively little time to execute.

jQuery: "the write less, do more JavaScript library"

Indeed, jQuery provides a simple and fast way to manipulate web pages, emphasizing the interaction between JavaScript and HTML. Even a few lines of code can make the User Interface (UI) more logical and way nicer to look at.

At first glance, we might think jQuery is only a different way to write JavaScript. However, after spending some time dealing with documentation or examples, we realize it's much more than a mere framework. It actually has features that make it easier and extremely straightforward to handle DOM elements (traversal, modification, and elements selection), deal with events (through specific calls), manipulate CSS, and create any type of effect and animation (sliding, fading, or combined effects).

Moreover, one of the main, big benefits of using jQuery over plain JavaScript is that the former hides the differences between browsers, at least to some level, relieving us of the onerous task of differentiating the code depending on the user agent.

Ultimately, it provides easy methods to access AJAX functionalities and extend the library itself through the use of plugins, which is the most powerful and useful way to interact with the jQuery API.

How jQuery works

To get the most out of this book there are a couple of things that we should have clear in mind:

- How to get a simple jQuery script to run correctly
- Understanding what callbacks are and how they work

Time for action – writing a basic jQuery script

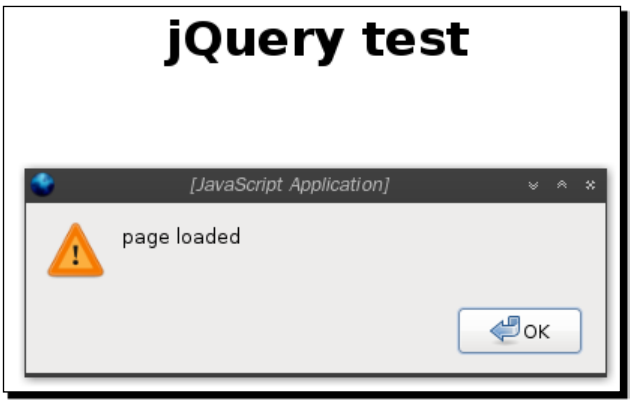
We're going to create a simple script to check if everything is set up correctly and is working properly.

1. Load the jQuery library; modify the `src` attribute of the `<script>` element to point to the path of your jQuery file.

2. Write some sample code inside the "document ready" event statement, to display a pop up message, as follows:

```
<html>
<head>
  <meta http-equiv="Content-type" content="text/html; charset=utf-
    8" />
  <title>jQuery test</title>

  <script src="jquery.js" type="text/javascript"></script>
  <script type="text/javascript">
    $(document).ready(function() {
      // runs once page is loaded and ready to be manipulated
      alert("Hello world!");
    });
  </script>
</head>
<body>
  <center><h1>jQuery test</h1></center>
</body>
</html>
```

3. Open the page in your web browser of choice and check if everything is alright.



What just happened?

There's no need to spend time talking about stuff like loading JavaScript libraries or displaying pop ups. Instead, it would make a good point to explain why, contrary to what most JavaScript programmers will do instinctively, we avoid adding code to our program once the `window.onload` event strikes—right after the page is loaded.

However, the JavaScript code isn't actually run until all images are finished downloading, (which could last for quite a lot of time). Instead, jQuery's "document ready" event checks the document and only waits until it's ready to be manipulated, leaving images and other media to load at their own pace.

Time for action – callback and functions

Callbacks are functions that are passed as an argument to another function and are to be executed at the appropriate time within the processing of the code (for example, when a click event happens or when an AJAX update is ready to be sent).

1. Inside the "document ready" statement, write these two functions with callback functions included:

```
$('#one').click(function() {
  $(this).hide(1000, function() {
    alert("hidden - callback function with one argument");
  });
});

$('#two').click(function() {
  $(this).hide(1000, myCallbackFunction);
```

```
});

function myCallbackFunction() {
  alert("hidden - callback function with no arguments");
}
```

2. Make sure you have two elements having IDs `one` and `two` respectively.
3. Point the web browser to this page to check the work done.

What just happened?

It's important to notice that, though the final result is the same, in this case, there is difference in the way callbacks are to be handled depending on them having (or not having) arguments:

- If the callbacks don't require arguments, writing the function name (not as a string, nor with any parenthesis) is enough.
- If the callbacks do have arguments to be passed along, we must register an anonymous function as the callback function and then execute the actual callback taking any number of arguments.

Another interesting point to understand is, in fact, what anonymous functions are and how they behave.

Following the previous example, we may notice we have bound the `click` event directly to a structure in which we defined a function. This is what is called an anonymous function. It has no name and can be defined on the spot, resulting in a useful replacement for a function that we might use only once (and that would be a waste of time to define) and then call in two different places.

Extending jQuery: Plugins

Apart from offering a simple, yet effective, way of managing documents, elements, and various scripts, jQuery also offers a mechanism for adding methods and extra functionalities to the core module.

Thanks to this mechanism, we are also allowed to create new portions of code and add them to our application everytime we need them. It results in a reusable resource that we don't need to rewrite in our next page or project.

Additional methods and functions created making use of this structure are then bundled as plugins. These can be subsequently used (and/or included) in new jQuery scripts, developed by the plugin authors themselves and by other people as well (if the code is released in some way—thus made publicly available for download and use).

The extremely easy-to-use **Application Programming Interface** (**API**) jQuery is built on (evidently developed without forgetting the very beginner programmers out there, and the immediate syntax jQuery made us all used to), combined with a bit of will, makes jQuery plugin development not too harsh to regular script coders with a minimum of experience in the field, as well as to those new to either plugin writing or jQuery internal mechanisms.

Of course, writing simple plugins is fairly easy, whereas a more complex plugin requires a more advanced programming background and a certain proficiency with both JavaScript and jQuery.

Also, it's important to know that most of the methods and functions jQuery is packaged with were written by taking advantage of the jQuery plugin construct itself, thereby pushing towards steady and frequent improvements of this complex plugin architecture.

Plugins basics

The question "So, what's all this about?" is likely to come naturally now.

In fact, this "plugin thing" may sound a little strange to newcomers, if they're not used to dealing with languages or frameworks that allow for such extension of features and options.

To dissipate even the slightest doubt, we're going to understand what plugins are and why they matter. Most importantly, we will see how is it possible to bring to light our own creation, starting from scratch and eventually shaping our original idea into a more concrete, working jQuery plugin.

Plugins are coded by making use of the jQuery API functions and methods, which are really handy on many occasions. However, plain JavaScript often happens to be used heavily, since, after all, it's the language jQuery is written in.

For those already familiar with jQuery syntax, methods, and features (everybody should be, when considering writing a plugin), flipping through the pages of any jQuery-related book is enough. However, if an inexperienced jQuery developer is reading this (even though they usually jump straight to some random code, so they're more likely to never see this part anyway), they'd better stop for a while and read some beginners' guide to jQuery programming first.

Code generated using the jQuery built-in tools and eventually packaged into a plugin, must then be included in the web page it's intended to work in, without forgetting that it requires a compatible version of jQuery to run properly. More generally, in fact, plugins are extra parts, not expected nor supported in any way by jQuery developers. These extra parts are attached to the main functions and add in new user-generated functionalities working on top of the core methods, functions, and options explicitly provided by the jQuery library for third-party use.

It is true that plugins, especially the well known ones, are normally supported by their authors. The authors are, usually, open to suggestions and feedback, and are constantly doing their best to make their work compatible and working flawlessly with the latest release of jQuery.

Suggested reading that could help greatly

Thinking of plugin development as a wild-goose chase is obviously overplaying it. Some old wise man would probably say impossible is nothing, and he couldn't be more right. If each of these resources, all of this documentation, and these tools are at hand, authoring a plugin is a trifle requiring some special effort at the beginning along with a good knowledge of the basis.

Books

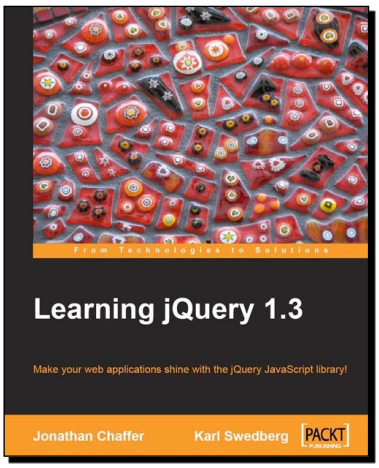
Printed books are, even in our extremely digitalized era, one of the best sources to learn from. Also, they come handy whenever we have to look something up and haven't the possibility to switch to another window or just need to check something up—the book is there, right next to our keyboard.

Learning jQuery 1.3

By Jonathan Chaffer and Karl Swedberg—Packt Publishing, 2009

Book page: `http://www.packtpub.com/learning-jquery-1.3/book`

TOC: `http://www.packtpub.com/article/learning-jquery-1.3-table-of-contents`



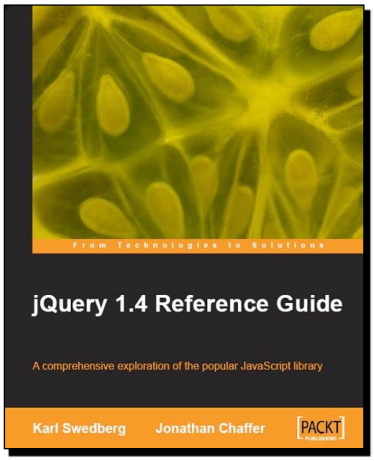
"Revised and updated for version 1.3 of jQuery, this book teaches you the basics of jQuery for adding interactions and animations to your pages. Even if previous attempts at writing JavaScript have left you baffled, this book will guide you past the pitfalls associated with AJAX, events, effects, and advanced JavaScript language features."

jQuery 1.4 Reference Guide

By Jonathan Chaffer and Karl Swedberg—Packt Publishing, 2010

Book page: `http://www.packtpub.com/jquery-1-4-reference-guide/book`

TOC: `http://www.packtpub.com/toc/jquery-14-reference-guide-table-contents`



"Revised and updated for version 1.4 of jQuery, this book offers an organized menu of every jQuery method, function, and selector. Each method and function is introduced with a summary of its syntax and a list of its parameters and return value, followed by a discussion, with examples where applicable, to assist in getting the most out of jQuery and avoiding the pitfalls commonly associated with JavaScript and other client-side languages."

Online reference and documentation

Beside printed paper resources, some online papers are often of great help when we are in need of some quick reference or other people's opinion as well. Here are some of the most useful and well-known pages we'll become familiar with.

jQuery.com

The official home of jQuery, provides lots of useful links, tutorials, and documentation:

- `http://jquery.com`
- `http://api.jquery.com`
- `http://docs.jquery.com/Plugins/Authoring`
- `http://plugins.jquery.com`

Nettuts

There are many, many resources on this site. Have a look through the archives and search for jQuery-related tutorials to get overwhelmed by articles upon articles.

They also have a very well made video series called *jQuery for absolute beginners*—worth watching.

- `http://net.tutsplus.com/`
- `http://net.tutsplus.com/articles/web-roundups/jquery-for-absolute-beginners-video-series/` (jQuery video tutorials for beginners)
- `http://net.tutsplus.com/tutorials/javascript-ajax/15-resources-to-get-you-started-with-jquery-from-scratch/` (list of interesting links to get you started)

Cheatsheets

The following are some useful and practical references. These contain API references with detailed description and some sample code. Everybody should try some of these and pick the one that is most helpful to them.

- `http://www.cheat-sheets.org/#jQuery`
- `http://woorkup.com/2009/09/26/jquery-1-3-visual-cheat-sheet/`
- `http://www.javascripttoolbox.com/jquery/cheatsheet/`
- `http://geek.michaelgrace.org/2009/06/jquery-cheat-sheet/`
- `http://api.jquery.com/browser/` (downloadable AIR version)

Forums and mailing lists

There are many people out there happy to help. Asking in the following forums might be useful to get different point of views for one problem and discuss topics of common interest:

- `http://forum.jquery.com/`
- `http://www.sitepoint.com/forums/forumdisplay.php?f=15` (JavaScript forum)
- `http://old.nabble.com/jQuery-General-Discussion-f15494.html` (jQuery mailing lists)
- `http://groups.google.com/group/comp.lang.javascript/` (JavaScript group)

Pop quiz

1. What is the point of preferring jQuery's own "document ready" statement to the more common, plain JavaScript `window.onload`?
   A. It's easier to remember and does the very same thing.
   B. The page loads faster.
   C. We know when jQuery is ready to operate.
   D. We don't have to load the whole library.
2. What are callback functions used for?
   A. To make the code nicer to look at.
   B. To make things even harder to understand.
   C. For backwards compatibility.
   D. To execute code right after another event has finished.
3. How can developers add functionalities to the core jQuery module?
   A. By developing plugins.
   B. By creating patches.
   C. By asking the jQuery development team.
   D. They can't.
4. Can plugins be used by anybody other than the programmer?
   A. Absolutely.
   B. Just friends.
   C. Nope.
5. How can plugins make use of jQuery's own methods?
   A. It's impossible to access jQuery methods.
   B. Through the API.
   C. Patching the original code.
   D. Using a third-party framework.

Summary

To sum it all up, in this chapter we've learned a bit of background about the jQuery library, as well as much interesting information related to its development and usage.

An important point is the one concerning basic jQuery usage, which should come naturally by now. Writing plugins is not so different to coding jQuery scripts, and this should be clear, but requires a little knowledge of what this all is about—we just started to get into this, more will come later on!

The next chapter is all about jQuery plugin architecture, and will cover topics explaining how plugins actually work and the main points of the plugin writing process.

2
Plugins Basics

Even though we're going to discuss and cover plugin authoring in more depth, we shouldn't forget plugins coming from other developers.

Also, knowing and understanding what makes plugins work and how one is supposed to use them, apart from being an integral part of plugin creation, is key to getting to a valuable and versatile final product everybody can use with no difficulties, possibly modify, and eventually tweak so it fits their needs.

Specifically, in this chapter we're going to discuss:

- Using plugins
- Looking for and actually using a plugin
- The structure of a plugin
- Making a plugin work
- Function plugins
- Methods plugins
- Basic plugins
- A few key things to remember

Using plugins

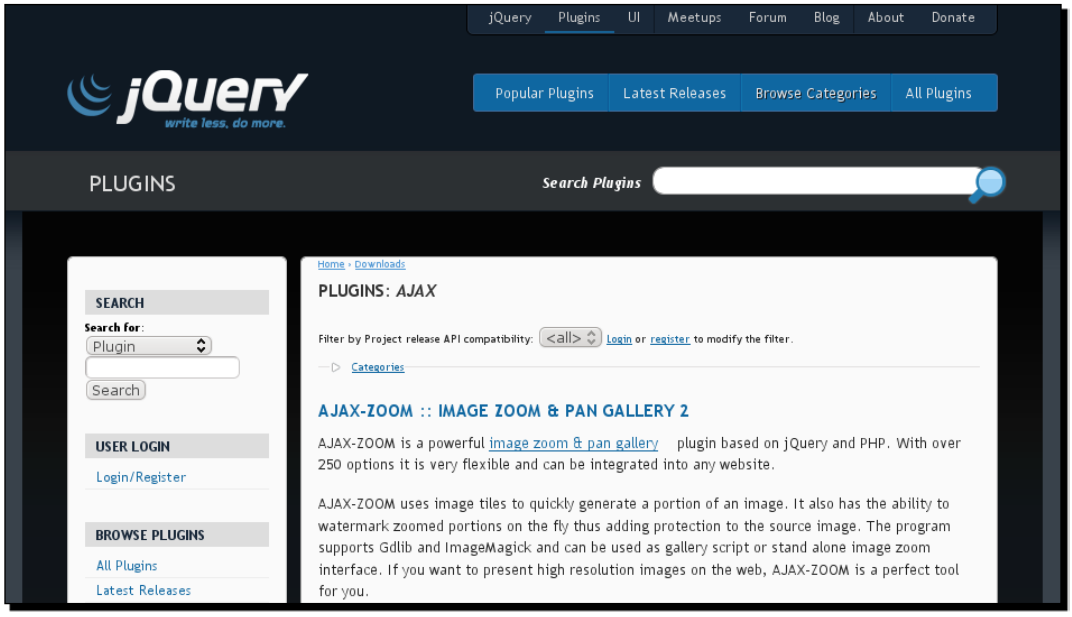
While considering one of those many plugins available for download through the Web, the very first question we might ask ourselves is: "Well, what am I supposed to do now?".

And, this is perfectly normal: we're just starting out after all!

Time for action – looking for a plugin

Suppose we're facing an extremely common, often underestimated, problem: form validation. How do we look for the best plugin that fits our needs?

1. Open up your web browser of choice, and leave it open. We're going to deal with the Internet a lot!
2. Glancing through the online resources section at the end of Chapter 1, one link, more than others, stands out—keeping in mind that the central topic of this book is jQuery plugins: `http://plugins.jquery.com`.



3. As you can see, there are lots and lots of plugins. You can browse by category, by name, or date. Or you can also search for a string.
4. There actually is a "Forms" category. However, it's more likely we bump into that famous needle while jumping onto the haystack than that we find what we need when we need it.
5. Enter "form validation" into the search field and start the search. An awful number of pages come up, as if everybody wrote a form validation plugin. Anyway, pick the Validation plugin by Jörn Zaefferer.

6. Skip to the **Releases** section, and download the most recent one.

Releases

Official releases	Date	Size	Links		Status	
1.5.5	2009-Jun-17	284.1 KB	Download	Release notes	Recommended for 1.2.x	✓
1.6.0	2009-Nov-30	287.78 KB	Download	Release notes	Recommended for 1.3.x	✓

View all releases

7. Surprisingly enough, all we get is a ZIP file filled with some files. Actually, we expect some files, such as the following, to be present in every package that we download:
   - `jquery.PLUGINNAME.js`
   - `jquery.PLUGINNAME.pack.js`
   - `jquery.PLUGINNAME.min.js`

     These are the core plugin files: the uncompressed, packed, and minified versions respectively.
   - `changelog.txt`
   - `demo` (directory) contains examples and sample plugin implementations.

8. To see how everything works, unpack all of the files into a new directory and open the `demo` directory on a web browser. Right after the URL, add `example.html`.

For example, if the path to the `demo` directory is:

```
http://localhost/~user/jquery-validate/demo/
```

It should then become:

```
http://localhost/~user/jquery-validate/demo/
example.html
```

9. The example page should now display, and we are presented with a form (in this particular case) that we can play with and check whether the plugin is working properly.

10. We can even enter data in the fields and see that, if we click on the **Submit** button with some fields blank (or containing invalid information), a notice appears to warn us of errors in the data.

11. But how does this happen? Having a look at the source code for the example page is extremely helpful and useful to our path to knowledge.

To see a page containing the source code of the current document, we would normally press *Ctrl + U* in most browsers, or search through the **View** menu for an option named similar to **View source**.

12. Nothing should look new to us, as it's just plain HTML code with some JavaScript mixed in it.

It's interesting to notice, though, that for a plugin to work, it is necessary to link to the jQuery library and the plugin main file.

In this particular example, we find the following lines near the top:

```
<script src="../lib/jquery.js" type="text/javascript"></script>
<script src="../jquery.validate.js" type="text/javascript">
</script>
```

Plugins work only if jQuery has already been included!

13. Also, there's another thing calling for our attention:

```
<script type="text/javascript">
$(document).ready(function() {
        $("#commentForm").validate();
});
</script>
```

We immediately recognize the "document ready" function and notice just one line of code that is actually required to make the page behave in the user-friendly way that we experienced just moments ago.

Actually, the plugin relies on the assignment of certain classes to each field, in order to determine which validation rules are to be applied.

14. As a side note, a shorthand version of the above code is often used and it reads:

```
$(function() {
    // do anything here
});
```

15. Many plugins work in this simple way, and are thus programmed keeping in mind ease of use and unobtrusiveness above all.

Obviously, there are often more functions the user can play with. Detailed instructions are sometimes packaged along with the files or can be found online.

What just happened?

Thanks to this first approach to the plugin universe, we got a pretty good understanding of what the plugins we will write should look like in terms of look and feel.

It's no secret that simple but versatile and extensible code is easier to maintain, extend, and understand for others (and eventually for ourselves after a couple of years). Modularization is what jQuery coding standards evidently lean towards. These standards prefer a clean and comprehensible code flow to a more chaotic, messy, and obscure program that would only result in people getting confused and ultimately not using a program that, although perhaps even superior in terms of options, lacks what is fundamental for a program: clarity.

Plugin developers follow along, trying to make their creations as easy to use and understand as possible, providing a simple way to make use of the functionalities of their products.

It should also be clear by now that the jQuery website is the main resource for every jQuery enthusiast and plugin developer, as it groups nearly everything dealing with the library, and provides an excellent documentation for those needing help.

Have a go hero – get another example running

Just as we did before, download at least another plugin of your choice and have a look at the files.

See it working and check the source code to look for similarities to the Validation plugin source code. Write down what you find as part of plugin implementation and what is only related to jQuery methods.

Eventually, take a look at what's inside the plugin code files.

Time for action – setting up our own page

We'll see in detail how to set up a working page using the Wayfarer tooltip plugin, which is a simple but rather complete and reliable script to create tooltips (that is, whenever we hover the mouse pointer over some text, a box pops out and provides some text or information).

You can download the Wayfarer tooltip plugin from its official page in the Plugins directory on the jQuery website:

`http://plugins.jquery.com/project/wayfarer-tooltip`

1. Unpack the files contained in the plugin archive into a new directory and make sure that you just keep what you really need: a copy of the jQuery library, the plugin's own files, and a new HTML file that we have to create.

   We should end up with the following files:

   - `index.html`: The HTML file on which we will work
   - `jquery.js`: The jQuery library
   - `jquery.wtooltip.js`: A plugin file containing code

2. Edit the `index.html` file, and add the links to both `jquery.js` and the `jquery.wtooltip.js` file.

   Don't forget the "document ready" statement (empty for now, but we're getting there!). The file would look similar to the following:

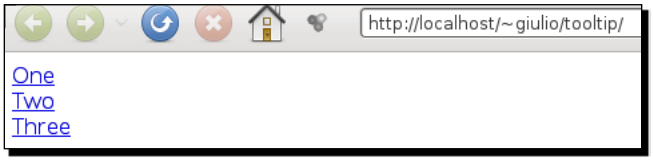
```
<!DOCTYPE html>
<html>
<head>
<title>Wtooltip test</title>
  <meta http-equiv="Content-Type" content="text/html;
    charset=utf-8">
  <script src="jquery.js" type="text/javascript"></script>
  <script src="jquery.wtooltip.js" type="text/javascript">
  </script>
  <script type="text/javascript">
    $(document).ready(function() {
      // code will be put here
    });
  </script>
</head>
<body>
</body>
</html>
```

3. Inside the `<body>` tag, add a few links to experiment on.

```
<a href="http://one.com" id="one">One</a>
<br />

<a href="http://two.com" class="link">Two</a>
<br />

<a href="http://three.com" id="three">Three</a>
```

Also note that this plugin is buggy when used together with the native `title` attribute. This is the reason we should rely on specifying the tooltip content as shown in the next steps.

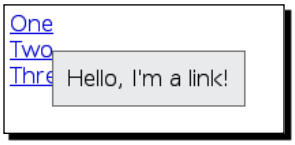
4. The page looks quite awful so far: a simple white page with three links and nothing more.



5. Browsing the documentation, we find out that we can actually specify a `content` option to create custom content:

```
$("a.link").wTooltip( {content: "Hello, I'm a link!" } );
```

The second link now shows the following tooltip when hovered upon:



6. Also, we had some dislike for the style of the tooltip. The documentation states that there's the possibility to specify an ID and/or class for the tooltip and use CSS instructions to change the look and feel of the elements:

```
$(document).ready(function() {
  $("a").wTooltip();
  $("a.link").wTooltip({ content: "Hello, I'm a link!" });
  $("a#one").wTooltip({
    content: "Hello, I'm a cool link!",
    id: "coolId",
    style: false // Removes all preset inline styles
  });
});
```

Following is the CSS code to obtain a colorful tooltip (far from artistic, though):

```
<style type="text/css" media="screen">
  #coolId {
    background-color: blue;
    padding: 5px;
    color: yellow;
  }
</style>
```

7. Hovering the mouse pointer over the first link will now result in a (supposedly) nicer tooltip:

What just happened?

This has been our very first attempt to modify the default behavior of a plugin, to better understand what we need to keep in mind when developing one.

During the course of the latest *Time for action*, a few key things should've got our attention—more than others, at least.

Options are used to pass to the plugin additional information to process; they are to be specified right after the call to the main plugin function.

There is no trouble passing one option; passing two or more is just a little bit different: as we have just seen, a comma (,) separates two options and values are specified right after a semicolon (:).

The last option does not need a comma at the end.

Options are written inside curly brackets ({ , }). Those quite familiar with JavaScript should have immediately thought of objects!

Structure of a plugin

Before diving into plugin programming, it could be helpful to have a look at what a jQuery plugin actually looks like.

Until now, we've just dealt with the methods alone, calling the plugin's own functions and making it work with elements in the page.

Types of plugins

jQuery plugins can be divided into groups depending on their function.

Most jQuery plugins fall into the category of jQuery selection functions (`$.fn.xxx`), which allow to perform an operation on a set of elements.

However, we can also provide additional standalone functions and objects (`$.xxx`) or create custom selectors (`$.expr.filters.xxx`) and animations (`$.fx.step.xxx`).

By having a look at jQuery's documentation pages about plugin authoring, we can find a simple code snippet for a log-writing plugin.

Browsing the excellent documentation freely available on the jQuery website is not only, obviously, helpful, but sometimes is fundamental to learn how things actually work.

For this example, we are accessing the web page located at `http://docs.jquery.com/Plugins/Authoring`—a simple search will find it anyway.

Time for action – types of plugins: Function plugins

The difference between function plugins and method plugins may be unclear at first, but we'll see they actually work in really different ways and are not used to accomplish the same kind of task.

1. In order to create our function plugin, which will be able to report messages, we need to create a new file and name it `jquery.log.js`.

Note that `jquery.PLUGINNAME.js` is the recommended file name convention for jQuery plugins.

We can obviously name it in any other way we'd like, but in case we are interested in submitting our new plugin to the official jQuery plugin repository, correct file naming is very welcome.

2. Write the following code from the documentation page and then save the file. This is our plugin. And as such, we should be able to recognize all the essential parts it presents: first of all, we can immediately state this is a standalone function (`$.xxx`), which does not return a jQuery object.

Also, the anonymous function takes one argument (`message`), which is required, and notifies the user by either logging the message to the console (if there is any console) or using an alert box.

```
jQuery.log = function(message) {
if(window.console) {
    console.debug(message);
  }
  else {
    alert(message);
  }
};
```

3. Create a new HTML page, and set it up as you would do for any other plugin, including the `jquery.js` file as well as our new plugin file `jquery.log.js`.

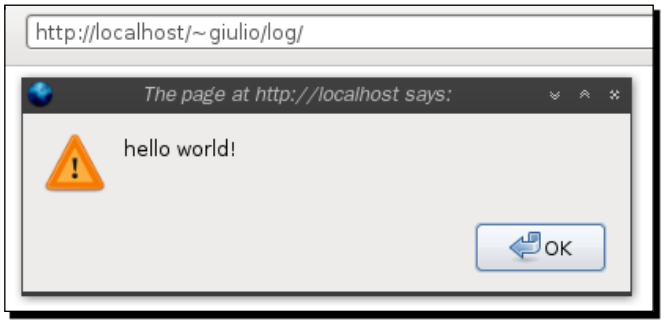
```
<script src="/path/to/jquery.js" type="text/javascript"></script>
<script src="/path/to/jquery.log.js" type="text/javascript">
</script>
```

4. Inside the "document ready" statement, add the following line:

```
$.log("hello world!");
```

   Then save the page.

5. Open the page in a browser; the message should now display!



What just happened?

We all know the dollar symbol (`$`) is a synonym to the `jQuery` word.

This means that the following lines are different ways to say the same thing:

```
$.log("hello");
jQuery.log("hello");
```

The only difference resides in compatibility (and possible ambiguity): while the dollar sign is mainly used in scripts and such, for plugin writing (code inside plugins) the `jQuery` word is preferred (unless we make use of closures, as described in the next chapter).

The reason is that the dollar sign is common to many other frameworks and libraries and `jQuery` has been chosen by jQuery authors to avoid compatibility issues—so that nothing breaks if other libraries are loaded as well.

When writing plugins, we want to make sure we are using jQuery's own methods and functions, as other libraries may (and definitely do not) work the way we are expecting them to behave. The `jQuery` word always (of course) refers to the jQuery library and, therefore, does not interfere with any other object that we may have loaded either in error or from necessity.

The other thing worthy of mention is the fact that we are used to calling methods attached to an element, perhaps using that handy chainability feature and queuing dozens of them.

But in this case, we have created a new function that does not operate on a collection of objects, and as such is attached to the jQuery object.

If we added a method instead, we would have called it with something like the following:

```
$.log("hello world!");
```

This results in the very same output.

Methods and functions

All new functions are attached to the jQuery object, and are not chainable.

On the other hand, new methods are attached to the `jQuery.fn` object, which allows chainability.

Have a go hero – exceptions logging

With the help of JavaScript and the jQuery documentation pages, create a new document and use the try–catch syntax to make the above plugin work and display the (eventual) exception.

Try using both the dollar sign (`$`) syntax and the `jQuery` object to check that there is no difference between the two.

Time for action – types of plugins: Messing with methods

Method plugins are one of the most popular approaches to jQuery plugins and can also be chained to other methods to perform more than one operation at once.

1. We're now going to create a selection plugin that changes background colors. To do so, first create a new file called `jquery.bgcolor.js` into the same directory as the log plugin.

   We should have three files now, plus the `jquery.js` file eventually.

2. Open the newly created file, and paste in the following code:

```
jQuery.fn.bgcolor = function() {
  return this.each(function() {
    $(this).css("background-color", "orange");
  });
};
```

3. Edit the HTML document (`index.html`), and make sure you link to the `jquery.bgcolor.js` file as well, after the `jquery.log.js` link.

We're starting to get quite a lot of scripts here. Let's have a look at what we're linking to and why:

- `jquery.js`: This is obviously required and fundamental. Nothing would work without the main jQuery library file.
- `jquery.log.js`: We are going to use this function in our next example, too; leave it untouched.
- `jquery.bgcolor.js`: Our new plugin, the one we're going to code next.
- Inline script with document ready statement and other miscellaneous JavaScript code.

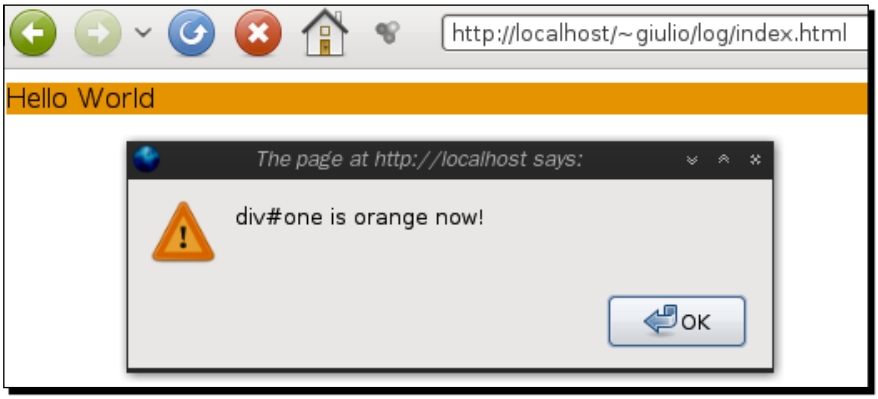
4. Write some HTML code in the document body. Make sure you have a valid element in it and every element is given a unique ID. For example:

```
<div id="one">
<span id="two">Hello World</span>
</div>
```

5. We can now add the following to the "document ready" function to make the DIV turn orange when the page loads and warn us with a pop up message:

```
$(document).ready(function() {
  $("#one").bgcolor() ;
  $.log("div#one is orange now!");
});
```

6. Finally, browse to the page URL and see our plugins in action! The DIV should now be orange, and a message should let us know the change that has just been made.



What just happened?

As we have just seen in practice, methods can be attached to elements present on the page, whereas functions are directly attached to the jQuery object and are in fact accessed differently.

When dealing with methods, it's important to remember a few points:

- Within the `this.each` loop, `this` refers to the current element being looped over.
- Outside the loop (thus, in the outermost function), `this` refers to the current instance of jQuery, that is, the entire element chain.
- The method must return the jQuery object to support chainability. Note that function plugins do not need this, as they won't be chainable.

The `this.each` loop is a means to go through all the matching elements on the page and ultimately operate on each of them. This means if we had two HTML elements identified by the very same tag (for example, two `span` elements), and we used the `$("span")` selector, we would end up operating on the first `span` element first, and then on the second one, letting ourselves wonder why we didn't use *ids* or *classes* instead (unless, of course, we wanted to affect all span elements!).

Have a go hero – more colors

Modify the background-colorize plugin so that the user is able to specify the color they want the background to turn into.

The user should be able to write:

```
$("div").bgcolor("red");
```

and see the DIV turn red, or:

```
$("div").bgcolor("blue");
```

and see the DIV turn blue.

Time for action – chaining

The concept of chaining, common in many scripting languages, such as Perl, is a very important aspect of jQuery development, as it allows different methods to be executed on the same selection with ease.

1. With the plugin created earlier, and modified so that a certain color can be selected and used as the background color for the element, we can now try chaining two or more calls to the same method with different colors and see what happens.

2. Set a new page up as we're used to, including all the necessary files. Inside the "document ready" statement, use the plugin on some element, like this:

   ```
   $("#id").bgcolor("red").bgcolor("blue");
   ```

   Note that it will display a blue background, as that is the latest one we have set.

   If we had used two plugins with two different functions, and had chained them, we would have seen both effects, as normally expected.

3. For example, try to use the plugin together with any other method that may modify the font size or color:

   ```
   $("#id").bgcolor("red").css({ 'color': '#123' });
   ```

4. We can now try to modify the script so the plugin does **not** return the jQuery object. This way we cannot chain methods, as, after the first call, the other methods don't have an object to operate on.

To prevent our plugin from returning the jQuery object, removing the word `return` is, in this case, enough. The script will run correctly and prevents the jQuery object from being returned.

5. If we try to re-run the above code, we will either get an error (if we have any sort of console installed and enabled) or just see the script is not working.

What just happened?

As we understood earlier, chaining is allowed only if the method we are trying to attach another call to does return the jQuery object.

The mechanism should be quite clear by now: if we don't return something, the other method doesn't know what element, object, or whatever it should work on as it's presented with nothing at all!

It's common practice to always return the jQuery object in every plugin that is intended to deal with chaining or is to be put in other code, or do tasks people would chain things with, so that nobody has to guess whether they can attach something else to the function or not.

As we're getting into creating more complex plugins, we'll probably completely forget about function-based plugins, as we'll deal mainly with element modifications and enhancement. Plugins of this type are often explicitly stated to allow for chainability as it's not uncommon to chain functions as much as possible.

The reason of this massive chaining is, above all, the reduced time needed to run the script. In fact, it's been calculated that the time a script with no chains runs is notably longer than the time needed to run the very same script applying chaining, since the code doesn't have to relocate the affected elements each time.

Other than that, chaining is often considered a better and cleaner way to write code. Even those who are glancing at the source code of a script written taking advantage of chaining understand how things work. However, by looking at a no-chaining script, one can get confused more easily, as the code can be messy and calls might not all be in one place.

Have a go hero – put it all together

We have two plugins now: one that somehow reports messages, and the other one that is able to change the background color of a chosen element to a color of our choice.

Wouldn't it be nice if, upon changing of the color, the plugin told us what elements were affected by the change and what color was selected?

Try to implement a sort of logging system into the background colorizer plugin, making sure you don't use just a simple *alert*, but make use of the log plugin instead.

Basic plugins examples

Having downloaded and played a bit with different plugins, we may now wonder what we are actually able to accomplish with plugins.

The answer is, almost everything!

As we have experienced, writing plugins is actually quite simple and they are easily modifiable to meet new criteria we may come up with even at a later time.

However, our experience has been with smaller plugins only. We still need to jump and move on to bigger and more complex plugins, similar to the ones popular at the moment that contribute greatly to simplifying the use of jQuery and JavaScript in general.

In fact, depending on our ability and the quality of our code and ideas, we can take advantage of the jQuery API in many different ways, and develop plugins that help us dealing with **forms** (as seen earlier), **windows** (that is, resizing, moving, whatever), **media** (displaying images, playing videos and sounds), **user interfaces** (making the Web a nicer place), menus (dynamically creating drop-down menus, sorting entries), and pretty much everything you can access through the DOM.

Apart from the "field" our plugin will deal with, it's extremely important to follow the suggested guidelines the jQuery team (and the community itself) has set, to make it easier to find and correct possible errors and incompatibilities.

Those plugins we have developed and messed with earlier in this chapter are just simple examples to show how things work in practice. The realization of an advanced plugin full of functionalities and innovations is far from the aim of these first few chapters. We're now establishing the basis for what will come, shaping our mind so that it's now able to reason in a plugin- and object-oriented way, and starting to scratch the surface of coding standards and plugin structure and usage.

A few key things to remember

Very quickly, here are some key things that will come in handy later, when we look back for some reference on this topic.

For every chapter, we should write down the most important things that we think will help us and/or are of fundamental importance at any moment.

- jQuery documentation is gold. Always go back to the documentation pages when in trouble or in need of information. Reading it thoroughly wouldn't hurt either.
- This might sound stupid, but always remember to link to the `jquery.js` file (containing the jQuery library) or, no matter what, you might spend hours looking for some error that justifies the script not running or working properly.
- Prefer, when possible, the use of a centralized copy of the library that can be cached. Google (`http://code.google.com/apis/libraries/devguide.html#jquery`) offers an online version of some of the most popular libraries. The more people use this, the more chances are that visitors to your website have already downloaded jQuery and have the impression of the page loading faster.
- Always prefer the "document ready" statement to any other non-jQuery functions to check whether the page has already loaded or not.
- This can never be stressed enough: plugins based on methods are completely different to plugins based on functions.
- Method plugins extend the `jQuery.fn` object.
- Function plugins **directly** extends the `jQuery` object.
- Method plugins **do** support chainability.
- Function plugins **do not** support chainability.
- Method plugins should **always** return the `jQuery` object (`this`, in the code), to allow for chainability.

Pop quiz

1. The jQuery code is contained in the `jquery.js` file. When do we need to make sure we have added a link to the library in our HTML documents?

   A) Whenever we deal with forms.

   B) Only if our scripts make use of some hidden functionalities of jQuery.

   C) Whenever we need jQuery (either for basic scripting or plugins).

   D) It's not required. We can check if we remember, as everything would work just fine anyway.

2. jQuery guidelines state that plugins should be named in a certain way, to prevent everybody from deliberately making up meaningless and misunderstandable file names. Provided `PLUGINNAME` is the name of a plugin, how should the file be named?
   A. `PLUGINNAME.js`
   B. `PLUGINNAME.plugin.js`
   C. `jquery.PLUGINNAME.js`
   D. `PLUGINNAME.jquery.plugin.js`
3. What does the term chainability refer to?
   A. To the possibility of chaining CSS selectors to better operate on certain elements.
   B. To the possibility of chaining functions.
   C. It's not important: this can be forgotten with no problems.
   D. To the possibility of chaining methods.
4. Which type of plugin extends the jQuery object?
   A. Method plugins.
   B. Function plugins.
   C. The jQuery object cannot be extended.
   D. Any type of plugin.
5. Which type of plugin extends the `jQuery.fn` object?
   A. Method plugins.
   B. Function plugins.
   C. The jQuery object cannot be extended.
   D. Any type of plugin.
6. Which type of plugin should always return the jQuery object and why?
   A. Method plugins, so that methods can then be chained whenever needed.
   B. Function plugins, so that functions can then be chained whenever needed.
   C. There is actually no need to return this object, as it's hardly used in plugin development, after all.
   D. Any type of plugin should return the jQuery object, since it's specified in the jQuery plugin authoring guidelines.

7. Which type of plugin have we developed if we are able to chain it with other jQuery built-in methods?

   A. It's a method plugin, because method-based plugins are the only ones that allow chainability.

   B. It could be a function-based plugin, as well as a method-based one.

   C. It's hard to say with no additional information on the type of object it returns.

   D. It's a function-based plugin, because function-based plugins are those that return the jQuery object—and thus permit subsequent concatenations.

8. Inside the `each()` loop, what does the word `this` stand for?

   A. It cannot be used at all: as a reserved word its usage is only permitted when writing scripts directly on the HTML document, inside the document ready statement.

   B. Nothing actually. It's a simple way of making the code look more elegant and nicer to look at.

   C. `this` is another way of referring to the jQuery object, whenever we are tired of writing `jQuery`.

   D. `this` is another way of referring to the current element chain.

9. Is the `this.each` loop necessary for plugins, and what does it do exactly?

   A. It's probably a necessary part for most plugins, even though it's only needed by the jQuery core for some internal routines and data handling.

   B. It's required for the plugin to work properly, because the code wouldn't run if the loop was missing.

   C. It loops through all of the elements that match the CSS selectors and, one after another, applies the same instructions specified in the method or function.

   D. It is not required and can be omitted if we need more memory to process a large amount of data.

10. When writing a plugin, what do the following symbols or words mean?

    `$`, `this`, `$(this)`

    A. Alias to jQuery object, current element being processed (jQuery object), current element being processed (DOM object).

    B. Alias to jQuery object, current element being processed (DOM object), current element being processed (jQuery object).

    C. Current element being processed (DOM object), jQuery object, alias to jQuery object.

    D. jQuery object, current element being processed, alias to jQuery object.

Summary

This chapter was all about plugins basics, proper usage, and problem solving.

We had a relatively quick overview of how plugins are developed, from scratch to an easy-to-use and pretty intuitive tool users are able to make use of without many troubles.

Despite our little start in plugin authoring with a couple of tiny, with no expectation and limited snippets of code, we managed to go through some code listings and understood the difference between different types of plugins—some useful in certain situations, some that are helpful in others.

However, apart from the obviously important rant about various plugins and their applications, it's worthwhile to notice the fundamental role simple jQuery statements, functions, and methods play in plugin coding. It's all based on the jQuery library after all!

This is the very first occasion one bumps into to either prove oneself a professional jQuery plugin developer or an amateur programmer just starting out.

Either case, the goal is to get to the end with a good grasp on writing plugins that deal with different fields of application—and we'll get to the end in some way!

After having our first nibbles of jQuery plugins, in the hope that everything will become more interesting and tastier, the next chapter covers the practical realization of a bigger, larger-scale plugin.

Nothing to be scared about, though; we'll just put into practice what we've learned so far, with a little, yet decisive, twist to make the whole look better and behave more to our liking.

We'll learn how things really work in detail, with particular focus on the versatility and the final goal of our very first plugin realization attempt, which will be, even though not so tricky as you might expect, of certain interest—especially if we're keen on understanding how every single component cooperates with others to make the whole, bigger picture stand out.

3
Our First jQuery Plugin

Now that we have got some first experiences dealing with plugins, we'll start off with our plugin creation adventure.

From now on, each chapter will cover a specific topic related to plugin creation, meaning each chapter will be focused on developing a particular type of plugin (that is, media plugins, form-related ones, and so on).

The current chapter, however, deals with the correct creation of a plugin of any sort, to sort things out and make clear, once and for all, the basic outline of a plugin and what is fundamental for successfully developing a plugin from scratch.

This chapter will be about the following topics:

- Defining our own default plugin structure
- Setting the basics for our first plugin
- Getting a step farther
- Dealing with options
- Using functions inside the plugin
- Closures: making functions private

Defining our own default plugin structure

To make things easier to remember and apply, we are going to start off with the definition of what we will be referring to when speaking of the basic template that all the plugins we are going to develop should conform to.

Actually, we have already had a quick look at it earlier in the previous chapter, but there's something more definitely worth saying.

From now on, we will call the following code the default structure for our plugins. This is what we will promptly copy and paste into each file we're going to write a plugin into.

```
jQuery.fn.PLUGINNAME = function() {
  return this.each(function() {
    // code
  });
}
```

Needless to say, the `this.each` loop iterates all of the matching elements. We return the jQuery object (`this`) to allow chaining. We extend the `jQuery.fn` object; all of the code will be put inside the function.

Also:

- The file name of every plugin we're going to develop will be `jquery.PLUGINNAME.js`.

  For the moment, remember to always avoid referring to the jQuery object with the dollar sign (`$`), to prevent possible conflicts with other libraries. We'll get to using aliases very soon.

- All of the functions that we write to make our plugin work should be private and not accessible from outside, in an attempt to avoid cluttering and possible backwards incompatibility.
- If not from the very start, at least at the end, a user will be able to specify options to control the plugin behavior.
- Default options for the plugin will be publicly accessible to allow for easier customization with minimal code.

The directory that the plugin resides in will also contain two other files, by default:

- `index.html`: This is our test page.
- `jquery.js`: This is the jQuery library that we need to make things work.

Setting the basics for our first plugin

As our first plugin, we might want to create something uncomplicated but somewhat impressive: what about something that, when the cursor is hovering over an element, substitutes the text contained with some words of our choice?

Time for action – our first plugin, Part I

Getting started in creating jQuery plugins in not difficult at all. For example, creating this simple plugin should help us in understanding how things actually work.

1. Given that our plugin name is *txtHover*, create all the directories and files we need by default, and copy over the `jquery.js` file.

2. Copy the default structure for our plugins to the plugin file and make sure the function is named accordingly. It should look like this:

```
jQuery.fn.txtHover = function() {
  return this.each(function() {
    // code
  });
};
```

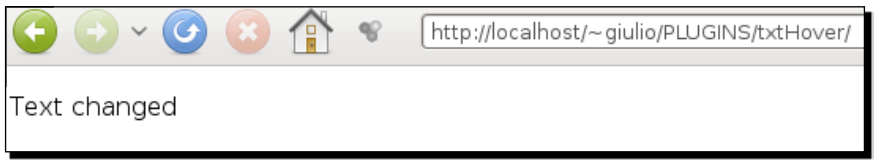
3. Nothing's easier to do than change the text contained in some element. Inside the plugin function, write the following to let the trick happen:

```
jQuery(this).text("Text changed");
```

4. To test this in action, we can modify the HTML document to look like this:

```
<!DOCTYPE html>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="jquery.txthover.js"></script>
  <script>
    $(document).ready(function() {
      $("p#one").txtHover();
    });
  </script>
</head>
<body>
  <p id="one">Some text.</p>
</body>
</html>
```

5. Unfortunately, the result is neither fancy nor satisfactory—something we've experienced earlier too. But we're just getting started; we won't stop here this time!



What just happened?

To tell the truth, there's nothing new in this piece of coding, as we've already analyzed the previous code (or, at least, some very similar snippets) several times before.

The plugin is working correctly so far. When the page loads, the text is changed to what we had defined into the plugin code.

However, there are a couple of things to pay attention to:

- The function `text()` from the jQuery API expects either one or no arguments to be passed: if there is no argument the function returns the current content of the selected element. The text string passed as an argument substitutes the element text otherwise.

  There are, however, some similarities with the `html()` function, which treats the text it operates on as if it were HTML code. That is, passing any tag to the `html()` function results in having the element possibly containing other elements after the operation (also, the same applies for getting HTML code from within the element), whereas the `this` function will just treat the HTML code as plain text and not affect any element hierarchy.

- The fact that the text cannot be changed, unless we directly modify the code of the plugin.

Getting a step farther

Despite the good realization, our plugin still misses the point. Our goal was to activate the text substitution whenever the mouse pointer hovered over the text to be replaced, whereas our current implementation does it right after the page is loaded.

We put it inside the "document ready" statement, after all!

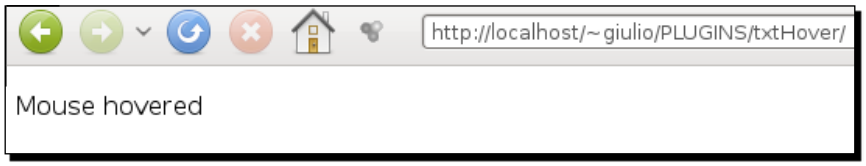
Time for action – our first plugin, Part II: Hovering

By adding little pieces of code one at a time, we can easily understand what we are going to do and which is the best layout for our plugin. Activating the plugin whenever the mouse pointer hovers over the selected elements is surely another little step that adds up to reach our final goal.

1. Back to our plugin file, we have to change the code so that the whole mechanism activates when the hover event is triggered. jQuery provides a function called `hover()`, which we can use to achieve our objective:

```
jQuery.fn.txtHover = function() {
  return this.each(function() {
    jQuery(this).hover(function() {
      jQuery(this).text("Mouse hovered");
    });
  });
};
```

2. Now, on to testing. Once the mouse pointer hovers over the text, it is effectively replaced by our string. But even if the mouse is moved, the text doesn't revert to the original.



3. In fact, looking at the hover documentation, we see the function can also take a second argument, that is, the pointer to a function to be executed when the mouse goes off the element.

   Our modified code will now look like the following:

```
jQuery.fn.txtHover = function() {
  return this.each(function() {
    var oldTxt = jQuery(this).text();

    jQuery(this).hover(function() {
      jQuery(this).text("Mouse hover");
    }, function() {
      jQuery(this).text(oldTxt);
    });
  });
};
```

4. The result is somewhat better now: the text is changed when we leave the pointer on the paragraph, and is changed again to its original form once the pointer is moved away.

What just happened?

We might be a little more satisfied with this evolution of our plugin, even though it's far from complete.

Its functioning is fairly straightforward: we have made use of a variable (`oldTxt`) to store the old content, and we then have proceeded to using two anonymous functions (`function(){ }`), passed as arguments to the `hover()` function, to handle the mouse hover event (write our string) and the mouse out event (text back to original).

There's still something to do though:

- We have used far too many instances of `$(this)` in our code and, on a larger scale application, a lot of memory would be wasted this way. It will be incredibly better for performance to make up a variable and use it every time we refer to the element with `$(this)`.
- The text cannot be changed, unless we directly modify the code of the plugin.

Have a go hero – html() versus text()

Read the documentation for the `html()` function.

Create a plugin that, once the mouse pointer hovers over an element, displays the HTML code of its content. The content should then change back to normal once the mouse pointer is moved away from that space.

What happens if the `html()` and `text()` functions are used one inside the other, that is, `$(sel).text($(sel).html())` or `$(sel).html($(sel).text())`? Just play around a little bit.

Dealing with options

An aspect that we left out in the creation of our plugin is the one concerning options and, thus, customization.

Implementing options not only will drastically increase the flexibility and ease of customization of the plugin, but will also quicken later modifications by the author himself or herself, as it results in the whole code being simpler to understand and the important bits would be all grouped together.

Time for action – our first plugin, Part III: Options

To allow for options to be specified and the behavior of our plugin to be customized and modified to users' liking, we need options.

1. The code we use for our plugin will be modified to look like the following:

```
jQuery.fn.txtHover = function(options) {
  var defaults = {
    txt: 'Mouse hover'
  };

  var o = jQuery.extend(defaults, options);

  return this.each(function() {
    var e      = jQuery(this);
    var oldTxt = e.text();
```

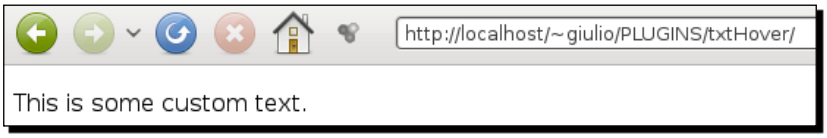
```
    e.hover(function() {
      e.text(o.txt);
    }, function() {
      e.text(oldTxt);
    });
  });
};
```

2. We are allowed to change the `txt` option now. In case we don't, though, the default value will be used (`Mouse hover`).

   The document ready statement should now contain the following:

```
$(document).ready(function() {
  $("p#one").txtHover({ txt: 'This is some custom text.' });
});
```

3. And the result would look something like this, with the text changed to whatever we like:



What just happened?

The structure we've just used to get our plugin packed with options is quite flexible and easily extensible.

We first made sure the function definition would include an argument (that is, options) for settings to be specified. The `options` object must be formatted following the inline object creation method (that is, `{ option1: value1, option2: value2 }` and so on).

But what would happen if the user does not specify any options? Or, if we are dealing with a number of options, wouldn't writing the same lines of code everytime to reset option values be incredibly annoying?

Instead, we create a new (plain) object (`default`) in which we store the default values of options we plan on accepting as valid. This way, users may check if the default value is acceptable to them and either leave it untouched (the plugin will then use the default value) or change it to their needs, resulting in the plugin making use of the customized value for that particular option.

The very next line is the way jQuery is instructed to combine the two objects. If any of the options we are going to make use of later on in the code (and want the user to be able to modify) is not passed to the function, we use the default one instead.

What we'll get at the end is another object (`o`) with the values we need for the plugin to work with. From this moment onwards, we'll use this object's values to access options we need during development.

The rest of the code is left untouched, except for the `o.txt` part that replaces the old string. Users are now able to choose what they want to replace the original content with.

Also, the keyword `$(this)`, much abused of in the previous version, has been replaced with the variable `e`, into which all of the information of the element has been copied. In a large-scale environment, this little tweak will result in a huge boost to the script performance.

Have a go hero – adding colors

We have seen in the previous chapters how accessing and modifying an element's CSS properties through the `css()` method is not only possible, but quite simple and straightforward too.

The `css()` method works very much like the `text()` and `html()` ones do, with the difference being that it always expects at least one argument to be passed, as the first parameter is responsible for selecting which property we want to get access to.

Using your knowledge about the above-mentioned method and what you've just learned from the latest *Time for action*, extend the plugin so it accepts two other options: `fgColor` and `bgColor`.

Needless to say, these two options make the users able to change the element's background and foreground color whenever the mouse pointer hovers over the element. When the mouse pointer gets off the area, colors should revert to original.

Keep in mind that, if either of those two options is not passed, the user does not intend to have the color(s) changed at all and they should appear as they already are, not just as black text on white background!

Using functions inside the plugin

Repetitive tasks are boring. And this is the reason functions exist. However, how do functions interact with the code of a plugin?

Very simply, every function declared outside the plugin's own code (note that defining a function right inside it is allowed) is accessible as it would be in any other piece of code.

Time for action – our first plugin, Part IV: Functions

For testing reasons only, we need a debug function to tell us what happens at certain points when the code runs. This will also help us in understanding when and how functions are accessible from a particular portion of code.

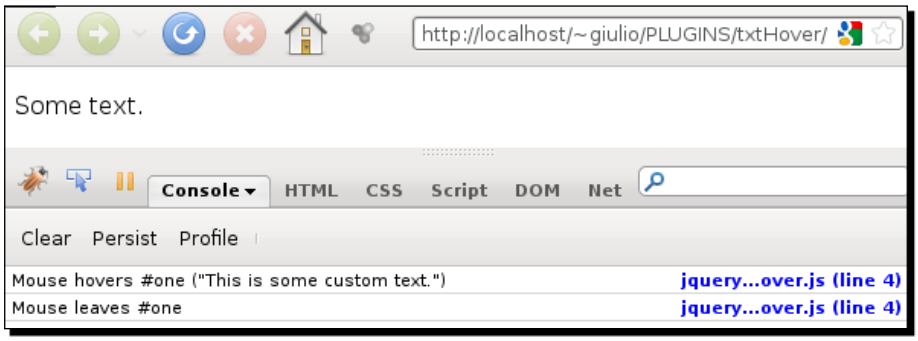
1. We could add the following function (from our previous log script) at the very beginning of the plugin file:

```
function _debug(msg)
{
  if(window.console) {
    console.debug(msg);
  } else {
    alert(msg);
  }
}

jQuery.fn.txtHover = function(options) {
  // the rest of the code
  // goes here
};
```

2. Our plugin would then read (with the addition of some debug facilities):

```
function _debug(msg)
{
  if(window.console) {
    console.debug(msg);
  } else {
    alert(msg);
  }
}

jQuery.fn.txtHover = function(options) {
  var defaults = {
    txt: 'Mouse hover'
  };

  var o = jQuery.extend(defaults, options);

  return this.each (function() {
    var e      = jQuery(this);
    var oldTxt = e.text();

    e.hover(function() {
      _debug ("Mouse hovers #" + e.attr('id')
               + " (\"" + o.txt + "\")");
```

```
        e.text(o.txt);
      }, function() {
        _debug("Mouse leaves #" + e.attr('id'));
        e.text(oldTxt);
      });
    });
};
```

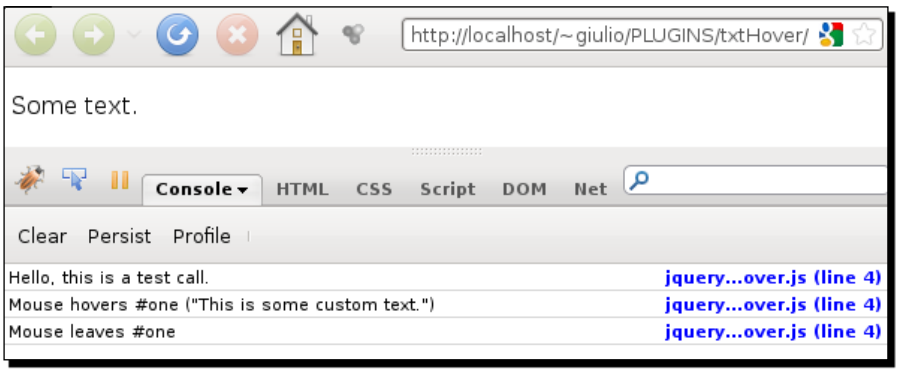


3. There's still some trouble, though: what happens if we use the debug function outside the plugin file?

We expect it not to be accessible or to be somehow unusable, as we have declared it inside the plugin file and it was intended for use inside the plugin only.

If we try to put a sample call right after the "document ready" statement, we can see the function to be actually accessible from everywhere in the code.

```
$(document).ready(function() {
  _debug("Hello, this is a test call.");
  $("p#one").txtHover({ txt: 'This is some custom text.' });
});
```



What just happened?

Dealing with functions shouldn't be much of a problem. We are supposed to be used to making heavy use of them.

However, the point is their accessibility from different parts of the code. At first, we create the function(s) we need without thinking about the number of problems that could arise if we do not protect them and struggle to make the functions we use in our plugin private (that is, not accessible from outside our plugin).

What we must be aware of are the issues we could bump into when defining a certain function in our plugin file without thinking that the very same name we originally thought would have been unique (for example, `_debug`) has been used somewhere else in the code to name a function whose goal is completely different.

The correct (or, at least, expected) behavior of our plugin is compromised and for no reason should we permit such a stupid error.

Another point of particular interest, in case it's not been noticed before, is the way the `o` object is used everywhere at any point of time that we need to refer to the current set of options, whereas `e` is the current element being processed.

Have a go hero – experimenting with functions

Play around with functions. Try to define some functions nearly everywhere it is possible and try calling them from other parts of the code to see what happens, which functions override which, and the difference between calls to functions with one or more arguments.

Also, make sure you notice where (that is, in what parts of code) a function can be defined and where, instead, it's not possible.

Is it possible to define a function inside the document ready statement? If so, what would it mean and is there any advantage?

Closures: Making functions private

With the aim of keeping our own functions, those needed by our plugin only, private, so that others cannot access and misuse them, we're going on to discovering closures.

Time for action – our first plugin, Part V: Closures

Closures make it possible for the developer (that is, us) to avoid creating functions in the main namespace and keep them private to avoid problems with naming, backward compatibility issues, and excessive cluttering.

1. We're now going to add a function to the `index.html` file, called `_debug`.

   This new function will take one argument (just like our plugin's one) and will do nothing more than just display an alert reporting some random text.

```
function _debug(argument)
{
  // Note we don't make use of the 'argument'
  // we just need it for the function to look
  // exactly the same as our plugin's debug function
  alert("Hi, this is the fake debug function.");
}

$(document).ready(function() {
  _debug("Hello, this is a test call.");
  $("p#one").txtHover({ txt: 'This is some custom text.' });
});
```

2. Now, on loading the page, how does the result appear? Is it what you expected? The newly created function completely replaced our plugin's function, making the behavior definitely not the same.

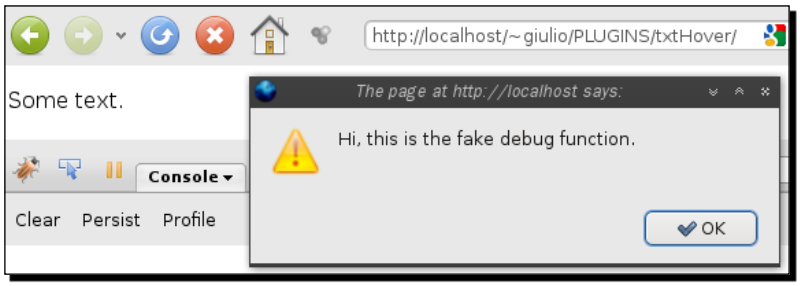
3. What would happen if we use closures to prevent this type of inconvenience from happening?

   Here is how closures work. This is how the plugin code should look now:

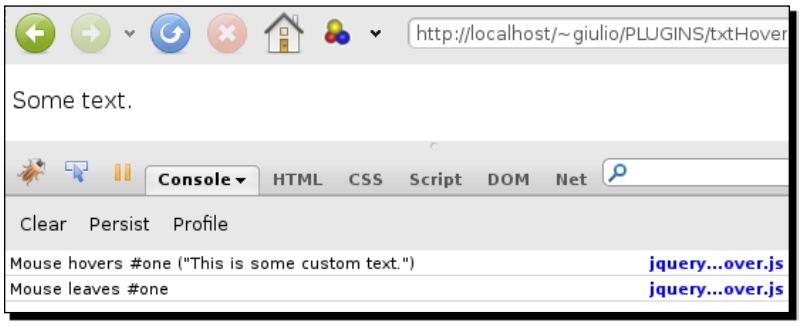
```
(function($) {
  function _debug(msg)
  {
    if(window.console) {
```

```
      console.debug(msg);
    } else {
      alert (msg);
    }
  };

  jQuery.fn.txtHover = function(options) {
    var defaults = {
      txt: 'Mouse hover'
    };

    var o = jQuery.extend(defaults, options);

    return this.each(function() {
      var e      = jQuery(this);
      var oldTxt = e.text();

      e.hover(function() {
        _debug("Mouse hovers #" + e.attr('id')
               + " (\"" + o.txt + "\")");
        e.text(o.txt);
      }, function() {
        _debug("Mouse leaves #" + e.attr('id'));
        e.text(oldTxt);
      });
    });
  };
})(jQuery)
```

4. And here is how the page looks now, after we have modified the plugin code. When the page loads, we get to see the pop up:



But when we hover the mouse pointer over the text we actually see what we expected to see:



What just happened?

Using closures is definitely a best practice that everyone should follow.

Not only does it allow functions to be defined for the plugin only (keeping them private and not accessible from outside the plugin's code), but many other advantages come as well.

The way in which closures operate is quite simple after all. By creating a closure, several things actually happen: first of all, closures create a function with `$` as its parameter.

Then that function is immediately called with the value `jQuery`, which gets mapped onto the `$`, letting us use the `$` sign without the fear of creating any kind of issue.

When we first defined the fake debug function, we probably didn't think it would've overwritten the whole content of the plugin's original function. Perhaps it would've given an error; or just the first call (the one right before the plugin call) would've been compromised.

Actually, this behavior is perfectly explainable. Looking at the file structure, we see the `jquery.js` file is included first, then comes the plugin, and finally the inline script part containing the fake function and the "document ready" statement.

This means the latest part quite naturally overwrites the earlier ones.

Thus, having defined the fake debug function at the very end, it obviously overwrites the old debug function coming before it.

The whole example would have been quite different if we had put the above-mentioned function before the inclusion of the plugin file, in which case the dynamics would have been worthy of notice.

In the latest version, the original debug function is kept private and thus the fake function cannot overwrite it, resulting in the correct behavior that we would've expected from the very beginning. As the page loads, a pop up is displayed with some text, whereas the hovering effect still works correctly thanks to closures preventing the functions declared inside them from being publicly visible.

Have a go hero – avoid conflicts

With closures in mind, try to replace all the `jQuery` words with the dollar sign, as if you where using normal jQuery syntax in a normal script.

If you even try to throw in some conflicts over the dollar sign, which is often used by other libraries as well (for example, by including another JavaScript library and using both to get something done), you can see the plugin still works correctly.

In fact, apart from keeping functions to themselves, this closure alias the word jQuery to the symbol `$`, acting as the `noConflict()` function would on the plugin code.

This is particularly handy because it allows the normal syntax, the one we're all most used to and that comes out without thinking, to be used with no worries at all.

Pop quiz

1. The dollar symbol (`$`) is a very handy way that we normally make use of in everyday scripting. We've just noticed it's not so good when speaking of plugin development, though. Why is that?
   A. It's an aesthetic reason, nothing less, nothing more.
   B. The dollar sign is going to be lacking support for plugin development in the upcoming version of jQuery.
   C. Just a matter of taste after all.
   D. Because of possible conflicts with other libraries that use the same symbol with a different meaning.
2. How many arguments does the `hover()` function take and what are they used for?
   A. It takes two arguments. The first is a callback to the function to call when the mouse pointer hovers. The second one is the callback for when the mouse pointer leaves.
   B. It takes two arguments. The first is a callback to the function to call when the mouse pointer leaves. The second one is the callback for when the mouse pointer hovers.

C. It takes one argument. It is a callback to the function to call when the mouse pointer hovers.

D. It takes one argument. It is a callback to the function to call when the mouse pointer leaves.

3. When cycling through a series of elements, what is the problem, if any, with using the same selector too many times (for example, using `$(this)` or `$("div span .sample")` repeatedly)?

   A. No problems actually.

   B. It's rather time consuming.

   C. It slows down the plugin, and using the very same selector many times is pointless. We'd do better to use a variable to which we assign the element selected once only.

   D. It takes up too much space and, when we're about to minimize the plugin, we may face trouble because of the excessive size of the script.

4. jQuery provides methods to access the content of an element: `text()` and `html()`. What's the main difference between the two, in terms of the value returned upon a call?

   A. No difference at all, `text()` is the old version of `html()` and, as such, is going to be flagged as out of date very soon.

   B. No difference at all, `html()` is the old version of `text()` and, as such, is going to be flagged as out of date very soon.

   C. `text()` returns the combined text contents of the element, whereas `html()` gets the HTML contents of the element.

   D. `text()` can be used with older versions of jQuery as well, whereas `html()` is for use with jQuery versions 1.4 and newer.

5. What do options, in terms of practical advantages, do for both the regular user and plugin author?

   A. They only provide more flexibility for the user, with no particular advantages for the developer, who actually has to struggle more to deliver a pointless feature a lot of people look for nevertheless.

   B. Options provide an easy and accessible way to change the plugin behavior and customize, tweak, and modify its interaction with the HTML document, for example, changing execution times, speed, and so on.

C. They only cause the user to be overwhelmed with lots and lots of useless settings to change, which often result in the plugin not working correctly and/or older versions not compatible with the newest ones.

D. No advantages whatsoever.

6. How can functions be used inside plugins, and are they available outside plugins?

   A. They're used just like they would be normally, and they can be accessed by other functions or blocks of code if we don't make them private somehow.

   B. Unfortunately, functions cannot be used inside plugins and we must find another way to avoid repeating code over and over.

   C. There are some required keywords to make use of when calling functions, or they won't work inside a plugin's code.

   D. Functions behave as they normally would, and are not accessible by outside code by default.

7. Closures are very important. Besides defining clearly where a plugin begins and ends, they also have other key functions that make them almost necessary for plugin development.

   In fact, what do we achieve by applying closures to our plugin's code?

   A. They only keep functions private.

   B. Closures help dealing with options from other plugins.

   C. They provide an easy way to access jQuery's API from within the plugin code, so that it's safer to use functions and aliasing the jQuery keyword.

   D. We can keep functions private and make use of the aliased dollar symbol as we would normally.

Summary

This chapter has covered, step by step, the creation of a simple plugin. At the very beginning, the plugin was really simple and far from being well thought out. However, by the end of the process we ended up with a simple plugin, capable of doing several things, and a structure evolving constantly after each step.

Perhaps without having noticed, we've taken care of event handling (for example, the hovering effect, with its hover and leave events), customization, modification, user personalization (making a set of options available and thinking about the default values), security, compatibility issues, and aliasing (making the plugin work within closures and having the possibility to use the dollar sign (`$`) again).

From the next chapter onwards, we'll put what we have learned until now into practice. We will develop various types of plugins whose aim is to provide a simple yet practical and professional solution to many topics that developers often have to deal with, when they're about to create a website and put it into the best condition to show off its content and design.

With special focus on the media aspect of the web, the next three chapters make a huge point on the creation of media plugins. These are those jQuery plugins that have something to do with various media types: namely images of all kinds, audio files (yes, music included), and videos—mostly from online streaming websites, which are the number one source for embedding short videos into web pages.

The next chapter is all about image handling and processing.

Our aim at the end of it is creating a simple image gallery, which is a great way to view images on the Web: big and quickly.

We will, just as we've done until now, have a step-by-step approach to the problem, thinking about possible solutions and solving problems we'll certainly face with the help of online documentation and previous examples.

4
Media Plugins: Images Plugins

We've just seen how to create a plugin from scratch—a real plugin. The plugin works in every little part of its code and actually does something that can be useful to somebody with little to no effort required in modifying the code. We made sure it allows for easy modifications and tweaks as well as a simple setup.

The fun still has to begin, though.

Wandering on the Internet, we've all come across some really neat plugins, each showcasing some fancy graphics, a number of surprising effects, hundreds of features, and incredible control of its application.

But the most intriguing thing, which should hit the plugin developers in their heart, is the code and how the fancy graphics and surprising effects have been realized.

Contrary to popular belief, there's nothing wrong in looking through the existing code to get inspiration and fresh ideas from the author. Shame makes its appearance whenever we copy (that is, copy and paste—Ctrl + C, Ctrl + V) that code instead of thinking about it and maybe finding a different, perhaps better, way of accomplishing the very same task.

This chapter will cover the following topics:

- Plugin overview
- Handling images
- Centering things
- Putting it all together

From this chapter on, we'll cover a series of topics related to plugin development (obviously!), one for each chapter, each focusing on a specific area particularly relevant and of relatively common usage.

For example, we're now going to analyze some types of plugins dealing with images. We'll create our very own Lightbox-like plugin to handle different thumbnails in the same document and display a real-size preview without the need to load another page.

In the next two chapters, we'll handle different media types (namely, audio and video files) instead and will have, by the end, a good understanding of how media files can be embedded and manipulated in a web page using the jQuery library.

Plugin overview

Of course, we cannot analyze in great detail each of the hundreds of image-related plugins available on the Internet. Instead, we'll focus on a certain type of plugin: preferring quality over quantity in our journey through plugins. We'll end up with a few plugins less, but we're sure each of them is complete and of great quality.

And, most importantly, we'll have understood how things actually work, making us able to create all of the other plugins with no trouble whatsoever. In fact, once we understand the basics, it's just a matter of time and practice to get the confidence required to develop complex plugins with more features and make them visually more appealing.

The plugin we're going to create is really close to (though simpler than) the Lightbox plugin. This type of plugin is so popular that it's almost impossible that anyone has never heard about it or seen it in action.

Basically, when an image is displayed on the page, a very handy possibility is offered to the user. Clicking on the (small) thumbnail will cause the image to get bigger and bigger until it reaches its original dimensions and is visible at full size on the very same page that the smaller one's found!

The image will thus overlay other elements on the page. Therefore, there is no need to load another page (or even make a blog page obscenely long) because of some pictures from our latest holiday.

Some versions also have the possibility to load another image upon the click of the mouse, so that a scaled-down copy of the picture can be displayed in place of the original and the page load times are sensitively reduced, especially when dealing with a discrete number of images.

Whenever the user clicks outside the image, on another picture, or on some close button placed nearby, the big photo disappears and the page is back to normal.

If you have never seen one of these plugins working, some of these web pages would be worth a visit. You will begin to understand how this whole thing can be done and what our final result is supposed to look like:

- `http://leandrovieira.com/projects/jquery/lightbox/`
- `http://static.railstips.org/orderedlist/demos/fancy-zoom-jquery/`
- `http://jquery.com/demo/thickbox/`

Now that we know what we are after, a quick summary of what the plugin needs to do, and what we need to keep in mind, is really necessary:

- Full-size previews must pop out at the center of the browser window, lying on top of other page elements.
- Images will disappear whenever we either click on another image to see its bigger version or click on the picture itself.
- A close button can be added later on.
- Effects (such as sliding, fading, and zooming) are no big deal to implement.
- Also, options should be made available for better plugin personalization.

The above points made clear, we can now proceed and get our hands dirty!

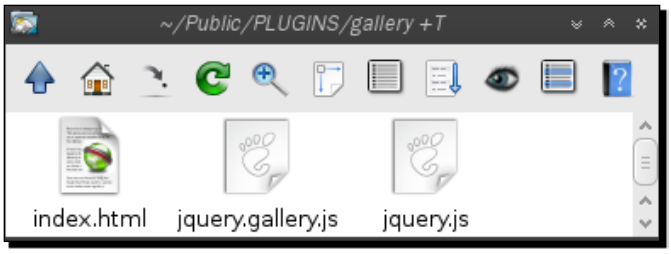
Handling images

The goal of our plugin is, so to speak, handling the selected images in the document and apply to them the styling and effects we like the most and feel appropriate to accomplish our preset task.

Time for action – showing images

Images play a big role today, and creating a plugin that is capable of manipulate image elements is a challenging task, which will result in a nice-looking and useful plugin.

1. First off, set up our default plugin structure. That is, create a new directory called something like "gallery" containing two files: `index.html` and `jquery.gallery.js`. Also, copy over the `jquery.js` file.



2. Paste the following code into the plugin file (`jquery.gallery.js`):

```
(function($) {
  jQuery.fn.gallery = function() {
    return this.each(function() {
      var e = $(this);

      e.click(function() {
        // show big preview
      });
    });
  };
})(jQuery)
```

3. Gather a couple of images from somewhere on your hard drive and copy them into the gallery directory.

4. Make the necessary modifications so that the HTML document `index.html` looks like this:

```
<!DOCTYPE html>
<html>
<head>
  <script src="jquery.js" type="text/javascript"></script>
  <script src="jquery.gallery.js" type="text/javascript"></
script>>
  <script type="text/javascript">
    $(document).ready(function() {
      $("img").gallery();
    });
  </script>
</head>
<body>
  <img src="img1.jpg" alt="" />
  <img src="img2.jpg" alt="" />
  <img src="img3.jpg" alt="" />
</body>
</html>
```

5. The page, loaded in a browser, will present the three images in their original size. However, clicking on one of them—to which the `gallery()` plugin is applied—doesn't do anything. We still need to write the code!

6. At this stage, the most important thing to take care of and understand is how the image is supposed to get bigger all at once.

   Actually, what we'll show is not the image itself, but rather a copy of the original picture, obtained by either reading the `src` attribute of the image and inserting another image with different properties (such as, greater height or another position) or using a jQuery API function called `clone()`.

The documentation for `clone()` is available on the jQuery website (`http://api.jquery.com/clone/`) and can be summarized by saying that it creates a copy of the set of matched elements.

7. Once we have cloned the image we have clicked on, we also have to insert the copy somewhere in the document, or it won't show up in any way!

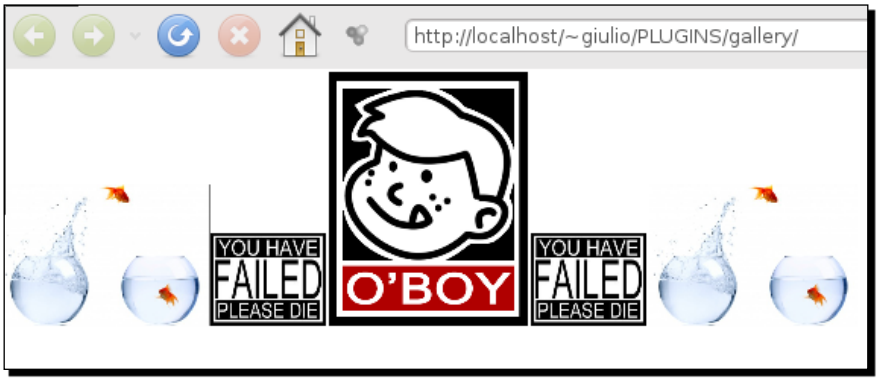
We have a list of many different methods we can choose from that all insert elements somewhere else in the page (such as `append()` and `appendTo()`, `before()`, `after()`, `insertBefore()` and `insertAfter()`, `prepend()` and `prependTo()`) and you'd better read the corresponding entry to understand what each does differently from the others.

What we'll make use of for our plugin to work is the `prependTo()` method, to add the cloned image right after the body tag.

8. With chaining, the code should now look like this:

```
(function($) {
  jQuery.fn.gallery = function() {
    return this.each(function() {
      var e = $(this);

      e.click(function() {
        e.clone().prependTo("body");
      });
    });
  };
})(jQuery)
```

9. What we get at this point is something just so deceitfully close to the end result. This will make us rather happy with what we've done with just a few lines of code.



Have a go hero – could it have been done differently?

Now that we got something tangible to look at, is `prependTo()` the only method that makes it possible to obtain something similar to what we accomplished just moments ago?

Try out the other above-mentioned functions and check—first by their description and the examples provided online—whether they're suitable or not for this situation.

Time for action – one step more

Despite the good results obtained so far, we're still facing some difficulties. One of them is, without doubt, the fact the images that are supposed to look like thumbnails are still of original size!

1. jQuery provides a set of functions—`height()` and `width()`—that directly operate on the selected element's CSS height and width properties respectively.

   At this point, it's a matter of seconds to add the following simple code snippet right before the click event in our code:

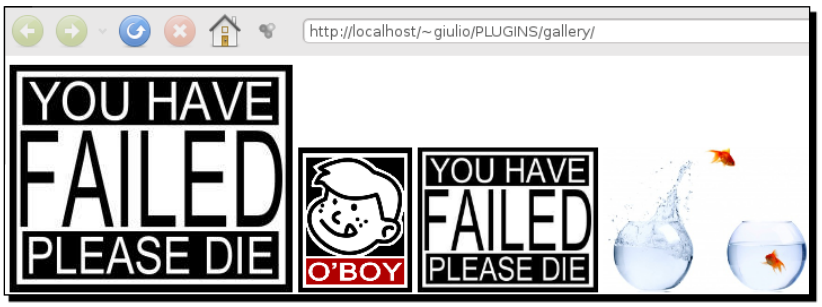
```
e.height(128); // make 128 the height in pixels of the image
               // that will display on the page and on which
               // we have to click to show the bigger version
```

2. Another problem arises though. Now, when we click on a picture (correctly sized, beware!), even the supposedly big alternative looks small. We cloned it, after all!

3. The solution would be to store the dimensions of the image before we change the size to 128px and use that saved value for the cloned version:

```
(function($) {
  jQuery.fn.gallery = function() {
    return this.each(function() {
      var e      = $(this);
      var h      = e.height();

      e.height(128);

      e.click(function() {
        e.clone()
         .height(h)
         .prependTo("body");
      });
    });
  };
})(jQuery)
```

4. Also, our aim was to see one big image at a time. As yet we're not able to make the picture disappear either by clicking on it or by clicking on another image. The images would simply lay one over each other, with the latest one clicked on top.



5. Manipulating some CSS would help tremendously.

Firstly, we have to assign a brand new ID to the element on which the user has clicked on, so we now know what needs to be made to disappear when another image in the set is clicked.

Also, we might want to use absolute positioning, so that the image is not subject to any previously set margins and/or padding:

```
(function($) {
  jQuery.fn.gallery = function() {
    return this.each(function() {
      var e      = $(this);
      var h      = e.height();

      e.height(128);

      e.click(function() {
        // removes the currently displayed preview (if any)
        $("#myGalleryId").remove();

        e.clone()
         .attr("id", "myGalleryId")
         .height(h)
         .css("position", "absolute")
         .prependTo("body")
         .click(function() {
          // whenever we click on the big preview,
          // it will disappear.
          // NOTE $(this) always refers to the currently
          // selected element, the preview in this case
           $(this).remove()
         });
      });
    });
  };
})(jQuery)
```

6. There's one last problem now: centering the image, which we'll solve by writing another plugin and putting the two of them together.

Have a go hero – implementing options

Yes, it's this moment again: options are important, and so should we treat them.

Add a couple of options using the structure we have previously learned and used. Examples could be making the user able to specify the size of the small thumbnail and the ID of the preview, in case `myGalleryId` has already been used in some other application.

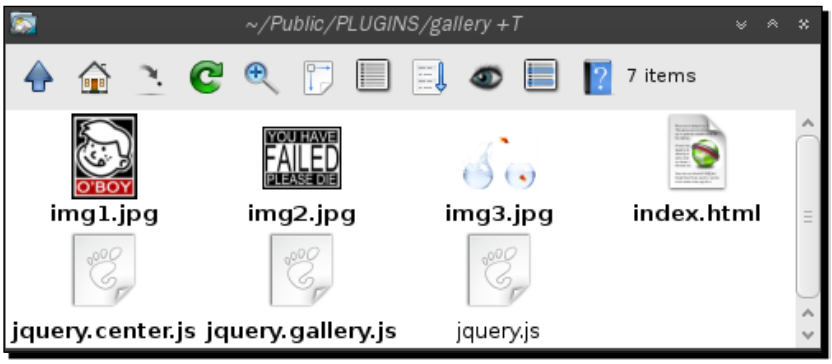
Centering things

The problem with centering things is real. In this case, it resides in the fact that we don't know the size of the image before we actually happen to deal with it and are about to center it according to the screen (or window) dimensions.

Time for action – turning theory into code

A possible, and simple, way of making things work out nicely would be, logically, to set the top margin of the picture (which is positioned using absolute values—beware, or this whole thing wouldn't work!) to half the height of the window minus half the height of the image itself. And the same goes for the space that there must be at both sides, which corresponds to half the width of the window minus half the width of the picture.

1. As we're about to create another plugin, we need another JavaScript file (`jquery.center.js` will do) to be placed inside the gallery directory and included from the index file as well. Each plugin then carries out a single function and things are decoupled.



2. The code, really simple, is very similar to its description, as we actually do nothing more than modify some CSS code so that we end up positioning the element exactly where we need it: at the center of the window.

```
(function($) {
  jQuery.fn.center = function() {
    return this.each (function() {
      var e = $(this);

      e.css({
        position: 'absolute',
        top:       ($(window).height() - $(this).height()) / 2 +
                     $(window).scrollTop() + "px",
        left:      ($(window).width() - $(this).width()) / 2 +
                     $(window).scrollLeft() + "px"
      });
    });
  };
})(jQuery);
```

3. Note we have selected and used the document (`$(window)`) that represents the entire page currently visible in the browser window.

4. Don't forget to include the new script to the HTML file, before the gallery plugin, since we need it for our implementation, but after the jQuery library.

   The head tag will now look like the following:

```
<head>
  <script src="jquery.js"></script>
  <script src="jquery.center.js"></script>
  <script src="jquery.gallery.js"></script>
  <script>
    $(document).ready(function() {
      $("img").gallery();
    });
  </script>
</head>
```

Have a go hero – vertical and horizontal centering

We've now understood how centering works.

It would be both nice and handy, though, to be able to center the element either vertically or horizontally.

This can be accomplished by adding a couple of Boolean options to the plugin (default to true) so users can choose, at their will, to have the picture centered horizontally, vertically, or at the very center of the window.

Putting it all together

Yes, we've finally got to the point where we're supposed to join the two scripts to get one final, definitive plugin that we can have fun with!

And, surprisingly enough, this step is as simple as saying it.

Time for action – the final step

What we're left to do is just apply the `center()` method we've just developed to our existing code that works in all of its parts, but cannot center the element on its own.

1. Back to our `jquery.gallery.js` file; the only thing we need to do to have the picture centered is edit the code so that a call to the `center()` method is implemented.

```
(function($) {
  jQuery.fn.gallery = function() {
    return this.each (function() {
      var e      = $(this);
      var h      = e.height();

      e.height(128);

      e.click(function() {
        // removes the currently displayed preview (if any)
        $("#myGalleryId").remove();

        e.clone()
         .attr("id", "myGalleryId")
         .height(h)
         .prependTo("body")
         .center()
```

```
            .click(function() {
              $(this).remove()
            });
        });
      });
    };
})(jQuery)
```

2. Note that, since absolute positioning is taken care of in the center plugin, there is no need to state it again in our gallery plugin.

Have a go hero – looks is everything

Even though we have a neatly working plugin, we'll never be out of things to do.

To obtain something worth using, a nice looking user interface is a fundamental requirement. Nobody, ourselves included, will ever use such a graphical failure as our plugin, offering nothing more than modestly well-written code that makes the whole actually work.

After all, what would you prefer to work with: something that just works but looks ugly or something that just works and is quite nice to look at?

Playing around with some CSS can definitely make things look a little better. Shadows (basically, a semi-transparent background image, slightly shifted relative to the picture preview) can be added, so could be a small close button placed on the top right-hand corner of the image.

After having added some CSS code for the pop-up image, you should include it as a complementary CSS file (such as `jquery.gallery.css`), as this is common occurrence for plugins.

Pop quiz

1. In our implementation, we have decided to make use of the `prependTo()` method instead of a variety of others.

   Apart from the effective final results, what differences are actually present between `prependTo()` and the very similar (in spelling, too) `prepend()`?

   A. There is no difference: `prependTo()` is an alias for the `prepend()` method.

   B. They perform the same task; with `prepend()` the selector to which the method is applied is the container into which the content is inserted. In contrast, the `prependTo()` method takes as argument the container.

   C. They perform the same task; with `prependTo()` the selector to which the method is applied is the container into which the content is inserted. In contrast, the `prepend()` method takes as argument the container.

   D. Even though their names are very similar, the two methods perform tasks completely different and doubts will hardly arise regarding their behavior.

2. Regarding the usage of `$(this)` inside the second click event in our plugin (the one that lets us make the picture disappear, so to speak), to what extent can this behavior be considered as following the usual jQuery rules?

   Isn't the expression `$(this)` always referred to the top most element which is first selected?

   A. `$(this)` always refers to the current element being processed.

   B. Depending on where it is used, `$(this)` can refer to different elements.

   C. This particular behavior is in fact strange, but justified by the special plugin structure that often makes the code act differently from what one would expect.

   D. Actually, this isn't even supposed to work. I wonder how this is possible?

3. Couldn't the centering and gallery plugins be both combined into one unique file to save space?

   A. Several plugins can be grouped together to fit into one file only to the detriment of code tidiness and organization.

   B. Several plugins can be written and stored into the same file, with no particular consequences. It's just a matter of personal liking.

C. "One plugin, one file" is one of the so-called "Golden rules of jQuery".

D. The more code you can fit into one plugin, even with some strange function calls and arrangements, the better.

Summary

Reaching the end of the chapter, a good number of new notions should have been presented to and learned by you.

Most importantly, the step-by-step structure should have helped somewhat in directing the flow of thoughts correctly and collecting the best ideas to obtain a quality final product.

The key points, fundamental for going ahead with no problems of any kind, are, obviously, related to the nature of this type of plugins and their practical realization. We're surely already tired of hearing about it but, no matter what, the importance of flexibility, extensibility, and modularity is always underestimated, and for some reason is still unknown to mankind.

This thought applies to media plugins more than others, as the images, audio files, and videos are more likely to undergo some sort of modification in terms of accessibility or even realization of basic plugin functions.

By no means should this discussion be considered exhaustive, as we've focused on a particular type of image-related plugin only. There are hundreds of different plugins out there. The approach that a developer is required to assume in order to successfully get his or her job done may be more or less similar to the one we had to take into consideration during the realization of the above-mentioned plugin.

For some further information about the type of image-related plugins one can bump into in the "real" world. I suggest you check out the following links, which collect a bunch of jQuery plugins ranging from gallery-style ones to sliders, layering, image modification, and such:

- 14 jQuery plugins for working with images:

  `http://sixrevisions.com/resources/14-jquery-plugins-for-working-with-images/`

- Top jQuery photo slideshow / gallery plugins:

  `http://blueprintds.com/2009/01/20/top-14-jquery-photo-slideshow-gallery-plugins/`

- 15 amazing jQuery image gallery / slideshow plugins and tutorials:

  `http://speckyboy.com/2009/06/03/15-amazing-jquery-image-galleryslideshow-plugins-and-tutorials/`

- Top 10 jQuery image plugins:

  `http://www.reynoldsftw.com/2009/02/jquery-top-10-image-presentation-plugins/`

Forgetting about images for a while, we're about to explore and learn about the surprising world of sounds. The next chapter will discuss the undoubtedly attractive way of handling sounds and audio files with jQuery, explaining how to realize a simple plugin that will astonish everybody because of its uniqueness.

5
Media Plugins: Audio Plugins

Media plugins play a huge and a much more important role in today's highly digitalized world. How many times can Internet addicts open up their web browser, read some pages, and not stumble on some interesting pictures that promptly catch their eyes?

And what about videos? Short, sometimes funny, videos are all over the network. Everybody happens to, sooner or later, click on the play button and check what one of these intriguing short movies is all about.

The same happens to audio files. The use of songs, and sounds in general, on web pages is spreading very quickly. Many websites offering streaming capabilities are seeing the light, resulting in more and more users being given the opportunity to listen to some music online.

Also, sounds can be included on web pages to provide music samples, in a totally different fashion than the streaming-oriented websites have made us used to.

Thinking about a page with a tiny music player is enough to get the idea of what we're going to end up with, right after we have solved some common problems plugin developers usually face when dealing with the topic of media plugins, following a somewhat individual approach.

Specifically speaking, this chapter will discuss the following:

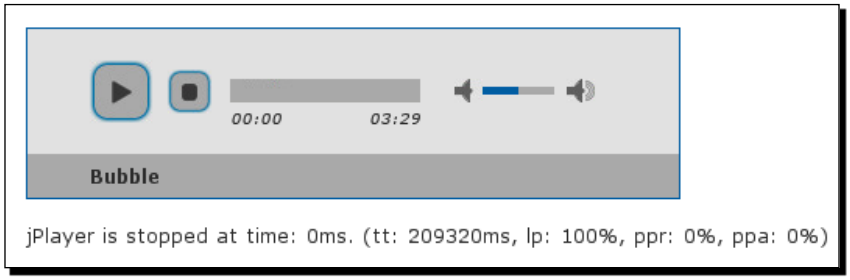
- Plugin overview
- Handling audio files
- The player
- Putting the plugin together
- Styling and multiple players

Plugin overview

The very first thing that we might want to do is actually get a good understanding of what the plugin we look forward to developing will be like at the end of the programming process and how we intend to proceed with its creation.

Analyzing some of the audio-related plugin examples that are covered in the following sections, and spending some time toying with the samples found online, will surely help make the entire process clearer in our mind, which may be a little confused right now.

Our goal is to play audio files, and nothing can be more useful than having a look at the marvelous jPlayer plugin (`http://www.happyworm.com/jquery/jplayer/`). It provides an easy-to-use wrapper to control audio files in every way possible on our web page. It provides keen attention to details such as CSS styling and HTML5 support.



Indeed, we need our player to cater for a few basic requirements. Namely, we want it to be able to:

- Play MP3 files:

  We'll be happy with playing MP3 files only for the moment. Though, we'll see making it able to handle different types of file is not much work after all.

- Pause the file reproduction:

  Definitely, this is a key feature we don't want to miss.

- Stop sound reproduction:

  Another basic feature we must include.

- Control volume (increase, decrease, and mute):

  Music can get annoying or users can for any reason decide to adjust the volume of the emitted sounds.

Handling audio files

Basically, there are two ways in which audio files can be played (and manipulated):

- With HTML5 tags
- Using Flash

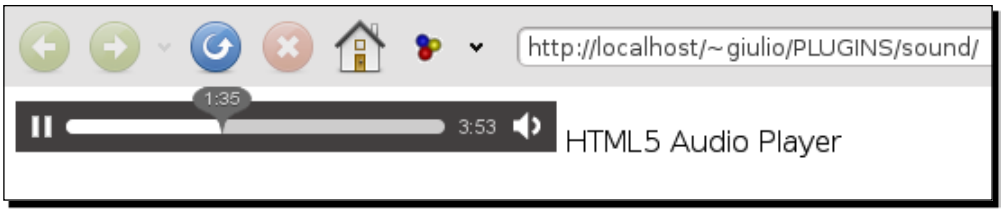
The HTML5 solution looks quite promising: simple to embed, simple to modify and manipulate. However, there's a problem. This HTML5 spec is not yet finalized, and there are endless discussions about which codec to support natively and why.

Currently, five attributes have been specified:

- `src` (URL): Content source
- `preload` (bool): Whether the audio will be loaded at page load, and will be ready to run
- `autoplay` (bool): Whether the file should play as soon as it can
- `loop` (bool): If present, whether the audio will start over again once finished
- `controls` (bool): Whether the browser should display default player controls

The HTML code will then result in something simple:

```
<!DOCTYPE html>
<html>
<head>
  <title>HTML5 audio</title>
</head>
<body>
  <audio id="player" src="sound.ogg" controls></audio>
</body>
</html>
```



The plugin is nothing more than a simple set of functions that make use of the built-in methods and attributes from the HTML5 spec:

- `play()`: To play the file
- `pause()`: To pause the file
- `buffered` (read only): Specifies the start and end time of the buffered part of the file
- `canPlayType()`: To check whether the file type is specified or not

Unfortunately, not all browsers support this spec, and many differences exist when speaking of supported file types for major browsers as well:

Browser	Ogg Vorbis	MP3	WAV
Firefox 3.5	✓		✓
Opera 10			✓
Chrome 3	✓	✓	
Safari 4		✓	✓
Internet Explorer 8			

On the other hand, Flash-based audio management is somewhat more accessible with all browsers, provided that a Flash plugin is installed or can be downloaded and installed.

The player

For our plugin to work, we need a Flash player that lets us load and play the desired files, in our case MP3 songs and sounds.

A very common choice for this type of plugin is the JW Player (`http://www.longtailvideo.com/players/jw-flv-player/`), on which a number of media-related plugins are based.

A free version of the player can be downloaded from the website, as well as some good documentation and many examples.

Time for action – creating the Flash player

Creating the basic Flash player and layout is really straightforward. All we need to do is download the Flash component and the JavaScript library to make sure that the player and our code can exchange information the right way.

1. Download the JW Player (Version 5.2 as of the time of writing) from `http://www.longtailvideo.com/players/jw-flv-player/` and copy both the `player.swf` and `swfobject.js` files into our new plugin's directory. Also, create the other files that we usually need to build a plugin and copy an MP3 file of your choice.

2. We are going to have a simple player with three links:
   - Play button
   - Pause button
   - Mute button

   Open the HTML file, and paste the following code:

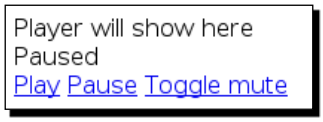
```
<!DOCTYPE html>
<html>
<head>
  <script src="swfobject.js"></"></script>
  <script src="jquery.js"></script>
  <script src="jquery.audio.js"></script>
  <script>
    $(document).ready(function() {
      // code
    });
  </script>
</head>
<body>
  <div id="player">Player will show here</div>
  <div id="status">Paused</div>
  <a id="play" href="#play">Play</a>
  <a id="pause" href="#pause">Pause</a>
  <a id="mute" href="#">Toggle mute</a>
</body>
</html>
```

3. Now let's see some information about the files we need:

   - `swfobject.js` is a script for detecting and embedding the Flash player, developed by Geoff Stearns (`http://code.google.com/p/swfobject/`).

   We need to interface the JW Player with basic JavaScript functions so that we can operate the Flash player effectively. Therefore we need the following files:

   - `jquery.js` is what we're working with. Once we have the basic JavaScript bound, we can use jQuery to easily access and modify everything in our document and make the plugin useful in some way.
   - `jquery.audio.js` is our plugin core code. Everything will be handled from here.



Putting the plugin together

Now that we have the layout done, and the JW Player files are in order and ready to be used, only the actual plugin code is left to write.

What we need to do in order to make everything work fine is:

1. Create the SWFObject to handle the audio file.
2. Add the desired (and required) variables to the above-mentioned object to specify file path, buttons visibility, and so on.
3. Place the SWFObject in the right HTML container.
4. Create a JavaScript object to manage the SWFObject through the `swfobject.js` JavaScript wrapper.
5. Add click events so that every time we click on a link, the respective action will be reflected on the player.

Time for action – creating the plugin

Now that we have the layout done and the JW Player files are in order and ready to be used, only the actual plugin code is left to write.

1. In the `jquery.audio.js` file, the following code is needed to obtain the normal plugin structure:

```
(function($) {

  $.fn.player = function(options) {
    var defaults = {
      swf:     "player.swf",
      pId:     "myPlayer",
      mp3:     "sound.mp3",
      mute:    false,
      pPlay:   "play",
      pPause:  "pause",
      pMute:   "mute",
      pStatus: "status",
      pStatusTextPlaying: "***Playing***",
      pStatusTextPaused:  "---Paused---"
    };

    var o = jQuery.extend(defaults, options);

    return this.each(function() {
      // do something
    });
  };
})(jQuery)
```

2. We have some default options already set, which will be what we'll need later, when dealing with elements and SWFObjects.

   - `swf`: This is the player path, which can be changed if you prefer another SWF Player instead of JW Player.
   - `pId`: This is the SWFObject ID. We'll use this to refer to the player object in our code.
   - `mp3`: This is the path to the MP3 file. It's been set to a file for testing purposes, but if the file is missing the program should let the user know with some sort of message.
   - `mute`: Sounds can be heard by default.

3. In addition, the following code is required to create the SWF player and put it inside the `#player` DIV we created earlier:

```
var playerObject = new SWFObject(o.swf, o.pId, "0", "0", "9");
playerObject.addVariable("file", o.mp3);
playerObject.addVariable("icons","false");
playerObject.write($(this).attr('id'));
```

4. The first line makes use of the SWFObject library and is a way to construct the object using `player.swf` (`o.swf`).

The so-created object is then assigned the `o.pId` as its ID (by default, it reads `myPlayer` but this value can be changed at runtime), whereas the following three arguments, which are integers, represent, respectively, player width, height, and the Flash version the content is published for.

As a side note, many more options are available, but not required, and can be seen in the official documentation: `http://code.google.com/p/swfobject/wiki/documentation`.

We have set both width and height to zero so the player doesn't actually show (but it's there, right inside the `#player` DIV!) and we can use JavaScript controls instead.

5. Next, we are going to bind events to something that has to do with the player's own behavior. Paste the following code right after the instruction we have already added to the `each()` loop.

```
// unfortunately, this is the only way to select the player
// object,
// since we need a JavaScript object to make the SWFObject work
var player = $('#' + o.pId)[0];


// we'll need these later
var mute = o.mute;
var playing = false;
var status = $("#"+o.pStatus);
status.addClass("status-container");

$("#"+o.pPlay).click(function() {
  player.sendEvent("PLAY", "true");
  status.text(o.pStatusTextPlaying);
  playing = true;

  // return false to prevent default processing of the link
  return false;
```

```
}).addClass("play-button");
  $("#"+o.pPause).click(function() {
  player.sendEvent("PLAY", "false");
  status.text(o.pStatusTextPaused);
  playing = false;

  // return false to prevent default processing of the link
  return false;
}).addClass("pause-button");
  $("#"+o.pMute).click(function() {
  if(mute) {
          player.sendEvent("mute", "false");
          mute = false;
        }
        else {
          player.sendEvent("mute", "true");
          mute = true;
        }

  // return false to prevent default processing of the link
  return false;
}).addClass("mute-button");
```

6. With these last modifications, the plugin is ready to work and can be implemented in our web page with the following code:

```
<script>
  $(document).ready(function() {
    $("#player").player({ mp3: "some-music.mp3" });
  });
</script>
```

Unfortunately, the JW Player API still has some problems in terms of stability. For this reason, a little tweaking may be needed in order to get the plugin to work.

Some support can be obtained from the official website (`http://longtailvideo.com`) and developer pages, as well as some third-party forums dealing with this kind of issue.

Have a look at some of the links in Appendix A.

Have a go hero – add controls

We've only seen a couple of controls for the JW Player.

Actually there are many more, allowing for a more detailed and precise control of the media file and providing the basics for some possible extension of functionalities such as progress bars, volume sliders, or similar.

With the JW Player documentation at hand (`http://developer.longtailvideo.com/trac/wiki/Player5Api#Plugins2`) and after having read the Slider plugin documentation pages (`http://docs.jquery.com/UI/Slider`) try to create a volume slider to adjust volume during the reproduction. Keep in mind that a volume (`<int>`) method from the JW Player makes the whole process easier.

Styling and multiple players

Alright, our player works. But it looks awful!

Fortunately, we can add some CSS styling to our plugin and make it a little nicer looking. After all, it's just HTML code and some JavaScript!

The point here is to make the CSS code as little intrusive as possible but, at the same time, very flexible. We might have multiple players on the very same page at once and we want them to look exactly the same.

Time for action – adding support for multiple players

The possibility to have multiple players on the same page is really nice, but what do we do with all those controls and similar stuff we have selected using specific IDs? And, most importantly, how can we count the actual number of players on the page?

1. First of all, we must identify each player with a unique ID.

   Also, we have to keep track of the number of players we have set up.

   Our plugin code will then need a variable (say, `count`) starting at zero and put right after the closure—thus the second line, after the one reading `(function($) {`, so it is available to all of our code.

   This way, whenever we include the plugin file into some HTML document and load the page, the variable is initialized at zero, meaning the first player is ready to be set up.

At the very end of the each loop, the `count` variable is to be increased by one. The next player can now be set up.

The value of `count` can then be appended to any ID referring to the player, so we end up with a unique reference for that object.

2. Another problem is the various control links. If we leave the code as it is, we only have three links actually working for all players, and it surely isn't the best way to make things work smoothly.

   Instead, we can let the user specify the ID of the three controls, while we modify our plugin code to act accordingly

3. Last but not least, if we want all the players to look the same, some type of class must be assigned to control links so that they can look similar when CSS styling is applied.

4. To sum it all up, after the above-mentioned modifications, our code will look like the following:

```
(function($) {
  var count = 0;

  $.fn.player = function(options) {
    var defaults = {
      swf:     "player.swf",
      pId:     "myPlayer",
      mp3:     "sound.mp3",
      mute:    false,
      pPlay:   "play",
      pPause:  "pause",
      pMute:   "mute",
      pStatus: "status",
      pStatusTextPlaying: "***Playing***",
      pStatusTextPaused:  "---Paused---"
    };

    var o = jQuery.extend(defaults, options);

    return this.each(function() {
      var playerObject = new SWFObject(o.swf, o.pId + count, "0",
         "0", "9");
      playerObject.addVariable("file", o.mp3);
      playerObject.addVariable("icons", "false");
      playerObject.write($(this).attr('id'));
      $(this).addClass("player-container");
```

```
        var player = $('#' + o.pId + count)[0]

        var mute = o.mute;
        var playing = false;
        var status = $("#"+o.pStatus);
        status.addClass("status-container");

        $("#"+o.pPlay).click(function() {
          player.sendEvent("PLAY", "true");
          status.text(o.pStatusTextPlaying);

          playing = true;
        }).addClass("play-button");

        $("#"+o.pPause).click(function() {
          player.sendEvent("PLAY", "false");
          status.text(o.pStatusTextPaused);

          playing = false;
        }).addClass("pause-button");

        $("#"+o.pMute).click(function() {
        if (mute) {
            player.sendEvent("mute", "false");
            mute = 0;
          }
          else {
            player.sendEvent("mute", "true");
            mute = 1;
          }
        }).addClass("mute-button");

        count ++;
      });
    };
})(jQuery)
```

5. We can now create any number of players, paying attention to the ID of the elements specified as controls. Creating two players (`player0` and `player1`) is enough to check the correct functioning of our plugin. Also pay attention to the creation of the necessary elements that will work as player controls.

```
<script>
  $(document).ready(function() {
    // All default: plays sound.mp3 and uses #play, #pause, #mute
    $("#player0").player();
```

```
    // Default values changed to our liking: of course other
    // elements (with ids #play1, #pause1 and #mute1)
    // are needed for the magic to happen
    $("#player1").player({ mp3:  "some-music.mp3",
                           pStatus:  "status1",
                           pPlay:   "play1",
                           pPause:  "pause1",
                           pMute:   "mute1" });

  });
</script>
```

Have a go hero – manage multiple sounds

Having the possibility of creating multiple players objects on the same page also implies more than one event can be sent to other players—leading to multiple sounds played at once.

Find a way (create a function or whatever) to pause all players prior to one of them starting to play its sound.

Time for action – adding some style

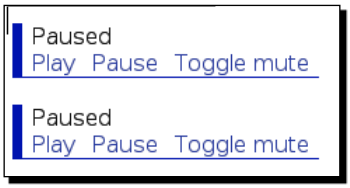
We've finally come to the point of making our player(s) better looking. Some simple CSS instructions will help our plugin to create much better looking Flash players.

1. The following CSS code, to be put inside a separate CSS file (`jquery.audio.css`), will make the players look different:

```
.player-container {
  display: none;
}

.status-container {
  border-left: 7px solid #06f;
  padding-left: 7px;
  margin-top: 20px;
  color: #666;
}

.play-button {
  border-left: 7px solid #06f;
  padding-left: 7px;
  margin-left: 0px !important;
}
```

```
a {
  text-decoration: none;
  border-bottom: 1px solid #06f;
  margin-left: -7px;
  padding-left: 7px;
  padding-right: 7px;
  color: #89f;
}
```

The players will now look as follows:



2. Playing a bit with CSS can then lead to obtaining a completely different appearance. Also, keep in mind that a number of surprising effects can be obtained using jQuery's own methods and plugins!

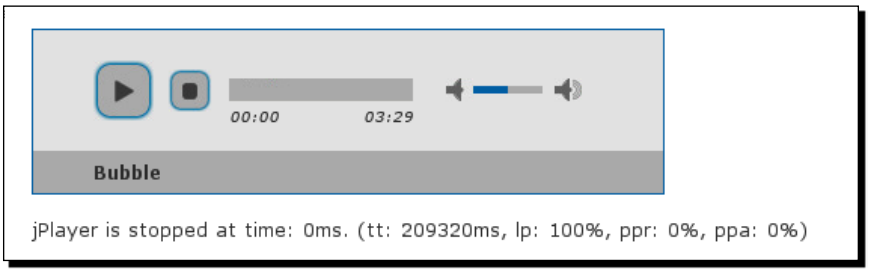
Have a go hero – improve controls

This is a text-based MP3 player, which you can style however you want.

It would be great if some buttons were displayed though, thereby adding that bit of spice and modern look that every player should have.

Have some fun with images, CSS, and jQuery effects in order to end up with a stylish media player, looking like the one that we saw at the start of the chapter: jPlayer.

Your end product should provide sliders for both time playing and volume controls, as well as buttons to play, pause, stop, and mute.



Pop quiz

1. Apart from the use of Flash to make the audio plugin work, we've seen how the HTML5 `<audio>` tag makes it really easy for everybody to include their media files into a web page.

   For what reason have we decided not to use this handy method to embed audio files?

   A. Browsers support different codecs, and debates are still open regarding which one to use for the next HTML release.

   B. The uncertainty of the spec and the fact that nothing can be considered definitive makes every implementation highly subject to possible future modifications.

   C. Both of the above.

   D. Neither of the above.

2. On the other hand, what inconveniences does the use of Flash imply?

   A. It's not always reliable due to its lack of support in all major browsers and its proprietary license.

   B. It's very difficult to work with, since it does not provide any simple way of interaction.

   C. Many people don't like it, relegating the use of Flash to a niche of "elite" programmers.

   D. It needs the Flash plugin to be run and, as such, some people might not have it installed and thus are not able to see Flash content.

3. What does the SWFObject library actually do? Why do we need it to make our plugin work?

   A. It offers optimized embed methods and a JavaScript API to work with Flash related information.

   B. It's needed by JW Player, the Flash player of our choice, to make everything work smoothly and with no problems whatsoever.

   C. It's a jQuery plugin to work with Flash objects.

   D. None of the above.

4. How can multiple players be handled if we only have one method for them all?
    A. They can't. And that's why this plugin won't work without some additional code.
    B. We've provided the user with the choice of the IDs to be used, so multiple players can be inserted into the same page and controlled by different elements.
    C. For every new SWFObject we create, a new ID is also created and automatically assigned to the right element in the HTML code.
    D. Thanks to CSS rules we can style player elements so they look the same even though they have different IDs.
5. Can we use another SWF player instead of JW Player, which we currently make use of?
    A. Yes, and the code remains the same.
    B. Yes, but we have to change the code.
    C. No, JW Player is the only one that can play MP3 files using the Flash plugin.
    D. Yes, but a new Flash executable has to be compiled for this to happen.

Summary

As we have noticed, audio files are fairly tough to deal with.

Not only do we have little to no options and possibilities to explore, but we also have to face the fact that there actually is no easy way to get around the problem of handling media files such as sounds and videos.

HTML5 looks very promising, as it provides easier handling, especially for media objects. But unfortunately it's not yet a standard and very few developers choose to implement this technology over the most widespread ones.

The use of Flash is, in fact, the common choice for developers who want to get this area clearer in their minds. However, as we have stated earlier, this technology is far from perfect and available for everybody, since many people consciously turn the Flash plugin off or don't install it because of either license-related issues (closed source), or lack of support or interest on their part.

Flash is also quite complex to deal with because of its nature. Once you have the executable, options are very limited, not to mention the impossibility for non-Flash developers, to change anything or get their own player (in this case) or (more generally speaking) application done by themselves.

Some JavaScript wrappers do exist, though, to help dealing with certain Flash players. For example, the JW Player wrapper is compatible with SWFObject, but not with many of the others out there, which, obviously, cannot provide an interface for every existing media player.

The problem with SWFObject is it's written in plain JavaScript and, as such, cannot smoothly cooperate with jQuery, especially due to the stability issues that affect the JW Player API.

A jQuery plugin exists that is based off the latest SWFObject. It provides some of its functionalities and lets you embed Flash content into any web page with jQuery-friendly syntax and methods.

A closer look at its home page (`http://jquery.thewikies.com/swfobject/`) might be useful out of curiosity, even though no real documentation can be found on those pages.

Regarding this topic, it is also difficult to provide a list of plugins to toy with as a reference. There are very few and they all rely on some Flash Player of their choice and another JavaScript library to control the player behavior.

With videos, in the next chapter, things will go better for sure, as some more possibilities are available and ready to be checked out.

The realization of a videos plugin is quite similar to what we've done so far, as media files in general (thus, audio and video files) are often treated the same, meaning media players are usually able to play both videos and sounds, depending on the file type they recognize upon initialization.

6
Media Plugins: Video Plugins

We've said it over and over again, however, as technology evolves, media content plays an important role in our everyday life.

Having dealt with images and audio in the previous chapters, we are now going to explore the last of the multimedia-type plugins that we intended to develop: video plugins.

For some time now, video files have been getting really popular. Many websites offering some kind of support and/or space to host videos and then send the links to friends and relatives have seen great success (think YouTube, Vimeo, and so on).

These type of websites offer the user a super-simple, quick way to make it possible for other people (even from far, far away places and unknown to the video owner) to watch a short film (for free), send in comments (thus, interacting with other users), and eventually include the video itself onto their own web page, to help spreading the content over the Web.

The following are the topics we're going to deal with in this chapter:

- Plugin overview
- Handling video files
- Embedding YouTube videos
- Adding preview thumbnails and the pop-up feel

Many plugins and updates have been released to allow users of the most popular frameworks or libraries to easily display videos from different sources on certain web pages. Users can also control, manipulate, and change the player's appearance, video reproduction, and other fundamental parameters.

On the other hand, the new HTML5 specification has introduced support for video by designing an element (`<video>`), for playing and manipulating video files in various formats.

However, HTML5 is far from completion and not many browsers support it yet—we'll have to wait for some time before it's a widely recognized and supported standard.

Plugin overview

Our goal for this chapter is to develop a jQuery plugin capable of potentially upon some event, embedding, styling (that is, animating, coloring, or whatever), and eventually playing a YouTube video whose link is specified by the user.

In other words: on being given a text link, our plugin will do the dirty work for us, making the entire thing better looking than necessary. However, nowadays good looks are a good investment!

Looking on the Internet for some plugins similar to what we're going to realize could be useful to better understand the actual goals and aims for this chapter.

With particular interest, a couple of plugins have some of the things we're looking for. For example, just by creating a link element (`<a>`) pointing to the YouTube video, the *jYouTubeVideo* plugin (written by Muhammad Hamed) makes the following possible:

Also, more keenly on the graphic part, the *Floaty* plugin, by Tohid Golkar, is responsible for the video popping up nicely, once the user clicks on the automatically-generated video thumbnail.

To finally write what are we running after, here is a short list of our main goals so far.

We want our plugin to be able to:

- Embed YouTube video files
- Display a small thumbnail
- Pop out a bigger window (of which we'll choose the dimensions) and play the video

Handling video files

In order to make the plugin work as we expect, and create what we need, we actually have little to do when speaking of "handling video files". In the true acceptance of the term, YouTube's own player is already ready and all we need to do is embed it into our web page using some HTML code.

Despite the almost ready-to-use package we find ourselves dealing with, there's actually something we should focus our attention on for a little while before we actually start working on our plugin: that is, YouTube embedded player parameters.

Whenever we link to a YouTube video, the URL can be subdivided into many different parts. As an example, just take the following address, linking to a random YouTube video, which will start playing automatically from the one minute mark and will be repeated endlessly:

`http://www.youtube.com/watch?v=1IVAR5MCGNY&autoplay=1&start=60&loop=1`

In our example:

- `v=1IVAR5MCGNY`

  Is the ID of the video we want to watch.

- `autoplay=1`

  Tells the player to automatically start the video.

- `start=60`

  Is the number of seconds from the start of the video at which the player begins playing the video.

- `loop=1`

  Forces the player to play the video again and again.

And that's it; with some simple additions to the URL we have obtained a slightly modified player that's closer to our needs. Of course, if we actually needed something more complex and advanced, we would work our way to some kind of framework, as we did with audio plugins, resulting in a rather functional plugin that cost us more effort than necessary (in this case).

A full list of YouTube embedded player parameters is available at this address: `http://code.google.com/apis/youtube/player_parameters.html`, and some examples are provided as well.

Embedding YouTube videos

Provided we already know how to embed a Flash object into a web page, this part won't be (or at least shouldn't be) difficult at all. The trickiest (but most rewarding) bits come later!

Basically, we need two different embed code blocks as seen in the following code snippet. One (`object`) is the W3C standard whereas the other (`embed`) is deprecated but oftentimes used for Mozilla files as a safe backup in case something goes wrong.

```
<object width="425" height="355">
  <param name="movie" value="http://www.youtube.com/
     watch?v=rSgfHjLmmj8"></param>
  <param name="allowScriptAccess" value="always"></param>
```

```
    <embed src="http://www.youtube.com/watch?v=rSgfHjLmmj8"
      type="application/x-shockwave-flash"
      allowscriptaccess="always"
      width="425" height="355">
  </embed>
  </object>
```

Our aim is to obtain some code similar to the above starting from something similar to:

```
<a class="video" href="http://www.youtube.com/watch?v=rSgfHjLmmj8">You
tube video</a>
```

Time for action – creating your first video plugin

Our first attempt to create a video plugin is rather simple. We can always modify its functionalities at a later time!

1. We should already have a new directory set up, called `video`, and the files we normally make use of already present in it.

2. Our `index.html` file will just have a link in it (`http://www.youtube.com/watch?v=rSgfHjLmmj8`). All of the work is done by our plugin!

```
<!DOCTYPE html>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="jquery.video.js"></script>
  <script>
    $(document).ready(function() {
      // Plugin call
      $(".video").video();
    });
  </script>
</head>
```

```
<body>
  <a class="video" href="http://www.youtube.com/watch?v=rSgfHjLmmj
8">Youtube video</a>
</body>
</html>
```

3. Once the element on which we want to operate is identified, the first thing that our plugin has to take care of is to retrieve the `href` attribute (thus the YouTube link) and create the `<object>` and `<embed>` code blocks, which will be inserted right after the link:

```
(function($) {
  jQuery.fn.video = function(options) {
    var defaults = {
      width:   425,
      height: 355
    };

    var o = jQuery.extend(defaults, options);

    return this.each(function() {
      var e = $(this);
      var id = e.attr("href").match ("[\?&]v=([^&#]*)")[1];

      $object = $('<object></object>');
      $param   = $('<param></param>');
      $embed   = $('<embed></embed>');

      $object.attr('width', ''+o.width)
             .attr('height', ''+o.height)
             .appendTo(e);

      $param.attr('name', 'movie')
            .attr('value', 'http://www.youtube.com/v/'+id)
            .appendTo($object)
            .clone()
            .attr('name', 'allowScriptAccess')
            .attr('value', 'always')
            .insertAfter($param);

      $embed.attr('src', 'http://www.youtube.com/v/'+id)
            .attr('type', 'application/x-shockwave-flash')
            .attr('allowscriptaccess', 'always')
            .attr('width', ''+o.width)
```

```
                .attr('height', ''+o.height)
                .appendTo($object);
        });
    };
})(jQuery)
```

Regular expressions

Though unknown to many, regular expressions play a huge role in a number of applications, from checking e-mail to phone and credit card authentication.

In our example, we need a regular expression to simply identify the video ID from the URL string.

The series of almost incomprehensible symbols "`[\?&]v=([^&#]*)`", basically means "take everything after `?v=` except for `&` and the # symbols". We have now checked whether the URL has a part of itself containing the video ID.

Have a go hero – fix some minor imperfections

Unfortunately, this plugin doesn't quite look as we'd like it to.

There are still things to do and fix, but some minor modifications can be done on the spot, right now:

- The text (**Youtube video**) shouldn't be visible. Remove it after the video has been inserted.
- The video doesn't play automatically and does not loop either. Add some options that, following the YouTube player parameters, make the user more aware of what's happening.

Adding preview thumbnails and the pop-up feel

The idea of having to display some sort of video preview may have puzzled somebody out there but, luckily enough, we know our way pretty well and are aware YouTube stores thumbnails online just for this reason!

The web address, common to all videos, which lets us access images on the YouTube servers is the following:

`http://img.youtube.com/vi/<VIDEOID>/<VERSION>.jpg`

- `VIDEOID` is, of course, the ID of the video the image has been taken from.
- `VERSION` can have a value of 'default', 0, 1, 2, and 3, where:
  - 'default' retrieves the default small (122x100) thumbnail.
  - 0 is for a bigger (440x360) version of the thumbnail.
  - 1, 2, and 3 are small images taken at different times of the video.

Time for action – adding previews

Having thumbnails added to our link to give the user an idea of what the video is about has never been easier. Here is how we can retrieve the images from YouTube and display them on our page.

1. Having seen the light, we can now go on modifying our plugin to get closer and closer to our objective.

   We already have retrieved the ID of the video using a regular expression, and all we need to do is replace the inner HTML code of the link with our image.

2. Our plugin, after the preview and after some parameters have been added, should look like this:

```
(function($) {
  jQuery.fn.video = function(options) {
    var defaults = {
      width:    425,
      height:   355,
      autoplay: 1,
      loop:     0,
      thumb:    'default'
    };
```

```
    var o = jQuery.extend(defaults, options);

    return this.each(function() {
      var e = $(this);
      var id = e.attr("href").match("[\?&]v=([^&#]*)")[1];

     e.html('<img src="http://img.youtube.com/vi/'+id+'/'+
       o.thumb+'.jpg" class="thumbnail" alt="Youtube video"" />');
    });
  };
})(jQuery)
```

3. It's important to note that, by changing the default value for the thumbnail option, other pictures can actually be displayed at the user's will.

Time for action – creating a pop up

The pop up is similar to the one we used for the gallery plugin, except for the fact that we're going to make it a little better this time by paying more attention to details.

1. Remember the `jquery.center.js` plugin from Chapter 4? Well, we need it now. So find it and copy it over to the video directory. Also, include the script in the index page.

2. We first have to understand how the pop up is intended to work. Whenever it's activated, the pop-up script should turn the document color into black (well, transparent black, for that nice effect we're all used to) and display a new element right above everything else.

   We need two new HTML elements and some CSS code (to insert into `jquery.video.css` and include together with the plugin) for this to happen.

   The HTML code to add at the beginning of the `body` tag is as follows:

```
<div id="overlay"></div>
<div id="popup">
  <div id="video"></div>
  <a id="close" href="#">Close</a>
</div>
```

Note that we need the preceding code only once regardless of how many video links are present on our page. However, it can also be generated by the plugin automatically to make it really self-sufficient.

The CSS code is as follows:

```
.thumbnail {
  border: 0;
}

#overlay {
  display: none;
  background-color: #222;
  opacity: .75;
  filter: alpha(opacity=75);
  position: fixed;
  left: 0;
  top: 0;
  width: 100%;
  height: 100%;
  z-index: 1000;
}

#popup {
  display: none;
  text-align: left;
  position: absolute;
  padding: 10px;
  border-radius: 5px;
  -moz-border-radius: 5px;
  -webkit-border-radius: 5px;
  border: 2px solid #333;
  background-color: #fff;
  z-index: 2000;
}
```

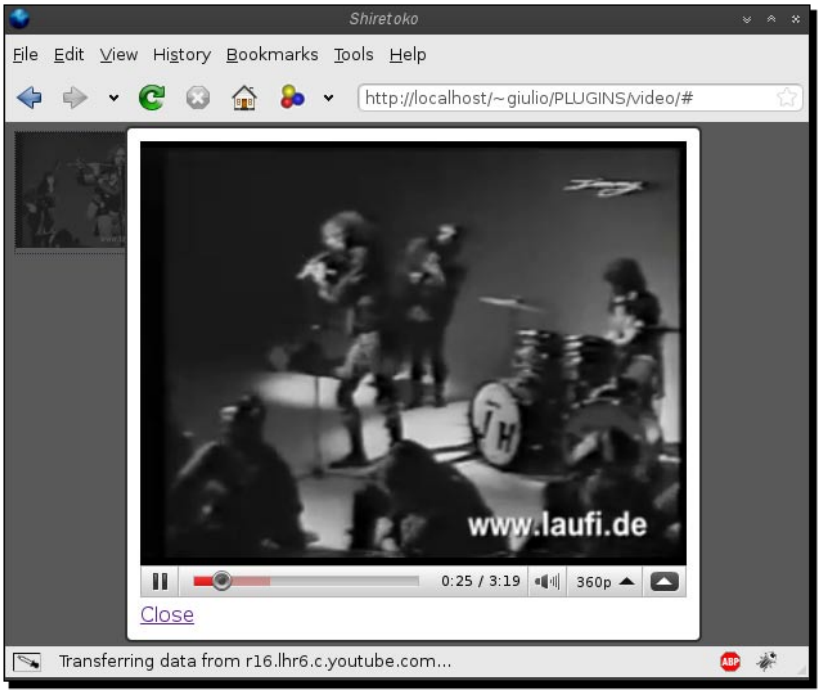
Please note that the CSS code for plugins is better stored in separate files (that is, `jquery.video.css`) and provided as part of the plugin package for easy inclusion in a page.

3. And now, our plugin will be responsible for registering events and make the whole thing work upon a mouse click, as follows:

```
(function ($) {
  jQuery.fn.video = function (options) {
    var defaults = {
      width:    425,
      height:   355,
      autoplay: 1,
      loop:     0,
      thumb:    'default'
    };

    var o = jQuery.extend(defaults, options);

    return this.each(function() {
      var e = $(this);
      var id = e.attr("href").match("[\?&]v=([^&#]*)")[1];

      e.html('<img src="http://img.youtube.com/vi/'+id+'/'+
        o.thumb+'.jpg" class="thumbnail" alt="Youtube video" />')
        .click(function(evt) {
        evt.preventDefault();

        $object = $('<object></object>');
        $param   = $('<param></param>');
        $embed   = $('<embed></embed>');

        $object.attr('width', ''+o.width)
               .attr('height', ''+o.height)
               .prependTo($("#popup #video"));

        $param.attr('name', 'movie')
              .attr('value', 'http://www.youtube.com/v/'
                     +id+'?autoplay='+o.autoplay+'&loop='+o.loop)
              .appendTo($object)
              .clone()
              .attr('name', 'allowScriptAccess')
              .attr('value', 'always')
              .insertAfter($param);

        $embed.attr('src', 'http://www.youtube.com/v/'
                     +id+'?autoplay='+o.autoplay+'&loop='+o.loop)
              .attr('type', 'application/x-shockwave-flash')
              .attr('allowscriptaccess', 'always')
```

```
                .attr('width', ''+o.width)
                .attr('height', ''+o.height)
                .appendTo($object);
            $("#popup #close").click(function() {
              $("#popup").fadeOut("slow").remove();
              $('#overlay').fadeOut("slow");
            });

            $("#popup").center().fadeIn("slow");
            $('#overlay').fadeIn("slow");
          });
        });
      };
})(jQuery)
```

4. Clicking on an image will now result in a pop up being displayed with fading effects!



What just happened?

In the previous two *Time for action* sections, we have achieved quite a large amount of work. However, let's see exactly how we managed to get to this point.

The part dealing with previews shouldn't be much of a problem. The only tricky part could have been retrieving the images from YouTube servers or, even earlier than that, parsing the URL with a regular expression to eventually obtain the video ID.

On the contrary, what might arouse some doubts and interest is the pop-up behavior.

We have used two divisions (`<div>`) for our purpose. The overlay is the one that, once the preview is clicked, occupies the whole window height and, put on top of other items using the `z-index` CSS property, paints the page transparent black. The other DIV, the pop up itself, is displayed above the overlay (thus, its `z-index` value must be greater than the value set for the `#overlay` DIV) and is the container of the video, which will be inserted into the `#video` DIV, replacing the existing HTML code.

The pop up can be styled in any way we want: it's just CSS after all!

Also, the close link does nothing more than fading out both the overlay and the pop up, and deleting the pop up, with the effect of stopping the video and making everything go back to normal.

The fade effect can also be replaced with some other, more advanced effect, for which lots of plugins are available on the Internet.

With some more tweaks, a cooler effect can be obtained. For example, when the user hovers the mouse pointer over a video thumbnail, image previews (being the three images taken by YouTube from different moments in the video) pass by to give a better idea of what the video is about.

Pop quiz

1. There are two ways to embed Flash content into a web page: using the `<embed>` tag or the `<object>` tag.

   We have used both, one inside the other. Why is using both recommended?

   A. For compatibility issues. `<embed>` works for Internet Explorer whereas `<object>` is for other browsers.

   B. To make sure at least one works.

   C. They are equivalent, except for the fact that `<embed>` is deprecated now and is mostly used to satisfy Mozilla browsers as a backup solution.

   D. They are equivalent, except for the fact that `<object>` is deprecated now and is mostly used to satisfy Mozilla browsers as a backup solution.

2. Referring to our example, how can the video code be added to the pop up using jQuery methods?
   A. Using either `after()` or any other equivalent.
   B. Using the `text()` method.
   C. Using the `html()` method.
   D. None of the above.
3. What has the `#overlay` DIV been used for in our plugin?
   A. Its goal was to obscure the visible page and prevent mouse interactions with other elements on the page.
   B. It represents the pop up itself, on which the video is displayed and played.
   C. As a simple container, it has no actual practical use.
   D. None of the above.

Summary

Apart from the successful realization of our video plugin, we have noticed how little jQuery (but JavaScript in general, for that matter) can do to handle media content (that is, particularly videos and sounds) in a simple and inexpensive way.

One of the reasons why we have little to no control over certain elements or objects at a relatively high level is the particular nature of the items themselves.

If, speaking of images, for example, we can manipulate many of their properties from HTML code (then accessible from JavaScript functions, too), the same doesn't necessarily apply for more complex media types that even require third-party plugins to work properly.

Despite the difficulties one may encounter when dealing with topics like this, we should never forget the main goal our plugin(s) have. If our intent was to pop up a video and display a preview, so be it. We won't be trapped in those additional plugins, players, and frameworks definitely not worth the effort, as the majority of video interaction we might ever need is, in fact, being able to easily and effortlessly implement a video clip from YouTube.

If, on the other hand, we aim a little higher and are willing to face the challenge of developing a more advanced jQuery plugin (with a Flash plugin and JavaScript—or jQuery—framework included too), many paths can be followed, each of which will greatly help in significantly increasing our knowledge.

However, a mid-way approach is round the corner. The new HTML5 specifications aim at creating some "guidelines" for media content so that it should look, feel, and sound the same for everyone and should be simpler to deal with.

Also, many methods for controlling media content should be already made available for JavaScript, and we had a look at them in the chapter dealing with sound plugins.

As for the next chapter dealing with forms plugins, a somewhat different approach will be used. As it's quite a heavy topic, some of the most popular and commonly-used plugin types will be analyzed (including form validation and submission) and some full-length code will be written and taken care of.

7
Form Plugins

Every website we visit on a daily basis has some kind of form for sure. It can be a form to submit comments, contact the author or owner, or just "sign" an online guestbook.

Forms make interactions between people possible. In plain words, once you've written your comment, idea, or suggestion, all you have to do is click on the "Send" button and something will happen (for example, the owner will be notified, a comment will be posted, or an e-mail will be sent, and so on).

However, as everyone can access online forms (yes, spammers and psychos too!), we might want to think of some way to protect our website from unwanted comments and our e-mail from weird messages. Some common examples are: by using Captcha, form (or e-mail, or both) validation techniques, and so on.

Also, for those who genuinely wish to contact us or state their point of view about an article we've written, we might want to provide some sort of improvements (for example, spell check, help labels, and so on). They may admire our beautiful form, created using jQuery enhancements, and may even ask us about it.

Specifically, the topics we're going to deal with in the following pages concern:

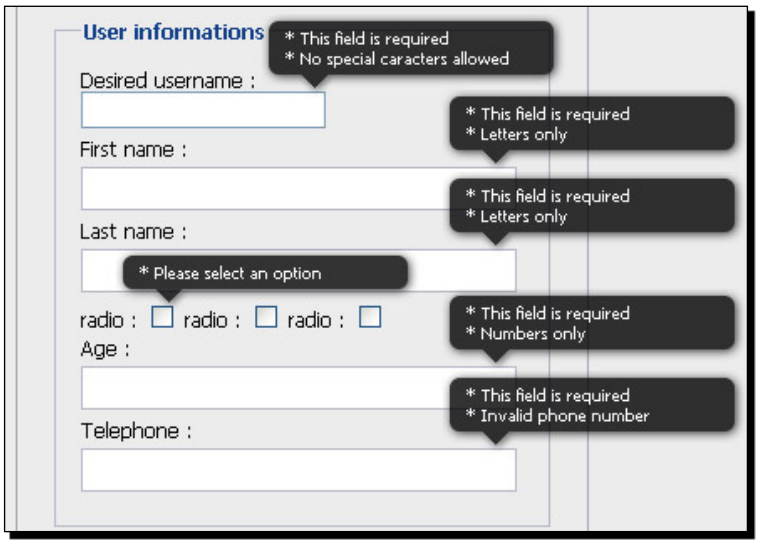
- Form plugins in general
- Validating forms
- Auto-growing textareas
- Other types of form-related plugins

As you can see, there actually are many different ways of dealing with forms, from different points of view such as security, accessibility, and general look and feel.

In this chapter we are going to see how some of the most popular form-related plugins can be realized and implemented on our web page or application with particular focus on the versatility and scalability aspects.

Form plugins in general

As strange as it may sound, form plugins are those jQuery plugins that, in one way or in another, are related to forms, be it for spicing up forms a little (changing colors, adding pictures, or whatever) or for validation purposed styling and/or usability improvements.



After all, forms are a practical and functional way of submitting any kind of information in different ways, but often lack security and give spammers and really annoying people the n-th chance to eventually spam and annoy more.

As for many other jQuery plugin categories, a lot of people have given their contribution in terms of code, functions, and plugins. Many interesting solutions have been proposed to solve some of the most common and widespread problems that arise when developers or users need to use forms.

For one, there's the always present problem of validation. Be it due to misunderstanding, distraction, or even on purpose, people are likely to submit forms containing missing or wrong information. Often, these errors are even incompatible with the type of information that was supposed to be entered, thus causing, once again, problems on the server side or to other people.

In conclusion, the main goal that all form-related plugins share is improving, and sometimes also simplifying user experience and interaction with this kind of element. In the end, no matter what the plugin really accomplishes, as long as it might be useful to somebody, it should provide an original and effective solution to this trouble.

Validating forms

Let's face it: the so-called "form validation" thing is one of those features we somehow both hate and love at the same time!

After having clicked on the **Submit** button, if a validation plugin kindly informs us that our application form contains errors or misspellings, we truly feel a blind rage running up our spine. However, we actually fall in love with that very same plugin when we know for sure that it is validating some little kid, spammer, or a diversely funny person who is trying to fill our own website's comments page with pointless posts and uninteresting (and unwanted!) text ads.

A form validation example, as seen on bassistance's form validation plugin, is as follows:

In fact, the latter's poorly thought-out attempt at putting people off can be oftentimes avoided by placing a Captcha at the bottom of the form or, sometimes, even a simple check against the e-mail address or some other detail can help keep undesired attention off our web space.

To tell the truth, controls over e-mail addresses and other text fields are actually more focused on helping out the user to fill in the form without errors of any kind—even though sometimes this kind thought might look troublesome, especially when we are in a hurry.

A less annoying attempt at making things work smoothly can be by implementing an inline error notification. Users will eventually be satisfied, and forms will be filled with no errors. Using this type of notification—instead of loading another page to display the error or even clear all of the fields forcing the user to re-fill the same textboxes for the n-th time—errors are displayed right next to the textbox, with no need of redirecting everything to another page in a very non-jQuery-like fashion.

Our aim at this point would normally be to develop a plugin that provides a set of methods to automatically check textboxes' content once the **Submit** button is clicked. However, keep in mind that a validation plugin is already included in jQuery and writing the following line of code is enough to have the form checked once the **Submit** button is clicked on:

```
$("#form-selector".validate();
```

Time for action – creating the form check plugin

Let's create our own basic form validation plugin.

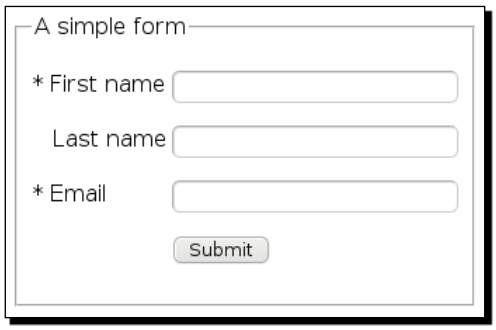
1. Create a new directory, call it `formcheck`, and copy over or create on the spot the files we will need.

2. Our HTML file only contains a form and the call to our plugin:

```
<!DOCTYPE html>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="jquery.formcheck.js"></script>
  <script>
    $(document).ready(function() {
      $(".formToCheck").formCheck();
```

```
    });
  </script>
  <style>
    form { width: 300px; }
    label { position: absolute; }
    input:not(.submit), textarea { margin-left: 100px;
       width: 200px; }
    .submit { margin-left: 100px }
  </style>
</head>
<body>
<form class="formToCheck" id="formToCheck" method="get"
   action="#">
<fieldset>
   <legend>A simple form</legend>
   <p>
     <label for="fname">* First name</label>
     <input id="fname" name="fname" class="required" />
   </p>
   <p>
     <label for="lname">   Last name</label>
     <input id="lname" name="lname" />
   </p>
   <p>
     <label for="email">* Email</label>
     <input id="email" name="email" class="required email" />
   </p>
   <p>
     <label for="comment">* Comment</label>
     <textarea id="comment" name="comment" class="required">
     </textarea>
   </p>
   <p>
     <input class="submit" type="submit" value="Submit"/>
   </p>
 </fieldset>
 </form>
</body>
</html>
```

The form is displayed as follows:



3. And now, our plugin code.

   The first things we must take care of are (A) to check whether the element we are working on is actually a form or not, and (B) to make sure we will handle everything when, and only when, the submit button has been pressed:

```
var form = $(this);

// (A)
if(!form.is("form")) return;

form.submit(function() {
  // (B) code goes in here
});
```

4. Now, we have to figure out a way to display errors, if any, and check if there are errors, meaning the purpose of the field has not been respected.

   To set things out, errors will be displayed in red right next to the field, and the textbox borders will also turn red for the occasion.

   The type of information we expect the user to write into each field will be indicated by the element's class and may be any combination of the following:

   - `required`: This points out that the field must be filled in order for the form to be valid.
   - `email`: This field contains an e-mail address and the content must, at least, look like a valid e-mail address.
   - `min#` (# being a positive integer): This is the minimum length of the contained string.
   - `numbers`: This is for fields that are designed to contain numbers only (no spaces either).

5. The first problem we face is how to select each input belonging to the form we're working on:

```
$(":input", this).each(function(index, element) {
  // code to execute for each element
  var e = $(element);
});
```

6. The required check is actually really straightforward: if a field is "required" it must not be empty, and thus have a number of characters lower than one.

   Also, if there is an error (that is no content entered) we'll add an "error" class to the element, so we can apply some CSS style to it at a later time.

```
if (e.hasClass("required") && e.val().length == 0) {
  // error
  e.addClass(o.errorClass);
}
```

7. The same principle applies to numeric fields, except that we will want to check whether the field is empty or not, as the regular expression will fail even if there is nothing in it.

   The other thing that might cause some trouble is: how can we check if the user has inserted numbers only?

   Using regular expressions, this looks like a very simple joke:

```
if(e.hasClass("numbers") && !/^\d+$/.test(e.val()) && e.val().
length > 0) {
  e.addClass(o.errorClass);
}
```

8. E-mail address checking is mostly the same, except for the regular expression string, which is quite a bit longer.

   Also note that the official standard for valid e-mail addresses (known as RFC2822) is available at `http://tools.ietf.org/html/rfc2822` (Section 3.4.1) and is all but useless since it weighs too much and is way too complicated for everyday usage.

   Regular expressions to validate e-mail addresses can be found anywhere on the Internet and can be substituted for this expression in case they work better:

```
if(e.hasClass("email") && !/^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-
zA-Z]{2,4}$/.test(e.val()) && e.val().length > 0) {
  e.addClass(o.errorClass);
}
```

9. Now, on to the tricky bit, which is checking if a certain class exists and finding a number right next to the class name.

We first check for the element to have the right class name format (string—that is, "min"—immediately followed by a number) and then we select the first element of the array so obtained for a comparison against the actual length of the text contained in the box:

```
var p = this.className.match(/min(\d+)/i);

if(p && e.val().length < p[1]) {
  e.addClass(o.errorClass);
}
```

10. As a last touch, we have to prevent the form from being submitted, or users won't be given a chance to correct whatever was wrong.

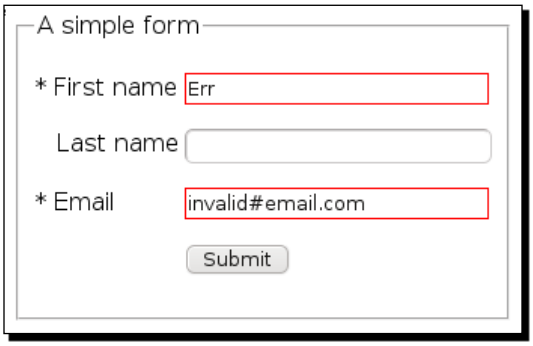
The jQuery `submit()` method submits the form when a flag variable is returned with a true value. On the contrary, if the variable value is false, the form won't be submitted at all and the user will be able to make the necessary changes and submit the form again.

So, adding a simple flag (`errorFlag`) that's increased anytime an error is detected and returned at the very end of the `submit()` function is enough to avoid undesired form submission.

11. As for CSS styling, we have little to add, but for plugins of bigger dimensions that make heavy use of custom classes, a separate stylesheet is usually required.

To make it work, though, we can just add this line to the HTML page inline styling and see the field borders turn red on error.

```
.error { border: 1px solid red }
```



Once everything has been put together, eventually the code will look like this:

```
(function($) {
  $.fn.formCheck = function(options) {
    var defaults = {
      errorClass: "error"
    };

    var o = jQuery.extend(defaults, options);

    return this.each(function() {
      var form = $(this);

      if(!form.is ("form")) return;

      form.submit(function() {
        var errorFlag = false;

        $(":input", this).each(function(index, element) {
          e = $(element);

          e.removeClass(o.errorClass);

          if(e.hasClass("required") && e.val() == '') {
            errorFlag = true;
            e.addClass(o.errorClass);
          }

          if(e.hasClass("email") && !/^[a-zA-Z0-9._-]+
            @[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/.test(e.val()) &&
            e.val().length > 0) {
            errorFlag = true;
            e.addClass(o.errorClass);
          }

          if(e.hasClass("numbers") && !/^\d+$/.test(e.val()) &&
             e.val ().length > 0) {
            errorFlag = true;
            e.addClass (o.errorClass);
          }

          var p = this.className.match(/min(\d+)/i);
          if (p && e.val ().length < p[1]) {
            errorFlag = true;
            e.addClass (o.errorClass);
          }
```

```
            });

            return !errorFlag;
          });
        });
      };
    })(jQuery)
```

Have a go hero – improve the user experience

Changing the border color is certainly a way to attract some attention to the wrongly filled textbox, but the user is left clueless about what the error is and how to correct it.

For every error, we should provide an explanation of why something went wrong and display the text right next to the erroneous field in the form.

Also note that the text should be easily modifiable by the user, who must be able to change any error description when initiating the plugin.

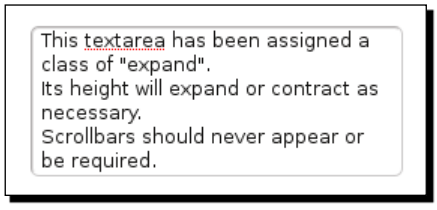
Does this plugin work with textareas as well?

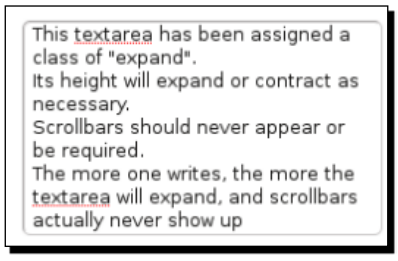
Make the necessary adjustments to add support for textarea elements in the same way we made input elements work.

Auto-growing textareas

When dealing with forms, users are (usually) supposed to enter information and fill all (or part) of the fields. For this reason, we actually have to take care of textboxes, which are the most versatile way of submitting information.

One of the simplest, yet most appreciated and useful, plugins we can develop to make sure the user gets a pleasant and original experience is about textareas. A common issue when writing text is not being able to see everything, or being limited to a certain size when speaking of textarea dimensions.



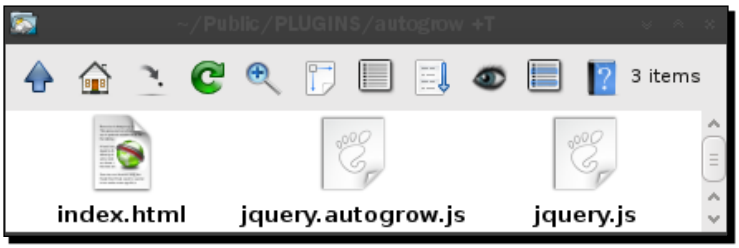


To get over this problem (which is nothing but a practical add-on that makes the user experience far more enjoyable), we are going to find a way to actually increase the height of the textarea we're writing in when necessary—that is, whenever a new line is added.

Time for action – creating the autogrow plugin

What we are aiming for is to find a way to make a textarea grow in size either indefinitely or between specific limits, as we might not want to obtain indefinitely long pages.

1. Our usual directory and files layout needs to be set up. The `autogrow` directory will contain the `jquery.js` file and the plugin file as well (`jquery.autogrow.js`).

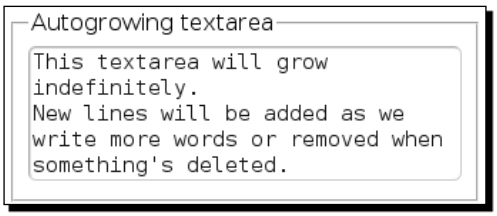


2. Our HTML file looks very simple, as we just need a textarea element to let the magic happen.

```
<!DOCTYPE html>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="jquery.autogrow.js"></script>
  <script>
    $(document).ready(function() {
      $(".autogrow").autoGrow();
    });
  </script>
</head>
<body>
<form action="#" method="post">
  <fieldset>
    <legend>Autogrowing textarea</legend>

    <textarea class="autogrow">This textarea will grow
indefinitely.</textarea>
  </fieldset>
</form>
</body>
</html>
```

3. Finally, we go on to our plugin file.

   Our plugin will provide only two options at this stage, which are `minHeight` (defaults to 0) and `maxHeight` (defaults to any high number close enough to infinity), respectively representing the minimum and maximum height that the textarea can assume after a size change due to new or deleted text.



4. Although the general overall structure is pretty much the same as for our other plugins, we will now have a closer look at what the code does and looks like.

   The basic code block for this plugin is the following:

```
var e = $(this);
var pValLength, pWidth, valLength, width, h;

if(!e.is("textarea")) return;

e.css("overflow", "hidden").keyup(function() {
  // more code
});
```

5. The above code is pretty simple but helps to understand the fundamental routes to get to the end of the development of this plugin.

   Apart from a couple of variables we'll need later, we're checking whether the element we're applying the plugin on is actually a textarea or not, or we won't be able to do anything with it.

   Also, we have hidden any overflow by default, as we will end up with a long scroll-less textarea.

6. This part is rather JavaScript-heavy, which means some knowledge about JavaScript functions wouldn't hurt at all, since the plugin is based on some element properties that are accessed through jQuery methods but eventually refer to DOM properties:

```
valLength = $(this).val().length;
width = $(this).attr("offsetWidth");

if(valLength < pValLength || width != pWidth) {
```

```
  $(this).height(0);
}

h = Math.max(o.minHeight, Math.min($(this).attr("scrollHeight"),
o.maxHeight));

$(this).css("overflow", ($(this).attr("scrollHeight") > h ? "auto"
: "hidden")).height(h);

pValLength = valLength;
pWidth = width;
```

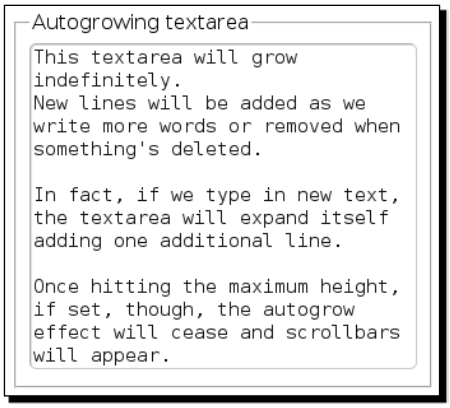
7. And that's it!

Once we have taken care of changing the size of the textarea depending on the need (also checking against the previous dimensions), we only have to update the current height needed and switch around the variables to get ready for the very next character typed in.

The following is the resulting code and the resulting sample screenshot:

```
(function($) {
  $.fn.autoGrow = function(options) {
    var defaults = {
      minHeight: 0,
      maxHeight: 9999
    };

    var o = jQuery.extend(defaults, options);

    return this.each(function() {
      var e = $(this);
      var pValLength, pWidth, valLength, width, h;

      if(!e.is("textarea")) return;

      e.css("overflow", "hidden").keyup(function() {
        valLength = $(this).val().length;
        width = $(this).attr("offsetWidth");

        if(valLength < pValLength || width != pWidth) {
          $(this).height (0);
        }

        h = Math.max(o.minHeight, Math.min($(this).attr
              ("scrollHeight"), o.maxHeight));
```

```
            $(this).css("overflow", ($(this).attr("scrollHeight") >
                          h ? "auto" : "hidden")).height(h);

            pValLength = valLength;
            pWidth = width;
          });
        });
      };
})(jQuery)
```



What just happened?

The final product might have amused you and, at the same time, made you wonder how this was all possible.

But let's have a closer look at the code and understand what has been going on to begin with. After all, nothing here is really complicated, just a little different to what we were used to until now.

The whole point in this plugin is to play a bit with element properties in order to adjust the textarea dimensions according to our needs (that is, more text being written in).

In the very first part everything should look clear. We first check whether the element we have called the plugin on is actually a textarea element (or the whole thing wouldn't make sense at all). The next step is to disable scrollbars. Now that the textarea increases in size as more content is added, who needs them after all?

We thus bind the `keyup` event to a simple routine by means of which we can change the textarea size with ease.

And we've finally got to the trickiest part. Here, our aim is to check if the height of the textarea needs to be increased due to a different `scrollHeight` value (that is, the "height" of the object's content).

Note that this number doesn't always change.

For example, if we start writing a lot of words into the textarea and reach a `scrollHeight` of, say, 200 pixels but we suddenly realize something is wrong in the text and we go on and delete half of it, the most logical guess is to suppose the `scrollHeight` value would be cut down to 100.

Unfortunately, we're wrong and a value of 200 is still considered as the value of `scrollHeight`. Needless to say, this behavior compromises our plugin if we don't take care of it and actually reset the height (setting it to 0) if the number of characters present after the `keyup` event fires is lower than the number of characters we had previously.

Have a go hero – improved autogrow

If you thought that the plugin can be considered finished, you haven't a grasp of jQuery's options yet!

A very interesting thing to do in order to obtain some more features and options, which just about any person would love to see in this type of plugin, is to apply some sort of effect to it. I don't mean adding purple lighting or a ringing bell, but a very simple fading or scrolling effect would be effective.

Also, if you're looking at the jQuery documentation, you can also figure out how to check the number of characters, words, or paragraphs written into the textarea. This can be used to compute and create a simple validation function of your own to make sure that a user doesn't enter more (or less) than a certain number of characters, words, or paragraphs into a single textarea.

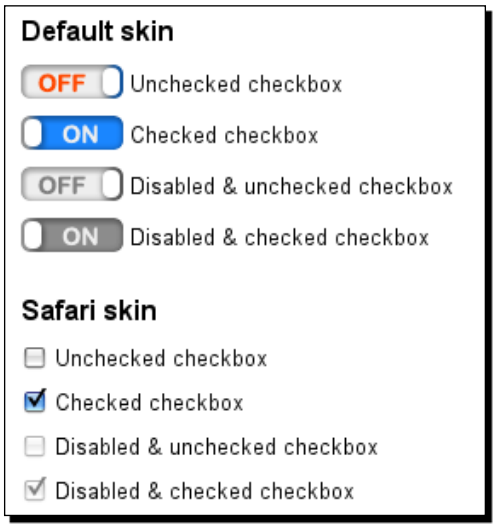
Other types of form-related plugins

Of course, there are other kinds of form plugins, as jQuery (and JavaScript) can manipulate nearly everything a web page has to offer. Lots of plugins (operating, in fact, on every aspect of the HTML code and providing visual aid, too) have been released.

Checkboxes and radio buttons

Checkboxes and radio buttons both play an important role in forms, in any situation involving multiple choice questions (just think the usual "gender" option, for which you have to select either "male" or "female"—or "undefined"). These elements are required to avoid a number of ambiguous answers. What would happen if, when asked to answer "are you male or female?" some funny guy replies with "whatever" or "ask your mum", having being given the possibility to actually enter information into some kind of textbox?

The problem arises when the desire to style, and/or add some kind of control over, the checkbox or radio values first comes to our mind. For example, wouldn't it be cool to change the look and feel of checkboxes and radio buttons? Well, by using some jQuery code together with a bunch of CSS instructions this is possible. Results similar to the following, obtained by using Window Maker, can be easily obtained:



However, you may be wondering: "How can this be accomplished?".

Firstly, through some CSS code, we must hide the original checkbox, and replace it with our own image or whatever.

We should also assign to our newly created "checkbox-replacement" element an ID from which we can easily get the original checkbox's ID.

For example, we may want to replace all the checkboxes with images whose IDs will be those of the checkboxes plus the string `-replacement`.

```
<input type="checkbox" id="mycheckbox1" />
<input type="checkbox" id="mycheckbox2" />

<!-- Our images would then be something like this -->
<img src="mycheckboximage.png" id="mycheckbox1-replacement" />
<img src="mycheckboximage.png" id="mycheckbox2-replacement" />
```

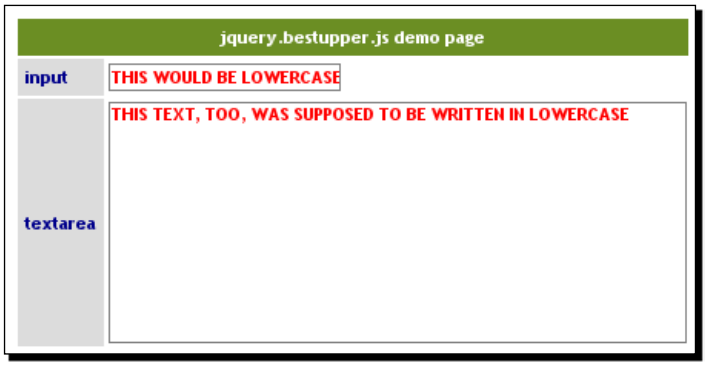
Whenever the user clicks on an image (that is, checkbox), something should happen to let him or her understand that he or she has actually (un)checked a checkbox. We then need to check, or uncheck through JavaScript, the corresponding hidden checkbox and change the image displayed according to the checkbox state:

```
$(".image-checkbox").click(function() {
  var $cb = $('#original-checkbox-id);
  var isChecked = $cb.is(":checked");

  // Toggle state
  $cb.attr("checked", !isChecked);

  // Change image
  if(isChecked) {
    $(this).attr("src", uncheckedImage);
  }
  else {
    $(this).attr("src", checkedImage);
  }
});
```

Text manipulation

Another very common problem is dealing with text and words of different origin. Sometimes we would like users to enter all lowercase or uppercase text due to software requirements or personal liking.

Mustafa Özcan has released a plugin for transforming all input into uppercase.



A similar result can be obtained in the old-fashioned way by creating two arrays, one containing lowercase characters and the other only uppercase ones.

Obviously, the trouble here is to catch the key the user has actually pressed by using the `keypress()` event to check what has been hit (either the keycode or ASCII character or both):

```
$("#myinput").keypress(function(event) {
  var keycode = event.keyCode;
  var ascii   = event.which;
});
```

Our next problem is to locate the character and replace it:

```
var i = $.inArray(ascii, lowercase_array)

// found in array: it's a lowercase character
if (i > -1) {
  // remove last character
  var newText = input.val().slice(0, -1);
    // add uppercase character
  input.text(newText+uppercase_array[i]);
}
```

Obviously, this rough approach can be made prettier and, more importantly, an option to be able to do the opposite (that is, transforming uppercase to lowercase) would be really neat and appropriate.

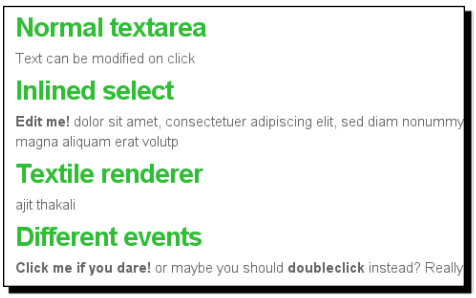
Edit in place

Believe it or not, creating an edit-in-place plugin with jQuery is simpler than it may seem.

But first, what exactly is this edit-in-place thing?

Just imagine that you click on any text on the page and a textbox pops up to allow you to edit the content. Also, once you have modified it and click on **Submit**, the updated information is displayed in the page and copied over to some database.

This is how edit-in-place plugins work:



We begin by replacing the text block with an input field, which will contain the original text and save it on blur:

```
$(".editable").click(function() {
  var e = $(this);
  var $input = $('<input type="text" />').attr("value",
                 e.html()).blur(function() {
    e.html($(this).attr("value")).show();
    $(this).remove();
  });

  e.hide().parent().prepend($input);
});
```

In fact, creating new elements with jQuery is as simple as writing the HTML into `$()`, as we would to select any element. This creates a new jQuery object that we can treat like any other, thus being allowed to attach events and methods to it.

The previous code snippet modifies the text (that is, hides the input and displays the text element) when the input loses focus.

Another approach to this problem is to create a form instead and change the text once the submit button is clicked on, thus leading to a more controlled and logical behavior.

Also, updated content is not saved anywhere in the above example, making it rather useless in the real world. Who would want to change some text knowing nothing actually changes on the server side?

For this reason, we may want to introduce a PHP script to redirect our form action to. The PHP page will then update the database or files whereas the jQuery plugin will have to fetch the content once more through an AJAX call or just copy the form content—the database will be accessed once less but will eventually step in on page refresh.

Have a go hero – add more options

We had a somewhat detailed look at the various types of plugins dealing with forms, and we also should have clear in mind what the difficulties are when dealing with this kind of plugin.

Based on the above instructions and code snippets, choose one plugin type we have learned of and develop a working example of it. Use the jQuery website, books, and ultimately the Internet as reference.

A very important thing to bear in mind, though, is to always look for the easiest way to do things (at least at first). Eventually, improve things by adding features, new options, and sick tweaks to get a really nice-looking and smooth plugin.

Pop quiz

1. Our attempt at creating a jQuery plugin for validating forms was mainly based on an already existing jQuery plugin called Validation, which does not provide a sufficient level of accuracy when it checks submitted forms. Is that correct?

   A. Yes, absolutely.

   B. Yes, even though the Validation plugin is also included by default into the latest jQuery release.

   C. No, the effectiveness and quality of the Validation plugin has been proved by its inclusion in the jQuery releases. Our attempt is just for educational purposes to demonstrate how such results can be accomplished using jQuery methods and syntax.

   D. Not sure.

2. In the implementation of our own validation plugin, we have made use of regular expressions to check the format of a particular string that has been entered into the selected text box.

   How does the `test()` function behave? What are the return values of the function we need to look for in order to understand that we actually have a match?

   A. `test()` is a method that takes two arguments: the regular expression and the string in which it looks for the regular expression.

   B. `test()` is a jQuery method that requires one argument, that is, the string to be searched and returns true if a match is found or false otherwise.

   C. `test()` is a JavaScript method that requires one argument; that is, the string to be searched and returns false if a match is found or true otherwise.

   D. `test()` is a method requiring the string to be looked for as its only argument.

3. In order to show errors to the users, our form validation plugin has to prevent the form from being submitted if any error is found.

   In addition to the error flag, which is certainly useful in this case, we also have to do some other things for the submission cancellation to occur.

   A. Submission can be avoided by either calling `.preventDefault()` on the event object or returning false.

   B. Submission can be avoided by either calling `.preventDefault()` on the event object or returning true.

C. Nothing. The error flag is enough, if set to false, to tell jQuery not to submit the form when the `submit()` event occurs.

D. Nothing. The error flag is enough, if set to true, to tell jQuery not to submit the form when the `submit()` event occurs.

4. Consider the following code, which we used in our form validation plugin. What is the content of the variable `p` most likely to be?

```
var str = "first-class min6 test";
var p = str.match (/min(\d+)/i);
```

A.
```
p[0] => "min6"
p[1] => "min"
```

B.
```
p[0] => "first-class min6 test"
p[1] => "min6 test"
p[2] => "min6"
```

C.
```
p[0] => "first-class min6 test"
p[1] => "min6"
p[2] => "6"
```

D.
```
p[0] => "min6"
p[1] => "6"
```

5. In the writing of our autogrow jQuery plugin we have used the `scrollHeight` property (and `scrollWidth` too) more than once.

   What does the property refer to?

   A. It's the height (or width) the scrollbar would be if we had not hidden them.

   B. It's the distance between the top and bottom (or left and right) edges of the object's content in pixels (px).

   C. It's the same as `height()`.

   D. It's the distance between the top and bottom (or left and right) edges of the object's content in points (pt).

6. Our autogrow plugin makes use of the `keyup` event. Why is this event preferred over the `keypress` event, which would result in a more logical conclusion, if we think the user might hold down the key and the textarea would then increase in size only once the key is released?

   A. Because the `keypress` event occurs before the character is actually displayed in the textbox, resulting in `scrollHeight` not being accurate at that point.

   B. Because the `keypress` event is significantly slower than the `keyup` event and the text would flicker and/or the resulting size would be inaccurate.

C. There is no reason, actually. It's a matter of personal taste, even though `keypress` is not supported by some older browsers and may be incompatible with slower computers.

D. None of the above.

7. Is there any difference between `scrollHeight` and `innerHeight()`?

   And if so, what would be the most accurate value to use in a jQuery plugin similar to ours?

   A. `innerHeight()` is a JavaScript method that returns the height of the element, including top and bottom padding, in pixels.

      We do care about padding in this case, so `innerHeight()` is our best choice.

   B. `innerHeight()` is a jQuery method that returns the height of the element, top and bottom padding excluded, in pixels.

      We don't care about padding in this case, so `innerHeight()` is our best choice.

   C. `innerHeight()` is a JavaScript method that returns the height of the element, top and bottom padding excluded, in pixels.

      We don't care about padding in this case, so `scrollHeight` is our best choice.

   D. `innerHeight()` is a jQuery method that returns the height of the element, including top and bottom padding, in pixels.

      We don't care about padding in this case, so `scrollHeight` is our best choice.

8. We have also used the `data()` method to store information into elements.

   How does this method actually work and what are some of its (dis)advantages (if any)?

   A. It stores arbitrary data associated with the matched elements.

      Unfortunately, the method is not compatible with most modern browsers, though.

   B. It stores arbitrary data associated with the matched elements.

      Functions (and references to functions) cannot be stored using a `.data()` call.

   C. It stores arbitrary data associated with the matched elements.

      Functions (and references to functions) can also be stored using a `.data()` call.

   D. It stores text-like data associated with the matched elements.

9. In order to develop a working edit-in-place plugin, what must we absolutely take care of if we intend to submit the modified information to a database?

   A. **Security**—information should be transmitted in a safe way.
   Speed—modified content should move rapidly from the client side to the server side.
   Interface—form styling and look and feel are fundamental.

   B. **Security**—information shall be transmitted in a safe way.
   Server-side scripting—a PHP (or equivalent) script should be used as form target so information is processed and submitted to the database for a change.
   Client-side validation—to prevent wrong content from being submitted and possible errors from being saved.

   C. **Client-side validation**—to prevent wrong content from being submitted and possible errors from being saved.

   Security—information should be transmitted in a safe way.
   Interface—form styling and look and feel are fundamental.

   D. **Valid HTML code**—to avoid any kind of misunderstanding as jQuery selectors are concerned.
   Server-side scripting—a PHP (or equivalent) script should be used as form target so information is processed and submitted to the database for a change.
   Security—information should be transmitted in a safe way.

10. Are keycodes recognized by JavaScript?

    And if yes, how can the developer access and make use of them, and in which form?

    A. They cannot be accessed using JavaScript only, and other frameworks and/or programming languages should be used to obtain better results.

    B. Keycodes can be accessed through the `keycode` or `which` properties of any event regarding keys being pressed.

    C. Keycodes can be accessed through the `keycode` or `which` properties of any event.

    D. Keycodes can be accessed only through the `keycode` property of any event regarding keys being pressed.

Summary

In this chapter, we have seen how form plugins of different kinds can be realized and, in particular, we have had an in-depth look at how form validation plugins and textarea autogrow plugins can be developed.

Also, some other interesting ways of interacting with forms have been presented in the form of code snippets and examples given for some of the most common and widespread types of plugins the average user usually deals with on a daily basis.

Ranging from checkboxes and radio button improvements to an interesting realization of a simple edit-in-place plugin allowing in-place editing of a text paragraph, we have also seen and learned how many other useful jQuery plugins, such as the case changer one, actually work and find application in the real world.

With particular interest, we also had a chance to see many new methods and functions at work in these plugins, allowing for a more detailed and expert understanding of how jQuery works when speaking of plugins.

As a side note, the increasing importance and multiple roles the element `$(this)` plays in plugin development when we're selecting elements should be noticed, as its role and meaning in the jQuery syntax and coding standards is worth a second look.

Unfortunately, due to space issues, as always, we haven't been able to cover everything form related, even though we did our best to get our hands on a little of everything. With much regret, we left out some very interesting plugin ideas, which are anyway available on the Internet and the jQuery plugin directory for those of you particularly keen on the topic.

For example, honorable mentions go to date picker plugins, advanced validation or file uploads, and AJAX form submission. The majority of these topics, though, are quite complicated to deal with and would be far beyond the scope of this chapter and book.

Speaking about chapters, though, we're quickly approaching a User Interface chapter (that is, the next chapter), which will provide a first overview of what User Interface plugins are like, in terms of objectives, requirements, and user interactions required to get the best out of them.

Immediately following, a couple of chapters will cover tooltip plugins and menu and navigation plugins, each guiding you through the creation of a quality plugin, which can be then easily expanded and modified at a later time.

8
User Interface Plugins

When thinking about a user interface, one may not understand the importance that the concept actually has when it comes to interface design and user experience—that is, every time we decide to put a website or web page online.

No matter how good the content might really be, if the user does not have the impression of a nice, pleasant website, he or she will most likely never come back a second time.

The reason is actually extremely simple and perfectly understandable. In fact, nobody could objectively like or think any good of some fundamental elements (menus, forms, and so on) randomly scattered around the page without even a tiny bit of coherency or common sense.

And a poor user interface means total dislike for the product. And something not appreciated is surely not going to sell (or guarantee any visits), no matter how good its features or content.

Looks and feel come first, or, at least, play an important part when we have to decide whether something is worth our attention or not.

We're now going to cover in detail the following topics:

- Positioning
- Setting equal heights
- Other examples of user interface plugins

As far as the looks of a web page are concerned, a talented web designer will certainly improve things and make everything look amazing. But the user interface is a somewhat more tricky issue.

For example, navigation menus are a true pain for most websites, as they fail at providing the user with a really useful menu. On the other hand, tooltip plugins are an incredibly neat and interesting solution to offer some sort of quick explanation without breaking the page content.

Of course, we want the user experience to be the best our guest has ever tried, but we are using jQuery to achieve this goal, which has some drawbacks—first of all, most of what is related to the user interface can be modified and improved using just CSS.

However, our plugins will not create anything that would make ourselves look just plain stupid or inexperienced for not having used a couple CSS instructions instead of two pages of JavaScript code.

What we're going to learn in this chapter is how some of the most common goals (in terms of user interface and interaction) can be achieved using jQuery plugins. These plugins can significantly improve the look and feel of most of the elements the user is usually bound to interact with.

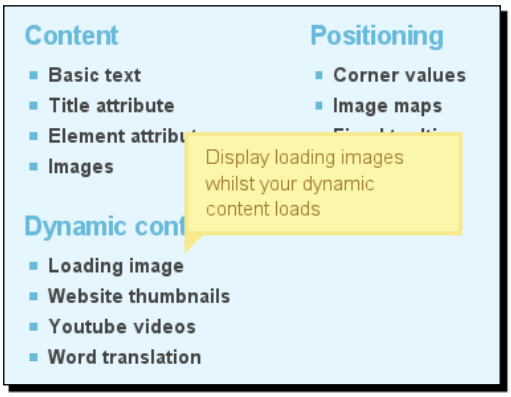
Positioning

If we were to explain what tooltip plugins do, in a couple of lines, we would be most likely to say that they allow the user to create tooltips with more or less ease on nearly any element on the page.

In fact, using jQuery selectors and its own event management features, this type of plugin can completely replace the dull tooltip that automatically pops out whenever a title attribute has been set.

Most of the plugins available also offer many advanced options and features to choose from. These range from a variety of visual effects to styling and fixed/variable width of the tooltip.

One of the most popular solutions is the qTip plugin by Craig Thompson (`http://craigsworks.com/projects/qtip/`). It provides a package containing everything a tooltip user would expect from a jQuery tooltip plugin, including the possibility to change the look and feel of the displayed tooltips and control their behavior in a rather specific way.



We're now going to have a look at, and understand, how this particular kind of plugin works.

First of all, our main concern might be along the lines of "How do I get to know where to display the tooltip and how can I make it show up?".

As for the latter, a shrewd reader would probably immediately raise their hand to point out that there is no problem at all, since we have already faced similar issues earlier in the book and, thanks to the help of some handy methods, we have finally written some code allowing us to control the exact position and displaying modes.

To get to a working solution for the first problem as well, we have to mess with JavaScript events in order to obtain the mouse cursor position. We will have a closer look at this in the next chapter, dealing with tooltip plugins in detail.

For the moment, reading some slightly outdated (but still very useful) articles by Peter-Paul Koch on Quirksmode (`http://www.quirksmode.org/js/events_properties.html#position`) and Evolt (`http://evolt.org/article/Mission_Impossible_mouse_position/17/23335/index.html`) is enough.

Time for action – understanding mouse movement events

As a demonstration, we're going to create a simple jQuery script that, when hovering the mouse pointer over any element, gives back the X and Y coordinates the mouse cursor is located at.

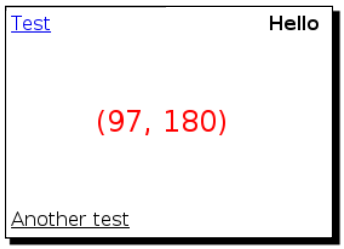
1. You may fill the HTML file with any element you like, this does not matter in terms of the end result, but it's fundamental you include a `span` element having a class `result`:

```
<span class="result">(x, y)</span>
```

2. The code to be written into the plugin file, instead, is something along the lines of the following, and should be called in the `index.html` file:

```
(function($) {
  $.fn.xy = function() {
    return this.each(function() {
      $(this).mousemove(function(event) {
        $(".result").text("(" + event.pageX + ",
          " + event.pageY + ")");
      });
    });
  };
})(jQuery)
```

3. Now, every time we hover over an element, the label will update with the current position. Also, the same happens if we happen to move the cursor on that very same element, to simulate the tooltip moving according to mouse cursor placement.



Have a go hero – improve the plugin

As we need to do this sooner or later (in next chapter, by the way!), you can now try to display any element in any area of the page (so it's not necessary to position the "tooltip" element right next to the cursor for now) once the mouse pointer hovers over the tooltip target.

For example, take an image. Then, make sure you append or insert the HTML code, which is needed for the image to show up when the mouse pointer hovers over the selected element.

The next step is to apply some sort of effect (sliding or fading) to it.

Also, you might want to make the element disappear (with some other effect) once the mouse pointer leaves the element.

The 'mouse out' event

Even though there is no such thing as a "mouse out" event in jQuery, there is the possibility to attach a function to a mouse out event in some way.

In fact, the `hover` event may take two arguments:

```
$(element).hover (mouseOverFunctionPointer,
mouseOutFunctionPointer););
```

Anonymous functions are also supported (obviously!).

Setting equal heights

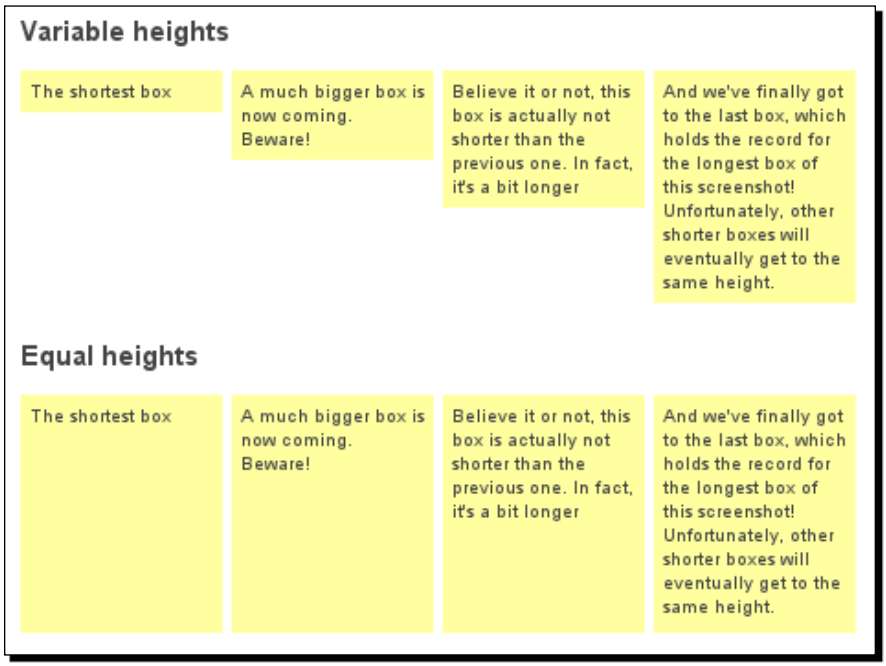
Who can state that they have never been presented with the problem of making two or more columns of the same height? Designers are terrified by this.

Yeah, I can almost hear all the CSS purists out there advocating the cross-browser compatibility and the consistency of a web page by avoiding any other code block, and eventually criticizing the need for JavaScript to be enabled. But let's face it, since table-based layouts have been abandoned for the more flexible, visually appealing, modern design guidelines, we've failed to see many HTML+CSS equal-height columns.

Indeed, JavaScript offers a simple, quick, and unobtrusive workaround, which is worth the effort even in terms of usability and performance.

However, with the so-called faux-columns technique, introduced by Dan Cederholm on *A List Apart* (`http://www.alistapart.com/articles/fauxcolumns/`), CSS can actually be used to create equal-height columns, even though this technique is best suited for entire page layouts, whereas it might seem relatively tricky when we have to deal with small boxes or parts of the page content.

It should be noted that the above-mentioned CSS-only solution works even if JavaScript is disabled, of course!



The solution in this case is rather simple.

Time for action – setting the same height

In this example, we will check which box is the tallest, then store the said value in pixels, and make all the other boxes of the same height.

1. Create the files we're used to working with, paying particular attention to the HTML file.

   This time, we have specific requirements for its markup, since we need a container element to put equal-height divisions in, as we're going to make our script work on children of the selected element(s).

2. The type of code we're looking forward to putting into the body element of our file is something like:

```
<h1>Variable heights</h1>
  <div class="container">
    <div class="column">
```

```
    The shortest box
  </div>

  <div class="column">
    A much bigger box is now coming.<br />Beware!
  </div>

  <div class="column">
    Believe it or not, this box is actually not shorter than the
    previous one. In fact, it's a bit taller.
  </div>

  <div class="column">
    And we've finally got to the last box, which holds the
    record for the tallest box of this page!<br />
    Unfortunately, other shorter boxes will eventually grow
    to this very same height.
  </div>
</div>
<br style="clear: left;">

<h1>Equal heights</h1>
<div class="container equal">
  <div class="column">
    The shortest box
  </div>

  <div class="column">
    A much bigger box is now coming.<br />Beware!
  </div>

  <div class="column">
    Believe it or not, this box is actually not shorter than the
    previous one. In fact, it's a bit taller.
  </div>

  <div class="column">
    And we've finally got to the last box, which holds the
    record for the tallest box of this page!<br />
    Unfortunately, other shorter boxes will eventually grow
    to this very same height.
  </div>
</div>
```

3. The equal-height script will take care of selecting, one at a time, the children of the DIV we're calling the method on. It will apply the height of the tallest column to the other elements (siblings):

```
(function($) {
  $.fn.equalHeight = function() {
    return this.each(function() {
      var tallest = 0;
      var e = $(this);

      e.children().each(function() {
        tallest =  Math.max ($(this).height(), tallest);
      }).css({"height": tallest});
    });
  };
})(jQuery);
```

Also, we might want to add some CSS code to style the columns, in a separate stylesheet (`jquery.equalheight.css`):

```
body {
  color: #333;
  font-size: 13px;
  line-height: 1.4;
}

.column {
  float: left;
  background-color: #cdeb8b;
  width: 160px;
  margin: 10px;
  padding: 10px;
}

.column:first-child {
  background-color: #deb1de;
}

.column:nth-child(2) {
  background-color: #c3d9ff;
}

.column:last-child {
  background-color: #ffff88;
}
```

'Each' loops

Also note that every time we enter a different element selection (that is, in this case, we create a new `each` loop) the jQuery object `$(this)` points to the changes. Assigning it to a variable at the very beginning of the code will not only quicken up the script, but will also help avoid confusion when a lot of different loops are needed and nested in the code.

Other examples of user interface plugins

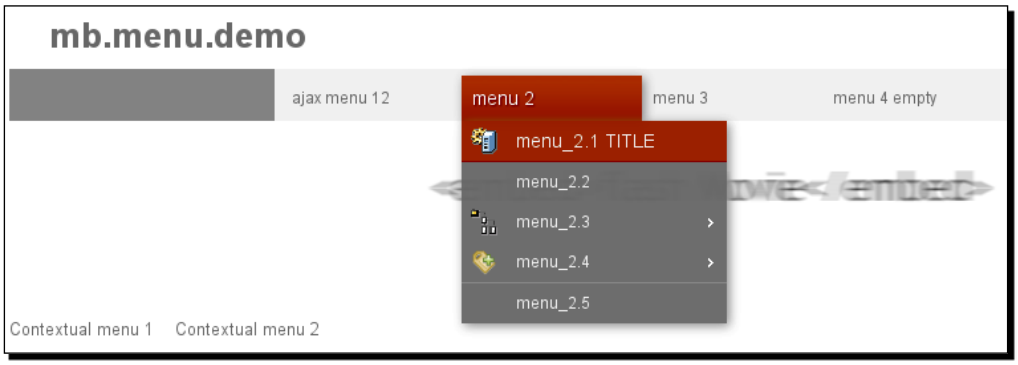
Besides the plugins that we have tried and analyzed earlier in this chapter, a lot more different scripts are available out there. These scripts offer some quality solutions to common issues and improve the overall user experience through the creation of a better interface or by tweaking the existing one to avoid various bugs, misunderstandings, and what not.

Menu plugins

Given their importance on the Web, menus should be handled with extra care.

Actually, the exact opposite happens: how many times have we happened to see a menu leading to nowhere, or a list of links scribbled down somehow and rather useless for the end user?

Many of the menu-related plugins available are aimed at improving the style of the menu itself, or providing an easy and foolproof way to build tree menus or even multi-level menus in seconds.

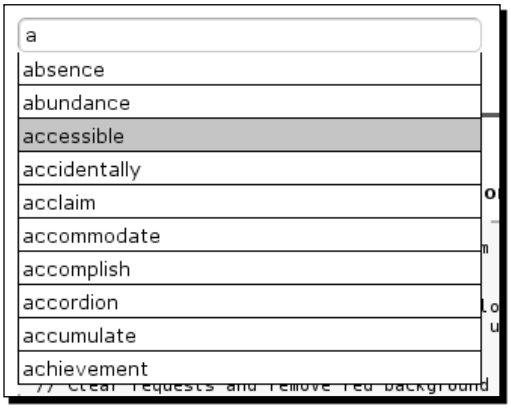


We will see how we can realize a jQuery menu plugin later in the book (Chapter 10), with particular focus on how to enhance the user interaction with these elements and how to obtain a nice-looking, easy to use and set up menu plugin.

Form enhancement plugins

Along with form plugins and accessibility plugins come the form enhancement plugins that are successful in delivering a more enjoyable user experience.

All of those form-related plugins fall in this category with the specific goal of making things easier to use, understand, and work with for the end user, such as autotab features, auto-completing fields (see the image below), input suggestions, and so on.

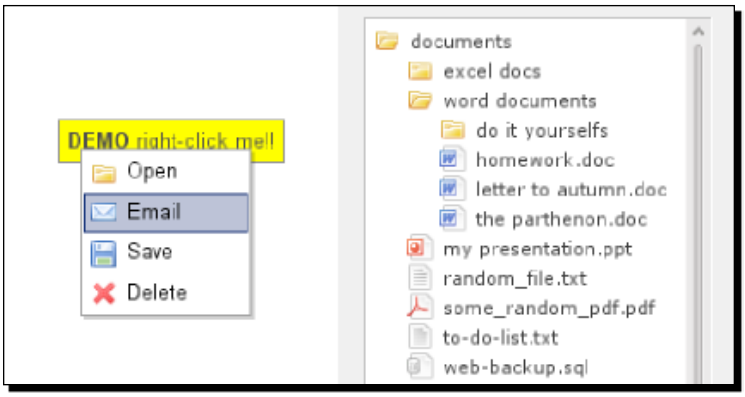


Context menus and tree menus

Sometimes, what the author of a plugin says is worth more than a hundred pages of approximate explanation and several turns of phrase. So, directly from his web page (`http://abeautifulsite.net/2008/03/jquery-file-tree/`), the creator of the jQuery plugin File Tree, Cory LaViska, says:

> *jQuery File Tree is a configurable, AJAX file browser plugin for jQuery that lets you create a customized, fully-interactive file tree with as little as one line of JavaScript code.*
>
> *Server-side connector scripts are used to read the file system on your server and return data to the client side script via AJAX.*

This is undoubtedly of great help to those willing to make a navigation menu from directories. Even if you want to let users browse some of your online folders, this plugin might come in useful.



Another interesting type of plugin dealing with user interface problems involves the creation of context menus, which overwrite the default mouse right button click event.

The inability to use the mouse buttons at one's will is something really annoying for the majority of people, which could lead to the user not visiting the page again. Of course, this does not apply to the web application, which on the contrary enhances the user experience by activating right-click menus for file browsing and similar tasks.

But using context menu plugins with a little bit of common sense can probably lead to a good integration of the latter with the above-mentioned file tree plugin.

The end result would, of course, allow the users to act very much as they would on their own desktop using their favorite file browser. It's worth giving it a shot!

Pop quiz

1. To create a tooltip plugin we need to make use of some events occurring if the mouse pointer moves over some particular spots.

   Namely, we want to use `hover` and `mousemove`.

   If we were to move the mouse over a very big division element and then move it around an indefinite number of times, how many times will the two events be triggered respectively?

   A. The `hover` event would be triggered once (when the mouse cursor enters the DIV); the `mousemove` event is triggered every time the cursor moves, so it's impossible to tell with this information.

   B. The `hover` event would be triggered once (when the mouse cursor enters the DIV). The `mousemove` event would be triggered once, too, if we moved the mouse without stopping at any time.

C. The `hover` event would be triggered constantly as long as the mouse cursor is over the element. The `mousemove` event would be triggered once, if we moved the mouse without stopping at any time.

D. The `hover` event would be triggered constantly as long as the mouse cursor is over the element. The `mousemove` event is triggered every time the cursor moves, so it's impossible to tell with this information.

2. Theoretically, a tooltip should disappear once the mouse cursor moves off the element we chose as the tooltip target.

   How can we recognize that, in fact, the mouse cursor has been moved away from the element and that the tooltip should be removed or deleted?

   A. We can't. That's why we wait until another element that has been assigned a tooltip is hovered so the previous tooltip disappears and the new one is shown instead.

   B. Adding another (anonymous) function as second argument of the `hover` event tells jQuery to run the contained code once the mouse moves out of the selected element.

   C. Using the `mouseoff` event, which jQuery provided by default, and is triggered whenever the mouse pointer, after having been over the selected element, is moved away from it.

   D. None of the above.

3. Consider the following code:

```
$(selector1).each(function() {
  $(this).css({"color": "red"});
    $(this).children().each(function() {
    $(this).parent().css({"color": "blue"});
    $(this).css({"color": "green"});
    $(this).siblings().each(function () {
      $(this).css({"color": "yellow"});
    });
  });
});
```

   What color would each of the elements eventually have?

   A. Selector1: blue
   each child: green

   B. Selector1: red
   each child:green
   last child: yellow

C. Selector1: blue
each child: yellow
last child: green

D. Selector1: red
each child: green

4. Speaking of equal height columns, is it possible to achieve similar results without any JavaScript script?

   A. No, JavaScript is the only way to go if we really look forward to having equal-height columns in our website layout.

   B. No, but we can use some server-side script (such as PHP) to tweak the user interface of the website so it looks like columns are about the same height, even though they are actually not. This solution should be used with care, though.

   C. Yes, even in many modern CSS-only designs equal-height columns are widely made use of, and many currently running websites actually have a very well coded CSS script that makes the magic happen.

   D. Yes, but only using the so criticized old table-based layout approach or through the use of images to create backgrounds in the container element can similar results be achieved.

Summary

This chapter has shown and explained what user interface plugins are and what are they supposed to take care of in the real word.

However, very little has been said about the actual ways we can develop them, mainly because a quick, theoretical approach helps out more than immediately plunging into development without any knowledge of the matter whatsoever.

To sum up what we have learned so far, we might want to notice the way tooltip plugins are gaining more and more importance as time goes by, especially thanks to their possibility of turning back into standard, unstyled tooltips if the user has turned JavaScript support off.

Also, another important role in the Web user interface is played by menu plugins. The goal of these plugins is to improve the behavior of navigation elements in order to make the user aware of their position and enhance the look, feel, and general first impression (which always counts) that the user gets of the page, after a couple of minutes of browsing.

In the next few chapters, we're going to discuss what we can do to further improve the user interface and experience of a generic web page. This will be done through the realization of a tooltip plugin and a totally different approach to menu integration and improvements.

The next chapter specifically deals with tooltip plugins, and aims to report, in detail, some examples of already existing plugins that work very well. The chapter will guide you through the process of the development of an aesthetically pleasant solution that lets the user change, modify, and customize options, colors (and looks in general), and behavior rather easily thanks to a slightly more advanced way of handling options.

Read on!

9
User Interface Plugins: Tooltip Plugins

We might want to start off by saying tooltips are very popular in today's web design. This is probably due to the value that the tooltips add to the overall look of a website and the sensible addition to a nice user experience that they will certainly contribute.

Also, as we have seen for many other plugins, which are not that difficult to create—provided we know exactly what to do and how to do it—there shouldn't be much of an issue with developing the tooltip plugin. It may actually take some time to understand how to position elements based on the mouse cursor position itself and how to move deftly with an increasing number of functions doing different things. However, we're now on the path to becoming experts, and fearless too!

The topics we're going to discuss include:

- Tooltip plugins in general
- Positioning the tooltip
- Merging pieces together
- Custom jQuery selectors

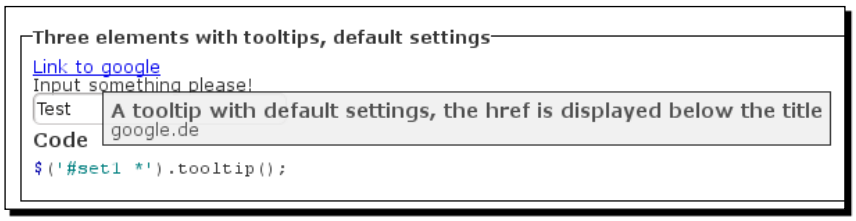
Before we get started, there is another little thing worth mentioning: we have decided to deal with tooltips and menus (in the next chapter) in particular detail. This has been done not only because these two topics are some of the most discussed and certainly stir up some curiosity, but because they also provide many different opportunities (more than other type of plugins, at least!) to introduce new concepts and ideas, even while keeping the complexity of the whole plugin at a minimum.

We can now go on to create our plugin, starting with basic functionalities, and subsequently adjusting its goals. We will add new, improved functionalities that, however, do not make the whole code look too difficult to understand—even after some time or for someone who's just starting out with jQuery.

Tooltip plugins in general

A lot has been said about tooltip plugins, but it's worth repeating the most important points with particular regard to the way tooltips are supposed to work, and how we want our tooltip to behave.

First of all, we might want to get an idea of what tooltips look like and a sample of what we will accomplish by the end of this chapter. Here is an example:



Also, with some more work and proper application of effects, images, and other relatively advanced techniques, we can also obtain something more complex and nicer looking, thus giving the user the chance to specify the style and behavior for the tooltip, as follows:

The idea is actually very simple. The elements we have selected will trigger an event every time we hover the mouse pointer over them.

The tooltip will then pop out, right at the mouse cursor position, retrieving the text portion from the `title` attribute of the said element.

Finally, whenever we move the mouse over the same element, the plugin will move and follow the mouse cursor until it goes off the boundaries of the element.

Positioning the tooltip

The first problem we have to face is, of course, how to make the tooltip appear in the right position.

It would be no trouble at all if we just had to make some text, image, or anything else show up. We've done it many times and it's no problem at all—just make their positioning absolute and set the right top and side distances.

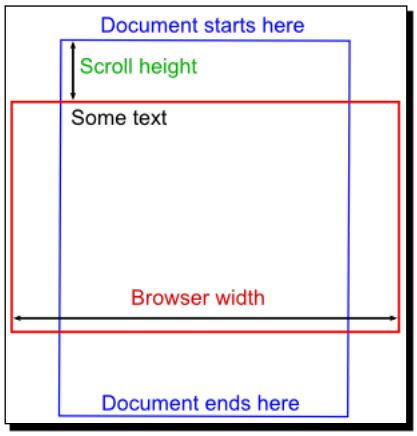
However, we need to take into account the fact that we don't know exactly where the mouse cursor might be and, as such, we need to calculate distances based upon the mouse cursor position itself.

So, how can we do it? It's simple enough; we can use some of the JavaScript event properties to obtain the position. Unfortunately, Internet Explorer always tries to put a spoke in our wheel.

In fact, the magnificent browser does not (according to this table, which is quite accurate: `http://www.quirksmode.org/dom/w3c_cssom.html#mousepos`) support *pageX* and *pageY*, which would normally return the mouse coordinates relative to the document.

So we need to think about a workaround for Internet Explorer, as jQuery (from version 1.0.4 onwards) does not normalize some of the event properties according to W3C standards (`http://api.jquery.com/category/events/event-object/`).

The following diagram (also provided in the code bundle) should clarify what the visible viewport is (that is, the browser window—the red box):



Whenever we scroll down, different parts of the document (blue) are shown through the browser window and hidden due to space constraints. The scroll height (green) is the part of the document currently not displayed.

Custom jQuery selectors

Suppose we have a page with some text written in, which also contains a few links to both internal pages (that is, pages on the same server) and external websites.

We are presented with different choices in terms of which elements to apply the tooltip to (referring to links as an example, but they apply to any kind of element as well), as follows:

- All the links
- All the links with a specific class (for example, `tooltip`)
- All the links with the title attribute not empty
- All the links pointing to internal pages
- All the links pointing to external websites
- Combinations of the above

We can easily combine the first three conditions with the others (and with themselves) using CSS selectors appropriately. For example:

- `$("a")`, all the links
- `$("a.tooltip")`, links having a `tooltip` class
- `$("a[title]")`, links with a `title` attribute (still have to check if empty)
- `$("a.tooltip[title]")`, links with a `tooltip` class and a `title` attribute

As for internal and external pages, we have to work with jQuery selectors instead.

Time for action – creating custom jQuery selectors

Although jQuery makes it easy to select elements using standard CSS selectors, as well as some other selectors, jQuery's own selectors are the ones that help the developer to write and read code. Examples of custom selectors are `:odd`, `:animated`, and so on.

jQuery also lets you create your own selectors!

1. The syntax is as follows:

```
// definition
$.expr[':'].customselector = function(object, index, properties,
list) {
  // code goes here
};
// call
$("a:customselector")
```

2. The parameters are all optional except for the first one (of course!), which is required to perform some basic stuff on the selected object:
   - `object`: Reference to current HTML DOM element (not jQuery, beware!)
   - `index`: Zero-based loop index within array
   - `properties`: Array of metadata about the selector (the 4th argument contains the string passed to the jQuery selector)
   - `list`: Array of DOM elements to loop through

3. The return value can be either:
   - `true`: Include current element
   - `false`: Exclude current element

4. Our selector (for external links detection) will then look, very simply, like the following code:

```
$.expr[':'].external = function(object) {
  if(object.hostname) // is defined
    return(object.hostname != location.hostname);
  else return false;
};
```

5. Also note that, to access the jQuery object, we have to use the following (since object refers to the DOM element only!):

```
$.expr[':'].sample = function(object) {
  alert('$(obj).attr(): ' + $(object).attr("href") + 'obj.href: '
+ object.href);
};
```

Merging pieces together

We have slowly created different parts of the plugin, which we need to merge in order to create a working piece of code that actually makes tooltips visible.

So far we have understood how positioning works and how we can easily place an element in a determined position.

Also, we have found out we can create our own jQuery selectors, and have developed a simple yet useful custom selector with which we are able to select links pointing to either internal or external pages. It needs to be placed at the top of the code, inside the closure, as we will make use of the dollar symbol ($) and it may conflict with other software.

Time for action – creating a tooltip plugin

We still need to think of a way to actually make the tooltip show up and disappear, which is no big deal after all. Let's see how.

1. You should have created the directory containing the right files by now; but anyway, call the plugin file `jquery.tooltip.js` and let's move on.

2. After copying and pasting the code from the various files we used before, we should end up with something like this:

```
(function($) {
  $.expr[':'].external = function(object) {
    return(object.hostname != location.hostname);
  };

  $.fn.tooltip = function() {
    return this.each(function() {
      $(this).hover(function(event) {
        // mouse hover
      }, function() {
        // mouse leaves
      }).mousemove(function(event) {
        // mouse moves
      });
    });
  };
})(jQuery)
```

3. We have to take care of a few things now:

- Check if the element is actually a link (who knows!)
- Check if the title attribute is not empty
- Make sure that the default tooltip is not displayed
- Apply CSS styling to the tooltip

4. We can then add the following code at the very start of the `each` loop. The event binding will, therefore, be conditioned to the fact that the selected element is an element with a `title` attribute.

Also, we will remove the `title` content so that the default tooltip will not show up (note we first save the title content so that we can use its text later on!).

```
var e = $(this);
var title = e.attr("title");

if(title != '') {
  this.title = '';

  // mouse hover, move and out events
}
```

5. Once the mouse pointer hovers, we need to create the `tooltip` element, insert the text, hide it in order to make it fade, and then apply the necessary CSS to it (update its position).

```
$('<div id="tooltip" />').appendTo("body")
                   .text(title)
                   .hide()
                   .updatePosition(event)
                   .fadeIn(400);
```

6. Eventually, we might want to create another short function to take care of the update position stuff—since we have to repeat it a couple of times and we have much more control in terms of what we need it to do with less code.

```
$.fn.updatePosition = function(event) {
  return this.each(function() {

    $(this).css({
      // we modify the distances by some pixels to make sure
      // the tooltip is visible and put in the best place.
      left: event.pageX+20,
      top:  event.pageY-20
    });
  });
};
```

7. As for the events related to the mouse moving out, around, and on the element, we have the following two lines to do the job very smoothly:

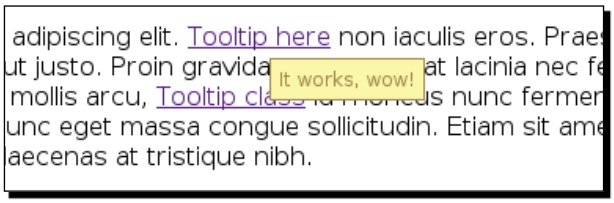
```
// mouse leaves
$("#tooltip").remove();

// mouse moves
$("#tooltip").updatePosition(event);
```

8. Last, but not least, we need to style the tooltip box to make it a little better looking and provide absolute positioning.

   Include the following code into a CSS file named after the plugin (that is, `jquery.tooltip.css`) and make sure that you link it to the page.

```
#tooltip {
  background: #fbf7aa;
  color: #a2844a;
  position: absolute;
  max-width: 150px;
  padding: 5px;
  border: 1px solid #a2844a;
  font-size: 12px;
}
```

9. Our tooltip is finally ready for using, and calling it as we described earlier certainly will work wonders!



10. And here is the complete code for our plugin. It has been broken down into smaller pieces and functions to make for better looking and more understandable code:

```
(function($) {
  $.expr[':'].external = function(object) {
    return(object.hostname != location.hostname);
  };

  $.fn.updatePosition = function(event) {
    return this.each(function() {
      var xy = getPageXY(event);

      $(this).css({
        left: event.pageX+20,
        top:  event.pageY-20
      });
    });
  };

  $.fn.tooltip = function() {
    return this.each(function() {
      var e = $(this);
      var title = e.attr("title");

      if(e.is("a") && title != '') {
         e.removeAttr('title')
    .hover(function(event) {
          // mouse hovers
          $('<div id="tooltip" />').appendTo("body")
                                   .text(title)
                                    .hide()
                                   .updatePosition(event)
                                   .fadeIn(400);
      }, function() {
```

```
            // mouse leaves
            $("#tooltip").remove();
          }).mousemove(function(event) {
            // mouse moves
            $("#tooltip").updatePosition(event);
          });
        }
      });
  };
})(jQuery)
```

What just happened?

The whole process for getting a (finally) working tooltip plugin should be clear by now. Anyway, here's a quick checklist of what we have done and what needs to be done in order to develop a jQuery plugin similar to ours.

- Disable the default tooltip display for elements that match our selection and have a `title` attribute (setting it to an empty string is enough for the tooltip not to appear, but we have to remember to copy the said string to a variable so we can use the text contained afterwards).
- When the mouse pointer hovers for the first time on a link, a new tooltip division is created. The tooltip box will be appended to the body and will be given absolute positioning. So we can set its distance from the top and left edges and make it appear close to the mouse cursor.
- As the mouse pointer moves on the link, the tooltip should follow. We thus need to update its `top` and `left` values according to the mouse cursor location.
- Eventually, the mouse will move away from the link and the tooltip should disappear. The whole process will repeat over and over for each link in the page.

Also, a couple more points to discuss are directly related to this script's cross-browser support and further customization of the tooltip.

As for the first, even though we have actually never taken browsers into account until now, we should, of course, do whatever possible to get our plugin working on many different browsers.

This is to our advantage, after all. If the plugin isn't working smoothly in any situation, our web page will not look good, behave right, or do what we thought it would.

Step by step, plugin after plugin, we are slowly adding more and more concepts and code into our plugins. This makes for a more advanced approach to plugin development, taking care of even little imperfections and flaws that our software might have on different browsers.

As for what customization is concerned, not much can be said, actually. The tooltip box is contained in a DIV element and, as such, all the normal CSS rules apply.

With little effort, we can thus obtain a more complex and appealing interface for our plugin—provided we have some skills at CSS writing. Images can be added too, to give the tooltip some shadow or other effects that CSS-only solutions will not allow.

Have a go hero – add options

Finally, the user should be given the opportunity to customize the tooltip plugin according to their needs and liking. So an option-less structure (as it is now) isn't actually the best choice in terms of usability.

It's true that users can include their own CSS file and overwrite every single instruction. However, some tasks, such as making rounded corners, changing the background, and selecting the font color, are more suitable for immediate application directly operating on the code (and thus the function call) rather than going all the way for a separate stylesheet.

Add options to give the user the possibility to change the above-mentioned parameters in a quick and easy way. You should also parameterize the offsets from the cursor and can add, along with rounded corners, the animation used when the tooltip is shown.

Pop quiz

1. What are the tasks we have to carry out, and in what order, to successfully develop a working tooltip plugin for jQuery?
    A. Retrieve the mouse pointer's position, disable the default tooltip, create the tooltip and show it, and remove the tooltip object.
    B. Retrieve the mouse pointer's position, create the tooltip and show it, update the position as needed, and hide the tooltip object.
    C. Retrieve the mouse pointer's position, disable the default tooltip, create the tooltip and show it, update the position as needed, and remove the tooltip object.
    D. Retrieve the mouse pointer's position, create the tooltip and show it, hide the tooltip object, and update the position as needed.

2. Theoretically, a tooltip should disappear once the mouse cursor pointer moves off the element that we chose as the tooltip target.

   How can we recognize that, in fact, the mouse cursor has been moved away from the element and the tooltip should be removed or deleted?

   A. We can't. That's why we wait until another element, which has been assigned a tooltip, is hovered upon so that the previous tooltip disappears and the new one is shown instead.

   B. Adding another (even anonymous) function as the second argument of the `hover` event tells jQuery to run the contained code once the mouse moves out off the selected element.

   C. Using the `mouseoff` event, which jQuery provides by default, and is triggered whenever the mouse pointer, after having been over the selected element, is moved away from it.

   D. None of the above.

3. Which of the following code portions actually does what it's intended for?

   A. Select external links.

   ```
   $.expr[':'].sample1 = function () {
             return (object.hostname !=!= location.hostname);
         };
   ```

   B. Select external links.

   ```
   $.expr[':'].sample2 = function () {
             return (object.hostname == location.hostname);
         };
   ```

   C. Select external links.

   ```
   $.expr[':'].sample3 = function (object) {
             return (object.hostname == location.hostname);
         };
   ```

   D. Select links with title attribute matching one of the given strings.

   ```
   $.expr[':'].sample4 = function (a, b, c, d) {
             var e = eval ("([" + c[3] + ")]");
             for (var i=0; i<e.length; i++)
              if (a.title == e[i]) return true;
            return false;
         };
   ```

4. What is a custom selector?

   A. In addition to the class and ID selectors, jQuery provides quite a lot of new selectors, but we are also allowed to create our own in case we don't find what we need.

B. Just a way to tidy up our code, with no real application except for those situations where we actually need a function-like approach to a problem that couldn't be solved in any other way due to the particular code structure jQuery is written in.

C. Creating a selector lets us access a set of functions and method that we would otherwise never have had a chance to make use of. These new methods allow a simpler and understandable code-writing style we should look forward to only when writing particularly complex code portions.

D. None of the above.

Summary

By now the structure lying behind a plugin is rather clear and the points that all of the plugins seem to share come out with no problem whatsoever. This leads to a quicker and more confident approach to plugin development.

Also, the development pattern we have followed until now has, of course, maintained a certain linearity and coherency. But we have gradually introduced different, more advanced, techniques and have learned a few new tricks that might come in handy at a later time, too.

During the creation of our tooltip plugin we have faced some problems, for which we have provided some flexible yet direct solutions, thanks to the modularity and object-oriented approach jQuery lets us make use of, as well as the incredible options available to overcome a particular hurdle in many different ways.

Our code is getting cleaner and cleaner as we move along, and different functions do different things.

Specifically, a function is responsible for retrieving the horizontal and vertical coordinates of the mouse cursor, whereas a method has been designed with the only purpose of updating the tooltip position based on the distances from the top and left edges.

Our plugin main code eventually had the sole goal of merging these features into a single piece of code (that is the plugin itself).

With the successful realization of a working tooltip plugin, our first attempt at getting a better understanding of what a user interface plugin actually is, and how they are supposed to work to increase the overall quality and visual appearance of the web page, has happily concluded.

The next chapter will instead deal with a different aspect of the user interface: menus. Often it is overlooked and not given the right amount of importance. The fundamental groupings of links that link all of the web pages together are indeed a hot target for many user interface tweaks. We can easily experiment on these with the invaluable help of the jQuery library.

10
User Interface Plugins: Menu and Navigation Plugins

No matter what kind of website we are going to build or what particular design we are looking forward to realizing, following a given inspiration of any kind, we will need a menu to allow visitors to navigate through our creation.

It's as simple as that, and we can't run away from it in any way. All of the pages that we have uploaded to our web server and are designed to be available to the guests to read need to be linked somehow.

Sure, we might use a series of links randomly put inside some text. However, the point here is to create a functional and easy-to-use interface that even the web bunglers will be happy (and able!) to make use of.

Since the early days of graphical user interfaces, menus (particularly drop-down menus for that matter) have been heavily used in both desktop applications (look around and notice how many drop-down menus are surrounding you in this very moment!) and in websites and applications, which have quickly followed suit in the drop-down menu trend. This has eventually led to an unjustified use of them in many cases.

The topics we're going to discuss include:

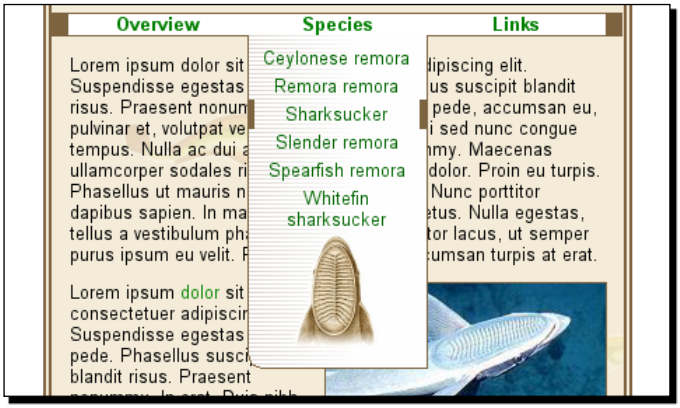
- Splitting the work in two
- CSS: drop-down menu and styling
- jQuery: spicing things up
- Creating the plugin

The purpose of this chapter is to demonstrate a simple, yet incredibly powerful and efficient way to create a jQuery plugin to make even plainer the process of making a drop-down menu starting from a list of links to different pages.

We can then go on to add effects (such as animations and sliding). We could also give the user the ability to modify colors and the overall look and feel of the menu that they will use the plugin on.

Splitting the work in two

To make things easier to sort out, we are going to work on two different pieces of code. We will then have to wrap them together to eventually obtain the desired plugin.



The first part is dedicated to the task of building a working CSS-only drop-down menu that works without JavaScript (even in Internet Explorer 6, possibly!), which requires a lot of extra markup and CSS code to begin with.

A lot of articles have been written on the topic of CSS drop-down menus, and several discussions arose concerning the most disparate issues in terms of cross-browser compatibility and best practices that developers and designers should follow to achieve high-level results**.**

A short, incomplete list follows. This might be useful to have a better grasp of what people are most interested in (when speaking of CSS-only drop-down menus) and what issues we'll be referring to at a later time:

- Suckerfish drop-down by Patrick Griffith on A List Apart: `http://www.alistapart.com/articles/dropdowns/`
- Create drop-down menus with CSS only by Janko Jovanovic: (`http://www.jankoatwarpspeed.com/post/2009/06/20/Create-dropdown-menus-with-CSS-only.aspx`

- How to make a simple CSS drop-down menu by Grace Teng on Evolt: `http://www.evolt.org/node/52030`
- Horizontal and Vertical CSS Menu Tutorial by Claire from Tanfa on SEO consultants: `http://www.seoconsultants.com/css/menus/tutorial/`
- The ULTIMATE CSS only drop-down menu by Stu Nicholls on CSSplay: `http://www.cssplay.co.uk/menus/final_drop.html`

As far as the second and last part of our development pattern is concerned, a slight effort is now required in order to understand and put in into practice the power that jQuery gives us on this occasion.

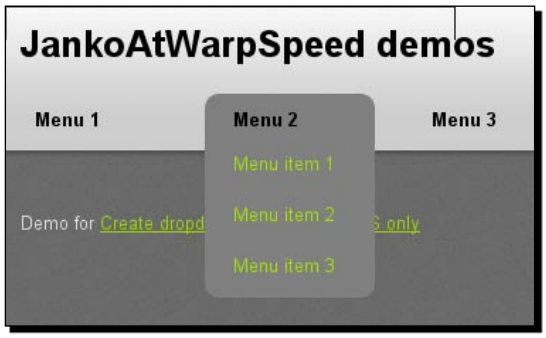
Provided we have a working CSS-only drop-down menu at hand, what we need to accomplish with jQuery is, in the first instance at least, adding a bit of flavor to what we have just developed. We might want to add animation effects, or even other more advanced techniques to get the best out of our menu implementation.

One thing must call our attention, now: we're still working on a precise, defined element—one, and only one element—doesn't sound like a plugin, does it?

In our last part, our objective is to make what we have obtained until that point applicable to any list element whatsoever, and effectively develop a jQuery plugin with all its related bells and whistles.

CSS: Drop-down menu and styling

Before starting to write code and thinking of how to get over some of the problems we'll stumble upon, we might want to have a look at how a CSS menu (or even a full working menu with jQuery effects) could look and behave when the mouse pointer hovers on any of its links.



As you may have noticed in the live applications of the previously mentioned menus, whenever we hover the mouse pointer over a top-level link, a new sub-menu pops up from nowhere. Also, if the link we hover on has no sub-menus (for whatever reason) nothing happens, and we are still able to click and follow the link on the top-level menu.

Time for action – creating and styling the menu

As our first step towards the end product (that is, for this section, a working drop-down menu), we have to define our basic menu hierarchy in order to eventually maintain a structured and semantic approach.

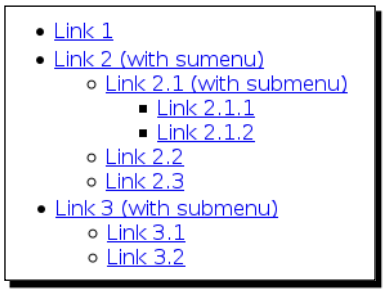
1. We'll use a simple, nested unordered list of standard links for this purpose. The top-level list (main menu) will be identified by the `menu` ID, whereas nested lists (sub-menus) will be defined as follows:

```
<ul id="menu">
  <li>
    <a href="#">Link 1</a>
    <!-- no sub menu -->
  </li>

  <li>
    <a href="#">Link 2 (with submenu)</a>
    <!-- submenu -->
    <ul>
      <li>
        <a href="#">Link 2.1 (with submenu)</a>
        <ul>
          <li>
            <a href="#">Link 2.1.1</a>
            <!-- no sub menu -->
          </li>

          <li>
            <a href="#">Link 2.1.2</a>
            <!-- no sub menu -->
          </li>
        </ul>
      </li>

      <li>
        <a href="#">Link 2.2</a>
        <!-- no sub menu -->
      </li>
```

```
        <li>
          <a href="#">Link 2.3</a>
          <!-- no sub menu -->
        </li>
      </ul>
    </li>

    <li>
      <a href="#">Link 3 (with submenu)</a>
      <!-- submenu -->
      <ul>
        <li>
          <a href="#">Link 3.1</a>
          <!-- no sub menu -->
        </li>

        <li>
          <a href="#">Link 3.2</a>
          <!-- no sub menu -->
        </li>
      </ul>
    </li>
</ul>
```

2. Even though this code listing looks awfully long, it's actually because the list hierarchy is more clearly visible this way.

As a first preview, here is what the previous code produces (with no CSS styles applied yet!):



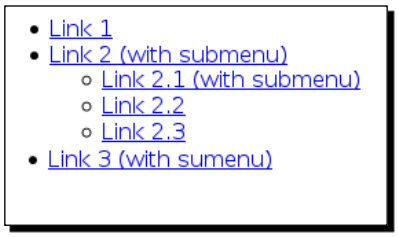
3. With our HTML and link structure written, we can now proceed to apply some basic hover techniques that are included in CSS.

It is best to take this opportunity to apply some basic styling to the list (the next step), such as floating the parent unordered lists in a row, and specifying the width of the unordered list to ensure that no shifting occurs when the elements are hidden or shown.

4. Firstly, we have to take care of hiding all of the sub-menus, which will be shown only when the mouse pointer hovers over their parent link.

```
#menu li ul {
  display: none;
}

#menu li:hover > ul  {
  display: block;
}
```

5. Now, hovering the mouse pointer over any of the links in the main menu list will result in its sub-menu element being shown. If we have another nested menu, its sub-menu will not be shown unless we hover the pointer over its parent link as well.

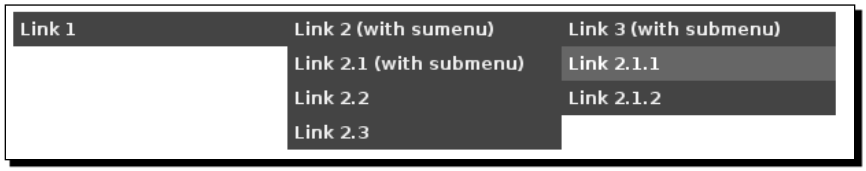


6. We then proceed to placing the menu in a horizontal fashion, and also specifying the minimum and maximum width allowed for menu items:

```
#menu, #menu ul {
  list-style-type: none;
  list-style-position: outside;
  margin: 0;
  padding: 0;
  position: relative;
}

#menu li {
  float: left;
```

```
  width: 200px;
  position: relative;
}

#menu li ul {
  display: none;
}

#menu li ul li ul {
  position: absolute;
  left: 200px;
  top: 0;
}

#menu li, #menu li a{
  background-color: #444;
  color: #eee;
  float: left;
  margin: 0;
}

#menu a {
  float: left;
  padding: 5px;
  width: 190px;
  text-decoration: none;
  font-weight: bold;
  font-size: 12px;
}

#menu a:hover {
  background-color: #666;
  color: #fff;
}
```

7. The CSS style will then help to make the menu work like the following. Of course, we still need to improve things using jQuery and also write a traditional plugin before we're done with it:



Have a go hero – overcome Internet Explorer problems

Unfortunately, the lovely `:hover` pseudo-class may have some problems with earlier versions of Internet Explorer; that is, it may not work properly.

To get over this problem and actually have a script working with older versions of Internet Explorer as well, we need to implement a JavaScript solution.

A possible and suggested solution is to make use of some of the methods that we have already analyzed and used previously. These methods provide a way to access the hover events of any element in the document.

With the help of such methods, we can then apply, through jQuery, the CSS code we need to make the whole thing work on other browsers as well.

Write the necessary code to make sure the script works in browsers that do not support the `:hover` pseudo-class.

jQuery: Spicing things up

OK, we now have a working CSS-only drop-down menu.

We also have jQuery at hands' reach. Not using it would only result in avoiding a lot of fun-to-write code and not seeing how things interact with each other.

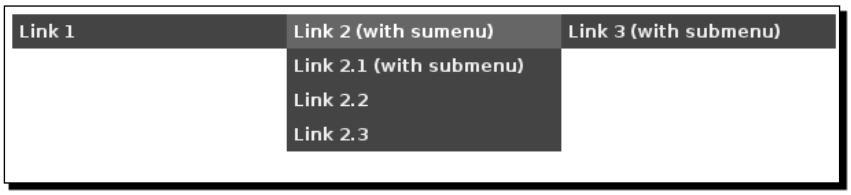
Time for action – adding a fading effect

Here we go again. Using our beloved functions will help us deal with mouse hover and mouse out events. Eventually, we will add some tweaks to our script and make everything look and behave in a cuter fashion.

1. Write the following code into the "document ready" statement, so that our menu elements will eventually fade in (on mouse hover) and disappear when the mouse cursor is moved away from the link.

```
$(document).ready(function() {
  $("#menu li").hover(function() {
    $("ul:first", this).css({ visibility: "visible"
     }).fadeIn(600);
  },function() {
    $("ul:first", this).hide();
  });
});
```

2. Hovering over any link, which has a sub-menu attached, will now lead to a fade effect, which will doubtlessly improve the overall look and feel of the web page.



Have a go hero – try out other effects

Of course, jQuery is not limited to fading effects only.

Try experimenting with different effects and animations (such as sliding techniques or menus popping out), which can all be applied to our new-born menu element with a couple of lines of code.

Do not worry about the outcome for now. We'll have a closer look at the actual power of animations and similar effects later in the book.

Creating the plugin

Just one last thing is left to obtain for our long-awaited plugin: the plugin structure itself!

Until now what we have created is a very cool script for animating a drop-down menu. However, our first goal was to have a plugin to be used in different situations to turn a standard unordered list of elements into a drop-down menu.

We're almost there, anyway.

Time for action – creating the plugin

As our last step, we have to merge all the pieces together to eventually obtain a plugin, which does exactly what we were after.

1. As usual, we need a directory named `menu` and the other basic necessary files as well. A CSS document (possibly named `jquery.menu.css`) is needed, too!

2. We'll need the CSS file to be included along with the menu plugin, as it will be part of the plugin itself, providing the basic style and overall look to the menu.

   The only change we need to make to the CSS code that we've already written is replacing all of the `#menu` occurrences (referring to an element ID) with `.dropdownMenu` (thus referring to an element class). The reason for this is that, in case we have more than one menu on the same page, we want them to be able to operate individually.

   Also, this means that, in order to make the plugin work, the user is **not** required to assign the `dropdownMenu` class to any list element that they need to be turned into a drop-down menu. This will be taken care of in our plugin as well.

3. Here is the simple plugin code that makes everything work smoothly:

```
(function($) {
  jQuery.fn.menu = function() {
    return this.each(function() {
      $(this).addClass("dropdownMenu");

      $("li", this).hover(function() {
        $("ul:first", this).css({ visibility: "visible" })
         .fadeIn (600);
        },function() {
        $("ul:first", this).hide();
      });
    });
  };
})(jQuery)
```

4. Now, having a standard list of links, we may use the following to turn it to something more appealing to the average user:

```
$(document).ready(function() {
  $("#menu").menu();
});
```

What just happened?

Even though the whole process is rather straightforward, there's always a little detail that could be overlooked and, thus, left behind.

In order to have a clear understanding of what's been done from the very beginning (HTML code to create a standard list) to the end (jQuery plugin), we may use the following points to better remember some of the so-called checkpoints:

- **Create a simple HTML unordered list**

  This is nothing but one of the basic requirements we're going to work on and with. In fact, this list (that will be our menu) is organized with a series of links representing the menu elements, and a handful of nested lists to represent our sub-menu structure.

- **Style the list into a menu-shaped element**

  Basically, we are looking ahead to obtain a horizontal drop-down menu, and, with this in mind, we should aim at creating something that at least looks like one. We then need sub-menus and second-level links to be hidden by default, but suddenly visible once the user hovers the pointer over the parent elements nonetheless.

  Also, once the mouse pointer moves off the link or clicks on it, everything should revert to normal: only top level links are shown.

- **Add jQuery effects**

  jQuery, once again, may be of great help in case we want something more than just a simple, plain, CSS-only drop-down menu.

  Still working fine on browsers that do not support JavaScript, our menu can have that nice flavor thanks to the bundle of effects the jQuery library provides along with the basic package.

- **Create the plugin**

  As silly as it sounds, we are not done yet. The plugin is still undone! To put already shamefully simple things down in an even simpler way, we are to make our script work on multiple lists at the same time, leaving out all the predetermined portions of the code.

  That said, we are replacing all the IDs and classes referring to a particular, specified element (or list) with references to elements belonging to the list we are running the plugin on.

Have a go hero – allow users to customize the plugin

We have got a nice, working plugin, sure, but what about user customization?

Create an options structure to allow users to specify some details (such as colors, menu width, and so on) without the need to edit any other file.

Changes will be made at run time. This means that the default stylesheet provides the basic look and feel of the menu, whereas the options passed by the user may change some particular aspects of the menu.

Pop quiz

1. After having read the possible way to obtain a CSS + jQuery drop-down menu, which of the following is the best path to follow in order to get a working plugin?

   A. The order in which we create the menu, CSS code, and jQuery functions does not matter as long as the plugin does not modify in any way the original CSS code.

   B. Creating the menu plugin is the last task, which is to be taken care of only once the CSS code has been implemented into the JavaScript portion of the script.

   C. Create the plugin after having put together the jQuery effects and the basic CSS styling that will be applied to the selected menu element, consisting of nothing more than a simple HTML element containing any number of nested unordered lists.

   D. None of the above.

2 Our plugin is known to have some issues when dealing with the Microsoft Internet Explorer browser.

Of what nature is this trouble and how can the problem be (possibly) solved?

   A. The problem lies in the impossibility, in certain circumstances, of Internet Explorer recognizing the `:hover` pseudo-class, forcing us to avoid its use if we aim at realizing a cross-browser solution.

   B. The problem lies in the impossibility, in certain circumstances, of Internet Explorer to recognize the `:hover` pseudo-class, forcing us to use JavaScript to make the script cross-browser.

   C. The problem lies in the impossibility of Internet Explorer to recognize the hover event, forcing us to use the CSS `:hover` pseudo-class to get things done.

   D. None of the above.

3 From a compatibility point of view, on what kinds of browser can our plugin work correctly (and why)?

   A. Our plugin code is intended to work on all browsers, since its code makes the menu work even without the aid of jQuery, which is responsible for the animation part only; it is true, though, that the user will have to manually add the right class name to their menu of choice, since, with no JavaScript support, jQuery won't load the plugin code, too.

   B. Our plugin code is intended to work on all browsers, since its code makes the menu work even without the aid of jQuery, which is responsible for the animation part only.

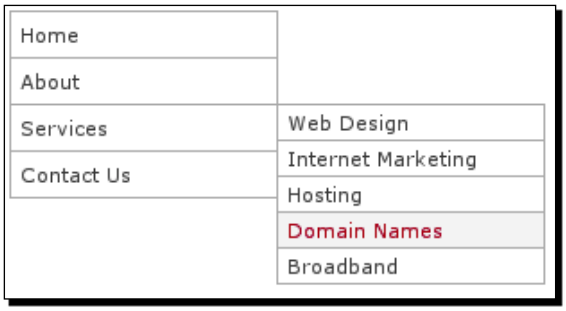
C. All browsers that support JavaScript, since our plugin heavily relies on this technology to make the menu work, which it wouldn't if there was no JavaScript code.

D. None of the above.

Summary

This chapter has guided us through the creation of a simple yet (almost) nice to look at and versatile plugin to make the realization of drop-down menus relatively easy and accessible.

We have dealt with a horizontal menu only, but this does not mean that no other menu types are available for use and inspiration. In fact, just as horizontal drop-down menus exist, so do vertical menus (which, of course won't "drop down").



The above image, taken from the result of an article at A List Apart (`http://www.alistapart.com/articles/horizdropdowns/`), for example, is a sample of what can be done applying some of the techniques explained earlier, but thinking of the menu vertically rather than horizontally, using only a few lines of CSS code to make it work exactly how a vertical menu is supposed to work.

Also, in case animations and the other various effects have left you wondering how is this possible, well, the very next chapter comes with great timing, since it will discuss in detail some of the ways such effects and other advanced animation techniques are obtained and can be included into our plugins with little effort.

11
Animation Plugins

As time went by, we've all noticed a significant increase in the number of websites that have turned to animation effects in order to make pages look (and feel) better, easier to interact with, and definitely fun to use.

People are drawn to this kind of enhancement, and they often don't resist the urge to click on something that moves smoothly or bounces across the edges of their browser window.

Effects such as these that we're talking about can be achieved in many different ways. One thing is for sure, jQuery can handle all of these animations, and we're actually going to have a closer look at what makes the "magic" happen and, most importantly, how it happens in terms of code and page layout.

Also, it is important to understand the different types of animation that can be achieved through the use of jQuery. These animations will certainly find different applications depending on their function.

The following topics will be covered in this chapter:

- Sliding
- Fading
- The `animate()` method

Sliding

Much like fading, sliding is a simple trick that, as the name suggests, makes something move smoothly; for example, a panel sliding from one edge of the page to the other.

Though fading techniques have vast usage in a number of different applications, I happen to like sliding better and see it as the "right" solution for many enhancements out there.

What does "sliding" actually mean?

jQuery provides the following three methods for applying sliding motions to elements on a web page:

- `.slideUp()`
- `.slideDown()`
- `.slideToggle()`

Though guessing what `.slideUp()` and `.slideDown()` do is not that hard, `slideToggle` is worth some words (from the jQuery documentation pages on the topic of `.slideToggle()` method):

> *The* `.slideToggle()` *method animates the height of the matched elements. This causes lower parts of the page to slide up or down, appearing to reveal or conceal the items. If the element is initially displayed, it will be hidden; if hidden, it will be shown. The* `display` *property is saved and restored as needed. If an element has a* `display` *value of* `inline`*, then is hidden and shown, it will once again be displayed* `inline`*. When the height reaches 0 after a hiding animation, the* `display` *style property is set to* `none` *to ensure that the element no longer affects the layout of the page.*

They are all called in the same way and do not need any arguments (that is, they are optional):

```
.slideUp([duration], [callback])
.slideDown([duration], [callback])
.slideToggle([duration], [callback])
```

Where:

- `duration`: Is a string or number determining how long the animation will run.
- `callback`: Is a function to be called once the animation ends.

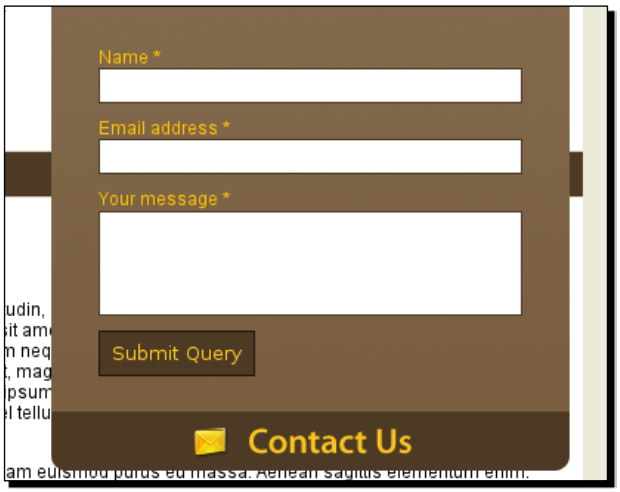
Again, duration can be a number (in milliseconds) or either of the strings "slow" and "fast". The default duration (if no argument is passed or a string other than "slow" or "fast" is passed) is 400 milliseconds.

Sample plugins that "slide"

The following are some examples on how to use these sliding techniques at their best and eventually implement a more complex plugin that makes use of both slide and fade effects:

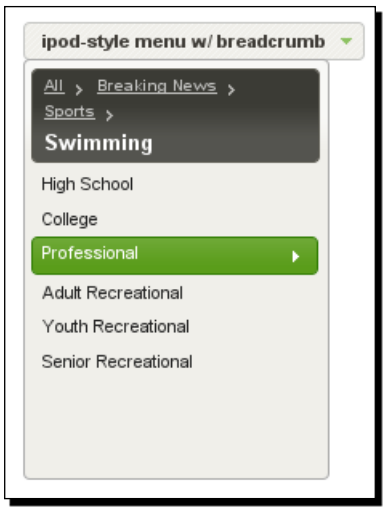
- **Slide-in contact form** by *Janko Jovanovic* – with step-by-step instructions!

  `http://designshack.co.uk/tutorials/creating-a-slide-in-jquery-contact-form`



- iPod-style drilldown menu by *Maggie Costello Wachs*

  `http://www.filamentgroup.com/lab/jquery_ipod_style_drilldown_menu`



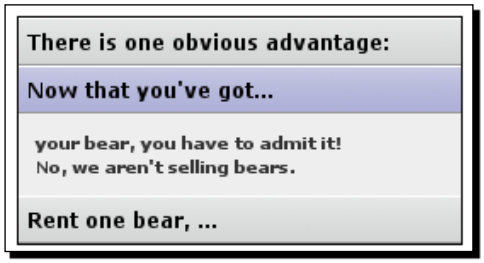
- CrossSlide by Tobia Conforto

  `http://tobia.github.com/CrossSlide`

Creating an accordion plugin (that slides!)

If you don't know what an accordion plugin is, just think about those fancy things that, basically, are a series of panes. When one pane slides down, the other pane slides up, leaving only one pane (the selected one) visible all the time.

Here is an example of what we'll end up with. Clicking on one of the headers will cause the related content to be displayed, and all other boxes to collapse with a scrolling motion (the image is from `http://bassistance.de/jquery-plugins/jquery-plugin-accordion/`):



Time for action – creating sliding panes

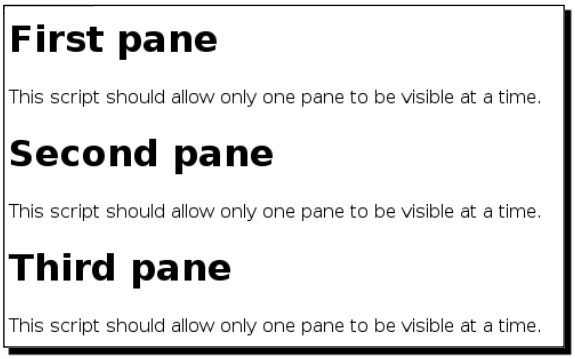
Our goal is to end up with a plugin that behaves in exactly the same way as the preceding example. There's nothing excessively complicated but, as usual, we will get to know some handy functions that might be useful in a number of different situations.

1. Create a new directory called `accordion`, and copy over the needed files. Then create a new JavaScript file for our plugin (`jquery.accordion.js`).

2. We first need to set up the structure of our accordion. It's very simple and straightforward, as we will make use of divisions. Each division represents a pane, and each pane will necessarily consist of at least two elements: a header and a paragraph.

```
<div id="pane-container">
  <div class="pane">
    <h1>First pane</h1>
    <p>This script should allow only one pane to be visible at a
       time.</p>
  </div>

  <div class="pane">
    <h1>Second pane</h1>
    <p>This script should allow only one pane to be visible at a
       time.</p>
  </div>

  <div class="pane">
    <h1>Third pane</h1>
    <p>This script should allow only one pane to be visible at a
       time.</p>
  </div>
</div>
```

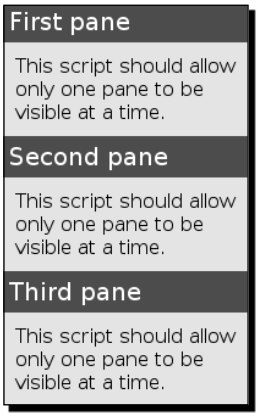
3. The header part will always be visible, and users will be able to see the panes by clicking on the corresponding header.



4. Create another file called `style.css`. We'll use this file to style our page and make it look a little appealing. Enter the following code:

```
#pane-container {
  margin: 0;
  padding: 0;
  width: 200px;
}

.pane h1 {
  padding: 5px;
  cursor: pointer;
  position: relative;
  background-color: #4c4c4c;
  color: #fff;
  font-weight: normal;
  font-size: 20px;
  margin: 0;
}

.pane p {
  padding: 10px;
  margin: 0;
  background-color: #e4e4e4;
}
```

This should result in a result similar to the following screenshot:



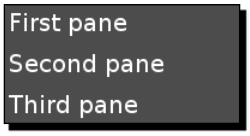
5. Now, our first goal is to hide all the parts that aren't supposed to be shown in the first place (that is, paragraphs).

Remember to call the plugin function on the pane container:

```
$("#pane-container").accordion();
```

Our basic plugin structure will then be filled as follows:

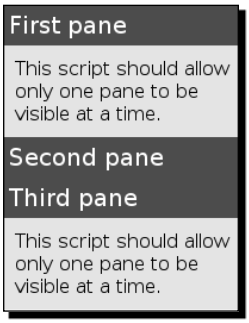
```
(function($) {
  $.fn.accordion = function() {
    return this.each(function() {
      $(this).find("p").hide();
    });
  };
})(jQuery)
```



6. Next, when the user clicks on a heading, the corresponding pane should slide down (or up if it is already visible).

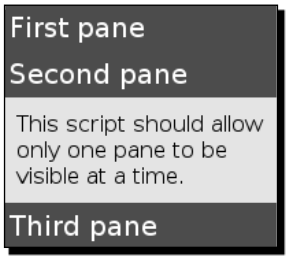
```
$(this).find("h1").click(function() {
    $(this).next("p").slideToggle(700);
});
```

7. It should be noticed, though, that panes won't slide back up when a different heading is pressed.



8. To fix this behavior, the following code determines what the panes should look like:

```
(function($) {
  jQuery.fn.accordion = function() {
    return this.each (function() {
      var e = $(this);

      e.find("p").hide();
      e.find("h1").click(function() {
        $(this).next("p").slideToggle(700).parent().siblings()
         .children("p").slideUp("slow");
      });
    });
  };
})(jQuery)
```

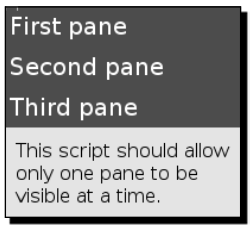


9. So far so good, but we can also add a customization option to let the user select the pane to be open by default (if the users want any pane open by default, of course).

10. Using the `:eq` (or :nth-child, which is 1-indexed) built-in selector, we can easily develop this feature, making the plugin determine which pane to show on page load.

```
(function($) {
  $.fn.accordion = function(options) {
    var defaults = {
      visibleByDefault: 2
    };

    var o = $.extend(defaults, options);

    return this.each(function() {
      var e = $(this);

      e.find("p").hide();
```

```
        e.children(":eq("+o.visibleByDefault+")").children("p")
         .show();
        e.find("h1").click(function() {
          $(this).next("p").slideToggle(700).parent().siblings()
           .children("p").slideUp("slow");
        });
      });
    };
})(jQuery)
```



Note that `:eq` is zero based. So the first element will correspond to 0.

What just happened?

The overall functioning of the plugin should be easy to understand. Some hints on how the functions have been put together would certainly help.

First off, `find()` and `children()`: our plugin is strongly based on them, and they each do very similar things.

However, though the `find()` method "gets the descendants of each element in the current set of matched elements", `children()` "gets the children of each element in the set of matched elements".

The difference might sound silly, but with `children()` we only get the first-level descendants (children, exactly!), whereas the `find()` method goes down many levels.

When the structure of the HTML is predetermined, it usually does not make sense to use `find()`, as it could result in poor performance. The performance hit might be ignorable, but it still makes the code less desirable.

If we had to use `children()` only, the portion of code that retrieves the paragraphs would read (according to the fact that we are allowed to go through one level at a time):

```
$(this).children ().children ("p")
```

As for `siblings()` and `parent()`, they respectively return what their names suggest; that is, siblings (all nodes at the same level) and parent elements of the selected element(s).

Note that `parent()` only goes up one level (much like `children()`), but you can use `parents()` to get a set of elements corresponding to all parent elements.

Lastly, as slide functions have already been dealt with, we're going to spend some words on the speed at which the animation completes.

This brief thought also applies to any other method that allows users to enter a custom "duration" value.

Note that the higher the number you supply, the longer the animation will take. Conversely, the lower the number you supply, the quicker the animation—this is not the speed, but how long the animation takes to complete (that is, the duration)!

Have a go hero – make changes

This simple plugin also lies open to different applications. Its simple design can be easily modified to work as a menu, which is not an unusual end product for this type of script.

Also, the event triggering the display of the panes can be changed to something more suitable to the situation.

Create another plugin, very similar to this one, which shows the panes whenever the mouse pointer hovers over the corresponding header. The pane should then slide back up once the mouse pointer leaves the area.

Also try to create another version of this plugin in which a default pane opens (siding down) when the mouse pointer does not hover over any other element of the selection.

Fading

Actually, there's little to say about these techniques as we've already used some of them in our previous plugins. However, a fresh explanation with some more details is always useful.

What does "fading" actually mean?

There are three fading methods that we can use in any jQuery code we write:

- `fadeIn()`
- `fadeOut()`
- `fadeTo()`

Any fade method is always a change in the opacity of the element. If the `fadeIn()` method progressively increases the opacity of the element until it is fully opaque, the `fadeOut()` method does the opposite—thus, starting from an opaque element, making it invisible (hidden).

The `fadeTo()` method can be used to fade an element to a certain value of opacity, in a scale from 0 (invisible) to 1 (fully opaque).

Fading methods are called as follows:

```
.fadeIn(([duration], [callback])
.fadeOut([duration], [callback])
.fadeTo([duration], [opacity], [callback])
```

Where:

- `duration`: Is a string or number determining how long the animation will take. The strings "fast" and "slow" can be used instead of 200 and 600 ms respectively.
- `opacity` (`fadeTo()` only): Is a number between 0 and 1 determining the target opacity.
- `callback` (optional): Is a function to call once the animation is complete.

Sample plugins that "fade"

Fading effects are used in a variety of plugins, often to enhance user experience and give the plugin an improved look.

However, here are some of the most interesting uses of fading effects and a quick explanation of how to obtain similar results:

- InnerFade by Torsten Baldes

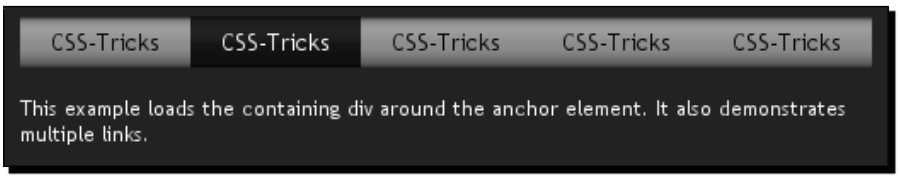
  `http://medienfreunde.com/lab/innerfade/`

- jGrowl by Stan Lemon

  `http://stanlemon.net/projects/jgrowl.html`

- **Color Fading Menu** by *Liam Goodacre*

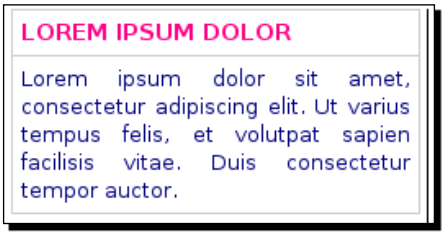
  `http://css-tricks.com/color-fading-menu-with-jquery/`



Creating a fading news ticker plugin

Our last interaction with fading effects is going to let us create a fade-related plugin and obtain some great results that can be applied to other plugins as well.

Though the fading effect cannot be reproduced in print, the following screenshot represents the final result, in terms of look and feel, of the news ticker plugin we're about to realize:



Time for action – creating the plugin

The news ticker plugin, which we will obtain by the end of this section, is a clear example of how the same (or similar) effects can be applied to different kind of plugins to enhance their appearance or usability.

1. Create a new directory, name it `news-ticker` (not new-sticker!), and copy over the necessary files.

2. Our plugin file, `jquery.ticker.js`, will obviously contain the code that we will need to create the animation. In the meantime, we can enter the basic HTML structure of a sample page into our file `index.html`. Each news will consist of a header and a body, which are both wrapped into a `news` division.

```
<div id="news-container">
  <div class="news">
    <h1>Lorem ipsum dolor</h1>
    <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
       Ut varius tempus felis, et volutpat sapien facilisis vitae.
       Duis consectetur tempor auctor.</p>
  </div>

  <div class="news">
    <h1>Vivamus lobortis faucibus</h1>
    <p>Vivamus lobortis faucibus dolor eget imperdiet. Maecenas
       gravida aliquet quam in congue. Ut lobortis est a quam
       convallis tempus. Maecenas nunc neque, ullamcorper at
       ultricies sed, malesuada in sem. Morbi at nulla in nisi
       imperdiet tempus.</p>
  </div>

  <div class="news">
    <h1>Morbi sagittis</h1>
    <p>Morbi sagittis tincidunt diam, id luctus velit fermentum
       semper. Morbi faucibus, diam a sollicitudin consectetur,
       tellus neque molestie lacus, dictum tempor justo neque id
       erat. Maecenas ut nunc at magna convallis dapibus.
       Suspendisse neque mauris, elementum at congue non,
       elementum nec ante. Nunc nisi leo, placerat eu lobortis
       sed, dictum non nisi.</p>
  </div>

  <div class="news">
    <h1>Praesent bibendum lorem</h1>
```

```
        <p>Praesent bibendum lorem at nulla vestibulum semper. Nunc
           eget tristique erat. Morbi ut nibh id eros auctor blandit
           sed eget nisi. Ut vel felis magna. Nunc odio augue, porta
           a vulputate sit amet, pellentesque in mi. Mauris luctus
           urna eget velit semper at tempus nisi volutpat.</p>
    </div>
</div>
```

3. The checklist of things to do, in order to make the plugin work as we expect it to, can be summed up as follows:

- Hide all news items except the first one, which is visible by default.
- The news item remains visible for a given amount of time, after which it fades out.
- At the same time, the next news item should appear with some fading effect.
- The cycle continues until the last news item has been displayed, and then the whole process starts over from the first item.

LOREM IPSUM DOLOR

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut varius tempus felis, et volutpat sapien facilisis vitae. Duis consectetur tempor auctor.

VIVAMUS LOBORTIS FAUCIBUS

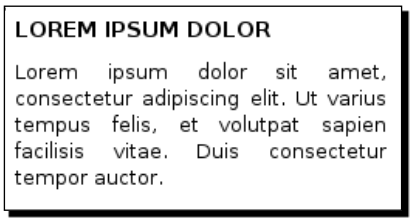
Vivamus lobortis faucibus dolor eget imperdiet. Maecenas gravida aliquet quam in congue. Ut lobortis est a quam convallis tempus. Maecenas nunc neque, ullamcorper at ultricies sed, malesuada in sem. Morbi at nulla in nisi imperdiet tempus.

MORBI SAGITTIS

Morbi sagittis tincidunt diam, id luctus velit fermentum semper. Morbi faucibus, diam a sollicitudin consectetur, tellus neque molestie lacus, dictum tempor justo neque id erat. Maecenas ut nunc at magna convallis dapibus. Suspendisse neque mauris, elementum at congue non, elementum nec ante. Nunc nisi leo, placerat eu lobortis sed, dictum non nisi.

4. Using the jQuery built-in selectors, we can easily select all but the first child of a given object:

```
$(this).children(":not(:first)").hide();
```



5. Immediately after, using the `setInterval` function, we can call a function repeatedly, with a fixed time delay between each call:

```
setInterval(function() {
  // function content
}, 2000);
```

6. Now, the technique to cycle over child elements is really simple:

   - We select the first child of the news container (that is, the first news item visible by default for the user to read).
   - Applying the `fadeOut()` method on the selected element, it disappears.
   - We select the next sibling of the element using the `next()` method and make it fade in.
   - We then move the news item that has been hidden to the bottom of the queue—that is, after all of its siblings, at the bottom of the news container.
   - Repeating this procedure over and over (thanks to the `setInterval` function) makes the news ticker work.
   - And, best of all, we can chain all of this together!

```
(function($) {
  $.fn.ticker = function() {
    return this.each(function() {
      var e = $(this);

      e.children(':not(:first)').hide();
```

```
        setInterval(function() {
        e.children(":first").fadeOut().next().fadeIn().end()
         .appendTo(e);
        }, 2000);
      });
  };
})(jQuery)
```

What does end() end?

"end() ends the most recent filtering operation in the current chain and returns the set of matched elements to its previous state."

In fact, our code selects the first child (`eq(0)`), fades it out, and selects the next item to make it fade in. If we had chained the `appendTo()` method at this point, the item actually appended to the container would have been the just faded in news, instead of the one that disappeared!

But thanks to the `end()` function, we go back to our previous selection and append the right element.

7. Also note that a problem we have is an erroneous fade effect.

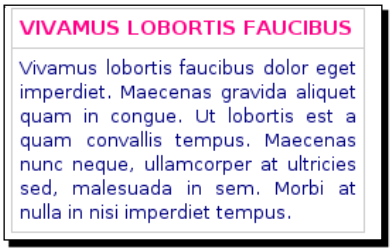
In the short period of time during which the two divisions fade out and in, it looks like they are, for a very brief moment, one right beneath the other and, once the top news item has completely faded out, the replacement element moves to its right place.

This is easily solved by using absolute positioning, which can be implemented through some CSS code (together with a basic style to make the whole thing look a little bit better):

```
#news-container {
  position: relative;
  width: 250px;
}

.news {
  position: absolute;
  left: 0;
  top: 0;
  border: 1px solid #ccc;
}

h1 {
  font-size: 14px;
```

```
  font-weight: bold;
  text-transform: uppercase;
  color: #ff0088;
  border-bottom: 1px solid #ccc;
  padding: 5px;
  margin: 0;
}

p {
  font-size: 13px;
  text-align: justify;
  color: navy;
  padding: 5px;
  margin: 0;
}
```

Here is our end result:



8. Of course, further improvements are possible. For example, we might want to let the user specify a custom time delay, or even start the animation from a news item other than the first one.

9. Using the standard options structure, any customization is as easy as writing a couple lines of code, while the `:first` selector lets us add some spice to our plugin.

```
(function($) {
  $.fn.ticker = function(options) {
    var defaults = {
      startWith: 0,
      showDelay: 2000
    };

    var o = jQuery.extend(defaults, options);

    return this.each(function() {
      var e = $(this);
```

```
            e.children(":lt("+(o.startWith)+")").appendTo(e);

            e.children(':not(:first)').hide();

            setInterval(function() {
              e.children(":first").fadeOut().next().fadeIn().end()
               .appendTo(e);
            }, o.showDelay);
          });
        };
      })(jQuery)
```

Have a go hero – add fading effects

Experiment with all the fade methods (`fadeIn()`, `fadeOut()`, `fadeTo()`) and create a simple plugin that, based on user input, can animate an object with either fade or slide functions.

The plugin will then need at least one parameter to let the user choose the preferred type of animation.

Also, an additional feature could make it possible for the user to choose opacity values to which the element will fade, if the right function has been selected.

The animate() method

The `animate` method is one of the most powerful animation functions we have a chance to use.

Given a set of CSS properties, this method performs a custom animation, transforming the selected (set of) element(s).

Understanding the jQuery animate() method

The jQuery `animate()` function is a very powerful way to manipulate HTML elements, adding any type of animation functionality to any kind of element on the page, as long as CSS properties can be applied to it.

The syntax of this method is as follows.

```
.animate(properties, [duration], [easing], [callback])
.animate(properties, options)
```

Specifically:

- `properties` is a map of CSS properties that the animation will move towards.
- `duration` is a string or number determining how long the animation will run.
- `easing` is a string indicating which easing function to use for the transition.
- `callback` is a function to call once the animation is complete.
- `options` is a map of additional options to pass to the method. The supported keys are:
  - `duration`
  - `easing`
  - `complete`: Same as callback
  - `step`: A function to be called after each step of the animation
  - `queue`: A Boolean indicating whether to place the animation in the effects queue
  - `specialEasing`: A map of one or more of the CSS properties defined by the `properties` argument and their corresponding `easing` functions

To better understand what easing functions are, we can read what the jQuery documentation reveals on the subject:

"An easing function specifies the speed at which the animation progresses at different points within the animation. The only easing implementations in the jQuery library are the default, called `swing`, and one that progresses at a constant pace, called `linear`. More easing functions are available with the use of plug-ins, most notably the jQuery UI suite (`http://jqueryui.com`)."

Time for action – creating your first animation

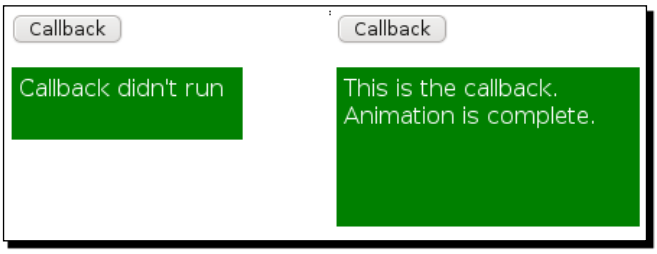
To test how animation works, we are going to create a series of simple scripts that will help us understand how this method works, and how its arguments can be used in order to obtain better results in our jQuery coding experience.

1. In an HTML file, paste the following code, which animates the `width` and `height` of a DIV:

```
<html>
<head>
  <script src="jquery.js"></script>
```

```
  <script src="jquery.accordion.js"></script>
  <script>
    $(document).ready (function () {
      $("#dimension-button").click(function() {
        $("#dimension-div").animate({ "height": "100px",
          "width": "200px" });
      });
    });
  </script>

  <style type="text/css">
    #dimension-div {
      background-color: green;
      color: white;
      width: 150px;
      height: 40px;
      padding: 5px;
    }
  </style>
</head>
<body>
  <button id="dimension-button">Click me!</button>
  <div id="dimension-div">Change dimensions</div>
</body>
</html>
```

2. Clicking on the button will make the green box grow in size; its height and width will increase at the same time.



3. In a similar fashion to combining animations, we can also queue them: once the first has finished, the other one will immediately be executed. This behavior can be avoided by setting the parameter `queue` to false when passing the map to the function.

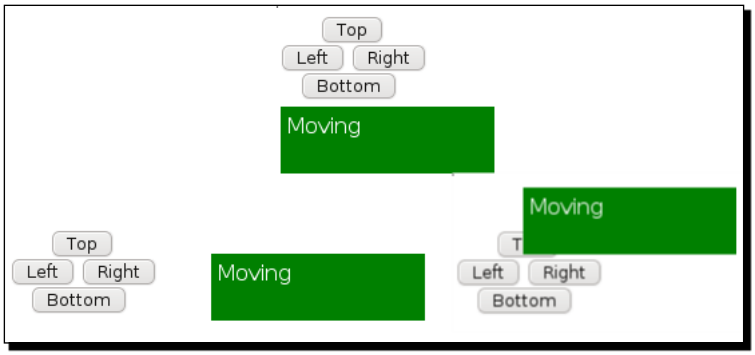
```
$("#dimension-div").animate({ "height": "100px" })
                   .animate({ "width" : "200px" });
```

4. Basically, all numeric CSS properties can be animated, except for a few (that is, colors) that have been added by an officially supported plugin, color (`http://plugins.jquery.com/project/color/`), nonetheless.

5. A moving object is also possible. Refer to the following code for a simple example:

```
<script>
$(document).ready (function () {
    $("#moving-button-top").click (function () {
      $("#moving-div").animate ({ "top": "-=50px" });
    });

    $("#moving-button-left").click (function () {
      $("#moving-div").animate ({ "left": "-=50px" });
    });

    $("#moving-button-right").click (function () {
      $("#moving-div").animate ({ "left": "+=50px" });
    });

    $("#moving-button-bottom").click (function () {
      $("#moving-div").animate ({ "top": "+=50px" });
    });
  });
</script>

<style type="text/css">
  #moving-div {
    background-color: green;
    color: white;
    width: 150px;
    height: 40px;
    padding: 5px;
    position: absolute;
    top: 220px;
    left: 8px;
  }
```

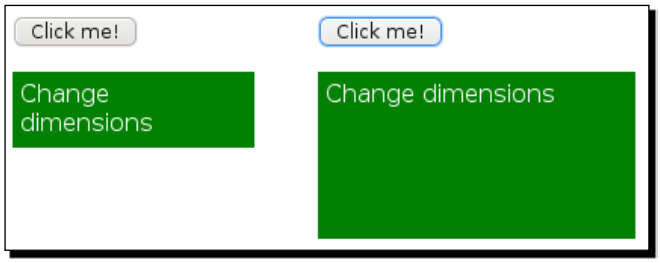
```
</style>
<button id="moving-button-top">Top</button><br />
<button id="moving-button-left">Left</button>
<button id="moving-button-right">Right</button><br />
<button id="moving-button-bottom">Bottom</button>

<div id="moving-div">Moving</div>
```



Having a `-=` (or `+=`) operator will increase (or decrease) to the current value.

6. Last but not least, a `callback` function, which will run once the animation is completed, once for each element, can be specified.

```
$("#callback-button").click (function () {
  $("#callback-div").animate ({ "height": "100px",
    "width" : "200px" }, 400, function () {
    $(this).text ("This is the callback. Animation is complete.");
  });
});
```



Have a go hero – experiment with animations

Look at some of the plugins we have realized in the previous chapters and choose one or two of them that you like the most.

Now, using what you've learned about the `animate()` method, try to add some animation effects to the plugins that you have chosen. For example, if the image plugin is what you have chosen, make an image container of a small size, and then let it grow bigger and bigger, until it fits the image dimensions.

To obtain bouncing effects and other more advanced animation techniques, you may also use the jQuery UI library (`http://jqueryui.com`)—which provides a set of functions to deal with effects, animations, and drag and drop—or the easing plugin (`http://gsgd.co.uk/sandbox/jquery/easing/`).

Don't forget to have a look at the jQuery color plugin, to obtain very cool effects with little effort required, including background animation and a bunch of other effects that you can have a preview of here: `http://plugins.jquery.com/project/color`.

Pop quiz

1. In the majority of the functions that we have seen and used in this chapter, a callback parameter can be passed.

   The callback function is executed once the animation (fade in/out, slide in/out, animate) is complete.

   However, in addition to this, the `animate()` method also has an option (step), which can be specified along with callback.

   Considering the following code, how many times will both functions run and what is the difference between the two of them?

```
.animate({
  "opacity": "show",
  "width": "100px"
}, { complete: callback(), step: step() ;});
```

   A. The complete function is fired every time an animation is complete; as for the example, it will execute twice: once after changing opacity and once after the width has been set to 100px.

      As for step, it will execute once every instance of animate is called with the same `step()` function passed in as an argument.

B. The callback function will be executed once the animation is complete; the step function is called after each step of the animation (thus twice in the example).

C. The callback function will be executed once the animation is complete; the step function is called before the first, and after each step, of the animation (thus three times in the example).

D. None of the above.

2. Considering the following code, what is the most significant difference between what `.parent()` and `.parents()` do?

```
<div>
  <ul>
    <li><span><strong>First</strong> test</span></li>
    <li><span>Another test</span></li>
    <li><span><strong>Last</strong> test</span></li>
  </ul>
</div>

var $ps = $("li:first-child strong").parents ("ul");
var $p  = $("li:first-child").parent ();
```

A. There is no difference in the code snippet, as the returned element is the list in both cases.

B. Referring to the code, the `$ps` variable holds the ancestors of the strong tag (`span`, `li`, `ul`, `div`, `body`, `html`), whereas the `$p` variable represents the parent of the first `li` element (`ul`).

C. Referring to the code, the `$p` variable holds the ancestors of the strong tag (`span`, `li`, `ul`, `div`, `body`, `html`), whereas the `$ps` variable represents the parent of the first `li` element (`ul`).

D. None of the above.

3. As silly as it sounds, what does `end()` do?

A. Removes all event handlers previously attached to the set of matching elements.

B. Stops the currently running animation on the matched elements.

C. Ends the most recent filtering operation in the current chain and returns the set of matched elements to its previous state

D. None of the above.

4. What is the main difference between the functions `children()` and `next()`, which can be used in a very similar fashion?
   A. `children()` gets the children of each element in the set of matched elements; `next()` gets all following siblings of each element in the set of matched elements.
   B. `children()` gets the descendants of each element in the current set of matched elements; `next()` gets the immediately following sibling of each element in the set of matched elements.
   C. `children()` gets the immediately following sibling of each element in the set of matched elements; `next()` gets the children of each element in the set of matched elements.
   D. None of the above.
5. Can items that are to be moved using the `animate()` method be positioned using anything other than absolute?
   A. Yes, the position of objects you wish to move acting on their `left` or `right` properties must not be static (thus can be set to absolute, fixed, or relative).
   B. Yes, the position of objects you wish to move acting on their `left` or `right` properties must be set to either absolute or fixed.
   C. No, the position of objects you wish to move, acting on their `left` or `right` properties, must be set to absolute only.
   D. None of the above.

Summary

Animation plugins are unquestionably useful, though perhaps not much in terms of mere practical utility. However, these plugins provide an interesting addition to the average user interface with little to no effects that we are used to seeing.

However, with the passage of time, over the years, we have witnessed an upward trend in creating very cool interfaces and a number of scripts (developed on some of the most widespread libraries) have been released and made available.

The majority of them are also very easy to use, even for total beginners, leading to the use (sometimes unjustified) of them in nearly every web application.

Most of the animation plugins we will have a chance of analyzing are aimed towards the creation of an "alternative" behavior for those that would otherwise be quite standard elements. For example, the sliding effect is an interesting add-on to the accordion plugin that we have created earlier in this chapter. It adds that fancy twist, which is extremely appreciated nowadays. People just love clicking on animated things to see how they move, change colors, and eventually fade away.

The exact same result, in practical terms (that is, a menu with sub-links or news panes that show after their parent element has been selected—but with no sliding motion), would still be extremely efficient anyway. However, users wouldn't enjoy it as much as they enjoy any smoothly moving thing—be it a box bouncing around the page or, in fact, a couple lines of text sliding up and down.

In a totally different fashion the next chapter deals with plugins—utility plugins—thought of as not actually being visible to visitors in the first place.

In fact, tasks like image preloading, handling cookies, or switching stylesheets are more likely to be a tool for the developer (or the web page owner, that is) rather than a form of entertainment for users. Users will nonetheless obtain some advantages themselves in terms of speed and usability of the web page or site they are reading or browsing.

12
Utility Plugins

Technically speaking, so-called utility plugins are those plugins that employ the `jQuery.pluginName` *functionality, thus classifying into a group of plugins that extend the jQuery object itself and are not chainable, since they don't return the jQuery object.*

In this chapter, however, we will deal with utility plugins in the terms of their purpose. We will cover a series of plugins whose goal is to make it easier for the developer or page owner to access and make use of tools that they may need for the creation of their website.

With this in mind, we will see how two rather simple plugins can be developed so that a series of functions and methods are made available to the user. These will eventually be used to enhance the site layout or to access, modify, and delete data (cookies, that is).

We'll focus on the practical applications and realization techniques that we might use in this chapter to obtain two well-structured plugins. We can easily expand and add to these plugins so that they become even more useful after a few lines of code are added.

Specifically, we will deal with functions and methods to:

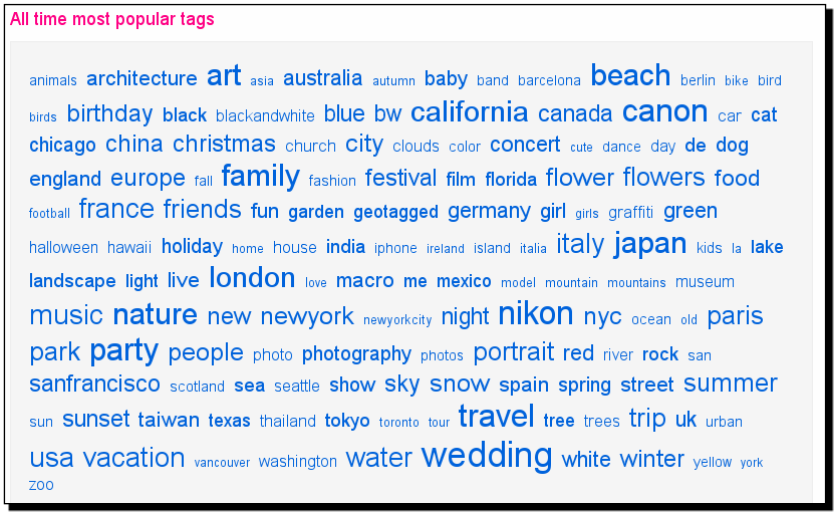
- Generate tag clouds
- Handle cookies

Generating tag clouds

A tag cloud (or word cloud) is a visual presentation of tags (or words) describing the content of a website or document.

Tags are, usually, single words linking to the associated items (articles, pages, or other links) whose importance is shown with either font size or color (or both). The bigger the size of the word, the more important and utilized the word is.

This tag presentation technique was first used by the popular image hosting service *Flickr*, and soon social bookmarking websites such as *de.licio.us* and *Technorati* picked up this idea too. Other users, particularly bloggers, decided to integrate this functionality into their websites to offer a better user experience and a quicker identification of post and topics of interest.



A bit of theory to start with

We better start off with some theory first, as this plugin requires a bit of reasoning as far as font size is concerned.

There are a few key points we must not overlook, namely:

1. A maximum font size should be set: The most common word will be as big as this value, but no greater size is allowed.
2. A minimum font size needs to be set, too: The least common word will back off to this size, but there will never be any smaller text.

3. Words shorter than three characters will be ignored: Conjunctions, articles, and such are not so useful after all.
4. To determine the font size for each tag, we will use the following formula:

$$s_i = s_{min} + (s_{max} - s_{min}) \frac{tag_i - tag_{min}}{tag_{max} - tag_{min}}$$

Where:

- s_i is the display font size for the current tag.
- s_{max} and s_{min} are, respectively, the maximum and minimum font sizes allowed.
- tag_i is the number of times the current tag is repeated (tag count).
- tag_{max} and tag_{min} are, respectively, the maximum and minimum number of times tags have been repeated.

With this in mind, we can start coding our plugin.

Time for action – creating a tag cloud plugin

Our plugin will change the font size of words contained in a parent element (for example, text contained in `<div>` or `<p>` elements) depending on the frequency that the words (tags) repeat with.

1. Create a new directory, called `cloud`, and copy over the necessary files. A new file named `jquery.cloud.js` must also be created, along with the `index.html` file that we'll use to test the script, once finished.
2. The HTML code we start with should look similar to the following:

```
<!DOCTYPE html>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="jquery.cloud.js"></script>
  <script>
    $(function() {
      $('p').cloud();
    });
  </script>
</head>
<body>
```

```
    <p>
      Lorem ipsum dolor sit sit sit sit amet, consectetur
      adipiscing elit. Ut varius tempus felis, et volutpat sapien
      facilisis vitae. Duis consectetur consectetur tempor tempor
      tempor tempor tempor tempor auctor.
    </p>
</body>
</html>
```

3. As usual, our plugin structure does not change, and we can go on specifying the options available and the standard loop cycle:

```
(function($) {
  $.fn.cloud = function(options) {
    var defaults = {
      minFontSizePercentage: 100,
      maxFontSizePercentage: 150
    };

    var o = $.extend(defaults, options);

    return this.each(function() {
      var e = $(this);
      // code here
    });
  };
})(jQuery);
```

4. Next, we have to find a way to read words and count how many times they appear in the text. Their size can be determined on this value. A possible solution would require the following steps:

- Convert all letters to lowercase (or even uppercase, the point is having them all look the same)
- Delete unwanted characters and line breaks
- Put words into an array
- Loop through the array (skipping unwanted words—shorter than three characters) counting how many occurrences we have for each word
- Clear the display area
- Calculate the font size for each tag
- Append each tag with modified size to the container

5. As for the first point, this is quite simple, thanks to regular expressions and built-in functions. We will retrieve the text using the `.text()` method, replace characters, and finally split items so that they can fill the `txtarray`. The following is the first line of our tag cloud plugin:

```
var txtarray = e.text()
             .toLowerCase()
             .replace(/[,.;:]/g,'')
             .replace(/[\n\t]/g,' ')
             .split(' ');
```

6. After having our array filled with words that we still need to count, we make use of a `for` loop to take care of the next few tasks.

```
var len = txtarray.length;

for(i = 0; i < len; i ++) {
  // skip if shorter than 3 characters or all numeric
  if(txtarray[i].length < 3) continue;

  // update tag counter: if it's the first time this tag appears,
  // it means we have to set the counter to 1 (i.e. appeared this
  // time only);
we'll increment the tag count by 1 otherwise.
  tags[txtarray[i]] = tags[txtarray[i]] ? ++tags[txtarray[i]] : 1;
}

// also, we must make sure the maximum and minimum counters are
// still truthful.
for(tag in tags) {
  if(tags[tag] > maxCount) maxCount = tags[tag];
  if(tags[tag] < minCount) minCount = tags[tag];
}
```

7. And, finally, we need to determine the right size using the above-mentioned formula and display the tag.

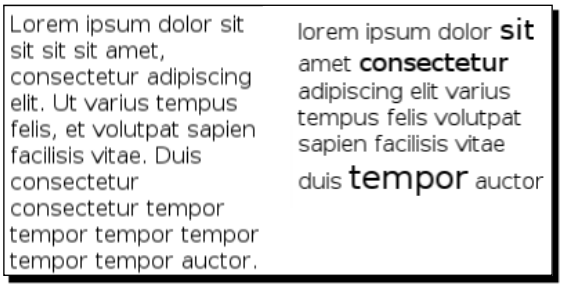
```
e.empty();

for(tag in tags) {
  size = (o.maxFontSizePercentage - o.minFontSizePercentage)
* (tags[tag] - minCount) / (maxCount - minCount) +
o.minFontSizePercentage;

  e.append('<span style="font-size: ' + size + '%">' + tag +
    '</span> ');
}
```

8. The following is the full code for the plugin:

```
(function($) {
  $.fn.cloud = function(options) {
    var defaults = {
      minFontSizePercentage: 100,
      maxFontSizePercentage: 150
    };

    var o = $.extend(defaults, options);

    return this.each(function() {
      var e = $(this);

      var txtarray = e.text()                  // retrieve text
                      .toLowerCase()           // change case
                      .replace(/[,.;:]/g,'')   // delete unwanted
                                                // characters
                      .replace(/[\n\t]/g,' ') // remove line
                                                // breaks
                      .split(' ');          // split words

      var i, count, tags = {}, minCount = 100000, maxCount = 1,
        len = txtarray.length;

      for(i = 0; i < len; i ++) {
        // skip if shorter than 3 characters
        if(txtarray[i].length < 3) continue;

        // update tag counter: if it's the first time this tag
        // appears,
        // it means we have to set the counter to 1 (i.e. appeared
        // this
        // time only); we'll increment the tag count by 1
        // otherwise.
        tags[txtarray[i]] = tags[txtarray[i]] ?
         ++tags[txtarray[i]] : 1;
      }

      // also, we must make sure the maximum and minimum counters
      // are still truthful.
      for (tag in tags) {
        if(tags[tag] > maxCount) maxCount = tags[tag];
        if(tags[tag] < minCount) minCount = tags[tag];
      }
```

```
        e.empty();

        for(tag in tags) {
           size = (o.maxFontSizePercentage -
o.minFontSizePercentage) * (tags[tag] - minCount) /
     (maxCount - minCount) + o.minFontSizePercentage;

          e.append('<span style="font-size: ' + size + '%">' + tag +
'</span> ');
        }
      });
  };
}) (jQuery);
```

The output is similar to the following screenshot:



What just happened?

Apart from the technical realization of the plugin, it could be interesting to understand the reasoning that leads to the determination of the font size for the tag.

We've already seen the formula, and applied it, but why is it that way?

tag_{min} and tag_{max} represent, respectively, the number of times the least-common tag and the most-common tag have been found.

If the value of tag_i is greater than tag_{min} (which should happen most of the time), we obtain a value by which we can multiply the value resulting from the difference of font size ($s_{max} – s_{min}$).

If the number of times the current tag has appeared is equal to the value of tag_{min}, the result from the multiplication will be zero. This is why we are going to add the minimum font size value after all our operations are done.

This series of processes also allows us to maintain a relationship between all of the values obtained this way, as they are all calculated against some constant ratio between other fixed variables.

It's also important to note how the font size varies depending on how many times the tag is found in the text.

Ideally, the font size would change following a logarithmic scale, so that the least common words would be significantly smaller than the most repeated ones, which would show in a very big font.

Logarithmic representation, though, makes sense in larger ranges of values, where power laws actually apply, as pointed out by some sources (`http://www.echochamberproject.com/node/247`).

Have a go hero – make some improvements

Far from perfect, this plugin can't be utilized on blogs or other websites to link to archive categories or pages, due to the fact it makes use of the `.text()` method to retrieve data.

This is quick and easy to change.

Also, there is a minor tweak concerning the title attribute of each word. A short, catchy phrase can be added to inform the user about how many times a certain word has appeared (or, how many blog posts are related to such a keyword).

An additional step would be that of sorting the list of words, based on how many times each word has been repeated in the text.

Cookie handling

As our last (but not least, indeed!) plugin, we're about to create a simple yet extremely useful set of functions, which will help enormously when dealing with cookies.

An important note on this plugin is that, unlike our previous attempts, we are going to extend the `jQuery` object to obtain a function-like plugin that we will call with something such as:

```
$.cookie_do_something();
```

How cookies work

Before we plunge into developing our plugin, it's important to understand how cookies actually work. Even if their functioning is rather simple, sometimes people get confused and try to set cookies for domains other than the one they're working on—which obviously doesn't work.

To avoid any misunderstandings, here is a quick guide to get you started.

set from / cookie path	domain.com	sub.domain.com
domain.com	domain and all subdomains	domain and all subdomains
sub.domain.com	does not work	subdomains only

The two possible scenarios are that we may be working on or browsing a web page located either on a top-level domain (`domain.com`) or any subdomain of the top-level domain (`sub.domain.com` or `www.domain.com`).

As far as `domain.com` is concerned, we are only allowed to specify the top-level path as the cookie path.

Any cookie set this way will be available to all the subdomains as well. This means we can use the same cookie for `sub.domain.com` and `domain.com`.

Things get a bit different when we are working with subdomains. If we set the cookie path to the top-level domain, cookies will be available everywhere, whereas, according to how the first example works, if the domain path is the subdomain itself (thus, we are setting `sub.domain.com` from `sub.domain.com`), cookies will be accessible from that subdomain and its subdomains. We will not be able to access cookies from our `domain.com`.

As a last remark, if we do not specify any domain, cookies are set using the address we are currently browsing.

Time for action – creating a cookie plugin

To put all the good theory into practice, a simple approach to create a plugin with which we can handle cookies follows:

1. Create a new directory called `cookie` that contains all the files we need as well as our plugin `jquery.cookie.js`.
2. We are going to create three functions:
   - `$.setcookie(cookieName, cookieValue, options)`: Creates a cookie with specified options
   - `$.getcookie(cookieName)`: Retrieves a cookie's value
   - `$.delcookie(cookieName)`: Deletes a cookie using the `setcookie` method and setting the cookie value to null; `$.delcookie()` and `$.setcookie()` with value null are equivalent
3. Setting cookies with JavaScript is fairly easy, maybe a bit tricky, but this page on *Quirksmode* (`http://www.quirksmode.org/js/cookies.html`) really helps to understand how things work.
4. We first make sure that options have been specified.

```
var defaults = {
  cookieExpires: 0,
  cookiePath: '',
  cookieDomain: '',
  cookieSecure: 0
};

var o = $.extend(defaults, options);

var name    = cookieName;
var value   = cookieValue;
var expires = o.cookieExpires;
var path    = o.cookiePath ? '; path=' + (o.cookiePath) : '';
var domain  = o.cookieDomain ? '; domain=' + (o.cookieDomain) :
'';
var secure  = o.cookieSecure ? '; secure' : '';

if(expires) {
  // create date
}
```

5. We perform a check on the `expires` variable. If it has not been set (thus left to 0), the cookie is set to expire at the end of the session and we do not have to modify that value. Otherwise, we need to make a string that reports how many days the cookie is intended to last for.

```
if(expires) {{
  var date = new Date();
  date.setDate(date.getDate() + expires;
  expires = '; expires=' + date.toUTCString();
}
```

6. Finally, we create the cookie.

```
document.cookie = name + '=' + encodeURIComponent(value) + expires
+ path + domain + secure;
```

7. We can now move on to the `$.delcookie()` function, which is entirely based on `$.setcookie()`, and only helps in terms of usability. Here is the one-liner:

```
$.setcookie(cookieName, null);
```

8. Finally, `$.getcookie()` has to return the value of the selected cookie. We make sure there are cookies stored, so we can avoid useless iterations, and eventually check each cookie name against the desired name passed as an argument.

```
if(document.cookie) {
  var i, cookie, cookies = document.cookie.split(';');
  var len = cookies.length, nLen = cookieName.length;

  for(i = 0; i < len; i ++) {
    cookie = $.trim(cookies[i]);

    if(cookie.substring(0, nLen + 1) == (cookieName + '=')) {
      return decodeURIComponent(cookie.substring(nLen + 1));
    }
  }
}
return null;
```

9. Using the `jQuery.extend` method, we end up with a plugin looking similar the following:

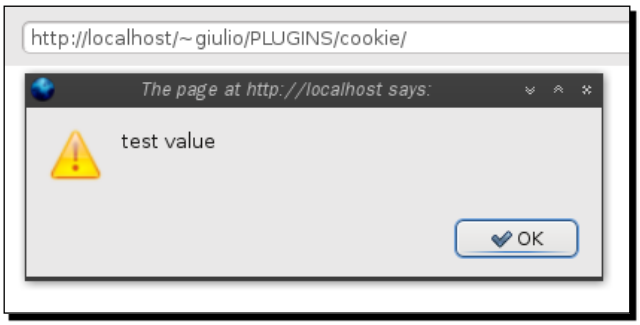
```
(function($) {
  $.extend({
    setcookie: function(cookieName, cookieValue, options) {
      var defaults = {
        cookieExpires: 0,
```

```
      cookiePath: '',
      cookieDomain: '',
      cookieSecure: 0
    };

    var o = $.extend(defaults, options);

    var name    = cookieName;
    var value   = cookieValue;
    var expires = o.cookieExpires;
    var path    = o.cookiePath ? '; path=' + (o.cookiePath) : '';
    var domain  = o.cookieDomain ? '; domain=' +
                  (o.cookieDomain) : '';
    var secure  = o.cookieSecure ? '; secure' : '';

              if(expires) {
                  var date = new Date();
                  date.setDate(date.getDate() + expires);
                  expires = '; expires=' + date.toUTCString();
                }

                document.cookie = name + '=' +
                encodeURIComponent(value) + expires + path +
                domain + secure;
  },
  getcookie: function(cookieName) {

    if(document.cookie) {
      var i, cookie, cookies = document.cookie.split(';');
      var len = cookies.length, nLen = cookieName.length;


      for(i = 0; i < len; i ++) {
        cookie = $.trim(cookies[i]);

        if(cookie.substring(0, nLen + 1) == (cookieName + '='))
        {
          return decodeURIComponent(cookie.substring(nLen + 1));
        }
      }
    }

    return null;
  },
```

```
    delcookie: function(cookieName) {
      $.setcookie(cookieName, null);
    }
  });
}) (jQuery);
```

10. We can now test if it's functioning correctly by creating a few cookies, and then retrieving their values.

```
$(function() {

  // set cookie. Expires at the end of the session, has no path
  // and domain has not been specified
  $.setcookie("testCookie", "test value");

  // pops up the cookie value
  alert($.getcookie("testCookie"));

  // a slightly more complete cookie
  $.setcookie("secondCookie", "test",
                { cookiePath:   "/",
               cookieDomain: "domain.com"
  });

  // deletes cookies
  $.delcookie("testCookie");
  $.delcookie("secondCookie");
});
```



Make sure you don't run this test directly from the file system, as cookies are set via a web server.

What just happened?

The plugin is not very difficult in itself. The trouble might come in when we are about to create the correct syntax to eventually set up (or read) the cookie using the available JavaScript functions.

A few pieces of must-know information that come in handy when speaking of cookies, are as follows:

- If `expires` is set to **any negative** number, the cookie is erased immediately.
- If `expires` is set to 0, the variable is not set and the cookie is trashed right when the browser is closed.
- Dates must be in the UTC/GMT format.
- There are two needed pieces of information: the name of the cookie and its value.

As for the function that reads a cookie's value, everything should be pretty simple and straightforward. We split `document.cookie` on semicolons (";") and we obtain an array of cookies set for the current domain.

We then loop through the array until we stumble upon a cookie with the desired name. Using the substring function, we retrieve the cookie's value and return it.

If no cookie is found, a null value is returned.

Have a go hero – make some small improvements

Very little can be added, except for some further check on types.

For example, the `expires` value (in days) can be used with either a number (of days) or a date, for which we do not need to convert the number of days the cookie is valid to a date.

Also, using the `typeof` function, we are able to check what kind of data we are dealing with, preventing unpleasant situations in which the `expires` variable equals "a few months".

Finally, what happens if the cookie name or value are not set?

Pop quiz

1. Tag clouds surely help to describe the content of a website in an easy-to-read fashion, providing a simple way to scan text and quickly find certain topics and/or pages.

   However, there are elements of a tag cloud that do not help this identification process.

Which are the elements that greatly help focus the attention of the user to the tag cloud area (and on the most important keywords)?

A. **Tag size**: Small tags and words attract more user attention than large tags, since the user is curious about them.

   Color and font weight: Bright colored bold tags are easier to notice as the user's attention is naturally focused on the louder items in the cloud.

   Centering: Tags positioned in the middle of the tags are more likely to draw attention than those placed near the borders.

B. **Tag size**: Large tags and words attract more user attention than small tags.

   Centering: Tags positioned in the middle of the tags are more likely to draw attention than those placed near the borders.

   Number of characters: The longer the tag, the more the user is likely to spend time reading the word. Often the user stops scanning the cloud at this point.

C. **Centering**: Tags positioned in the middle of the tags are more likely to draw attention than those placed near the borders.

   Number of characters: The longer the tag, the more the user is likely to spend time reading the word. Often the user stops scanning the cloud at this point.

D. **Tag size**: Large tags and words attract more user attention than small tags.

   Color and font weight: Bright colored bold tags are easier to be noticed as the user's attention is naturally focused on the louder items in the cloud.

   Centering: Tags positioned in the middle of the tag are more likely to draw attention than those placed near the borders.

   Number of characters: The longer the tag, the more the user is likely to spend time reading the word. Often the user stops scanning the cloud at this point.

2. Speaking of relatively small frequencies, which directly correspond to the number of blog posts or articles or pages dealing with a certain topic, how is font size for words belonging to the tag cloud determined?

   A. While for (very) small frequencies a simple, direct assignment of a number from one (or 100), being the minimum font size—corresponding to word that appears one or one times—to the highest frequency, also corresponding to the biggest font size, is possible, for larger values a scaling should be performed, using a formula similar to the one we have mentioned earlier.

B. A scaling should be always made, no matter the frequency of tags in the website or page, as it allows for a more precise handling of data involved in the calculation.

C. Scaling should be made for small frequencies only, as it's necessary that, with larger values, the relationship between the tags is kept unmodified. Doing otherwise would result in weird differences in size for very similar tag values.

D. None of the above.

3. We all talk a lot about cookies, and we are always concerned about what cookies do and the possible security issues.

   We even developed a plugin to easily create, read, and delete cookies.
   But what actually are cookies and what's their main purpose?

   A. A cookie is a text string, stored by either the web browser or the website, which contains bits of information useful to access or to store data from a website. Cookies can be encrypted for better security and privacy protection purposes.

   B. A cookie is an encrypted text string, stored by either the web browser or the website, which contains bits of information useful to access or to store data from a website.

   C. A cookie is a text string, stored by the web browser, which contains bits of information useful to access or to store data from a website.

   D. None of the above.

4. Creating a plugin to store cookies was not difficult at all, except for the way a cookie must be formatted to be valid.

   What does a cookie, with all the possible parameters, look like to be sure it will be accepted?

   A. `cookieName=cookieValue;expires=Fri, 19 Aug 2011 21:22:23 UTC;path=/;domain=example.com;secure=yes`

   B. `cookieName=cookieValue; expires=Fri, 19 Aug 2011 21:22:23 UTC; path=/; domain=example.com; secure`

   C. `cookieName=cookieValue; expires=Fri, 19 Aug 2011 21:22:23 UTC; path=/; domain=example.com; secure=yes`

   D. None of the above.

Summary

We eventually realized two plugins that can be classified as utility plugins, which thus help in the development of a website, either enhancing the frontend or managing backend tasks.

Of course, these plugins were just two simple solutions that fitted the category, as there would be a lot more concepts to take into consideration and discuss. A great starting point is, as usual, the jQuery website, which has a great listing of various utility plugins (`http://plugins.jquery.com/project/Plugins/category/57`) that provide some sort of service to the end user.

They range from multiple selection tools to binding functions and data handling, as well as some functions that integrate with the built-in methods in an attempt to extend the functionalities the core jQuery library provides.

As reported earlier in this chapter, the majority of these utilities are developed extending the jQuery object itself, thus not allowing chainability but offering the possibility to call the utility function by using as a simple syntax as a normal (non-jQuery) function.

In a totally different fashion, the next chapter, which is also the last one, will present you with a list of ten of the most popular and useful jQuery plugins. For each of them, a brief description and an overview of their functioning will be made available.

As a bonus, examples of use and integration with other scripts and/or jQuery plugins will be provided and explained. The next chapter will attempt to make a short handbook that you can use whenever you feel you need to check on something related to the jQuery plugins we deal with.

13
Top jQuery Plugins

A quick look at some pages on the official jQuery website is enough to understand that there are a lot of jQuery plugins out there. Many plugins even do the same thing, but in different ways; and some do the same thing in the very same way.

But, apart from the inevitable "clones", one may wonder "which are the plugins that I really need or are worth taking a look at?".

According to the official website, the plugins in the following selection are the most popular and appreciated. For each of them, we will analyze a few points (including a brief application section), and see some paragraphs that sum up the essential documentation for the plugin along with some examples.

With many thanks to their authors, who provided the jQuery community with such great plugins for us to use at no cost, the ten plugins we are going to analyze are:

- **Typesearch** by *Lim Chee Aun*
- **JSON plugin** by *Giraldo Rosales*
- **notNow** by *Sergey Vasilianskiy*
- **Webcam** by *David McNamara*
- **Quovolver** by *Sebastian Nitu*
- **ScrollToElement** by *Lauri Huovila* and *Neovica Oy*
- **PassRoids** by *Patrick Keefe*
- **Virtual Keyboard Widget** by *Jeremy Satterfield*
- **Sliding Doors** by *Frank DeRosa*
- **idleTimer** by *Paul Irish*

Typesearch

Information regarding this plugin can be found at the following links:

Project page	`http://plugins.jquery.com/project/typesearch`
Home page	`http://code.google.com/p/typesearch`
Download link	`http://giulio.hewle.com/gstuff/jquery/typesearch/`
Online demonstration	`http://giulio.hewle.com/gstuff/jquery/typesearch/demo.html`

Description

The idea behind Typesearch, sparked by the desire of **Lim Chee Aun** (the author of the plugin) to have a cross-browser OSX-style searchbox, is as simple as it is successful.

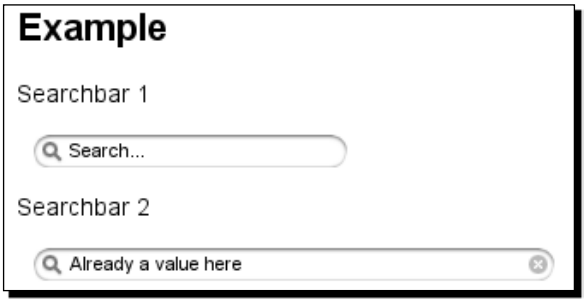
> *This plugin implements Safari's searchbar using* `<input type="search">`*, also mimics its functionality and look on other browser as well.*

The author, inspired by the jQuery implementation of a OSX-like searchbox (`http://www.brandspankingnew.net/archive/2005/08/adding_an_os_x.html`), decided to turn the idea into a jQuery plugin, which is easier to use and far less complicated than writing the same code over and over again for each page we need the search box for.

Needless to say, very few would call the OSX interface "ugly" or even "unattractive". Hence there is a quick and consistent increase in attention towards this tiny but handy script. No matter what, it will always draws interest due to its undeniably good looks and simplicity of use and implementation.

Although available for other JavaScript frameworks such as MooTools, Typesearch provides the best results as a jQuery plugin, giving the user almost full control over its operation and results.

The following image represents what the plugin actually does. As you can see, the two search bars have a OSX look:



Synopsis

The Typesearch plugin allows for an easy-to-use approach and a few lines of code are enough to get it working. The following code represents how functions should be set up and called:

HTML:

```
<input type="text" class="search" value="" />
```

JavaScript:

```
$('.search').typeSearch();
```

Options:

Name	Type	Default	Description
`results`	`int`	`0`	The `results` attribute is set to this value
`placeholder`	`string`	`'Search...'`	Text to display instead of empty text box
`autosave`	`string`	`''`	`autosave` attribute is set to this value

Time for action – obtaining an OSX-like search bar with the Typesearch plugin

To better understand how the plugin works, we are now going to try it out and see how we can obtain an OSX-looking search bar with little effort.

1. As we've already seen, the basic code to make the plugin work is really simple.

   If we haven't applied styling nor have tried to set any options, our code for the web page will look similar to the following:

```
<html>
  <head>
    <link rel="stylesheet" type="text/css" media="screen"
      href="css/typesearch.css" />
    <script type="text/javascript" src="jquery.js"></script>
    <script type="text/javascript" src="jquery.typesearch.js">
    </script>

    <script type="text/javascript">
      $(document).ready(function() {
        $("#searchbox").typeSearch();
      });
    </script>
  </head>
```

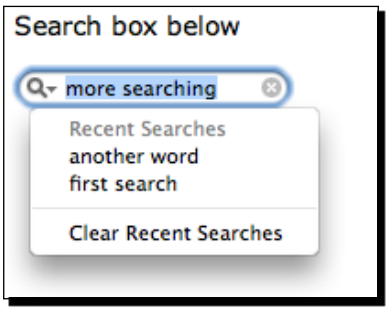
```
  <body>
    <p>Search box below</p>
    <input type="text" id="searchbox" value="" /><br />
  </body>
</html>
```

2. The previous code would have no other effect than activating the plugin with default options (since we passed no arguments) and styling the selected search bar to look like the ones we've seen in the previous image.

3. If we want to show 10 results and enable autosave, we will need to modify the code as follows:

```
<script type="text/javascript">
  $(document).ready(function() {
    $("#searchbox").typeSearch({
                    results: 10,
                    autosave: 'mysearch-id' });
  });
</script>
```

4. The above code, which only works for Safari, as it's the only browser that currently supports such attributes, displays a list of the ten recent searches and shares the list for all the search boxes that have the same `autosave` attribute.



What just happened?

Even though the functioning of the plugin is straightforward, we can get a little confused when dealing with options that actually work with certain browsers only.

In fact, there is no browser other than Safari supporting the `autosave` feature. It is then needless to say this plugin finds major application as a way to make a search bar look the same in every browser, but does not act in the same way for each of them.

Also, make sure you don't mess up file paths! JavaScript files can be moved with no problems, but references to the `images` directory are contained in the CSS file, which should remain in the `css` subdirectory.

You can obviously move the stylesheet or even images, but all paths must be changed—and it's quite time consuming!

Final thoughts

- ✓ Cross-browser and easy to use
- ✓ Nice looking, as most OSX-like things are
- × Sadly, no documentation available. Even though the plugin is so simple, it wouldn't have hurt at all.
- × Difficult to download as there is no packaged version on the official page and the user should have Subversion installed to check out the latest revision.
- × Some code is just plain JavaScript, and some parts can be implemented in jQuery, the final result being a jQuery plugin. The truth is that some JavaScript functions come in handy sometimes, and often run more quickly than their jQuery counterparts.

JSON plugin

Information regarding this plugin can be found at the following links:

Project page	`http://blog.nitrogenlabs.com/2009/03/json-plugin-for-jquery.html`
Home page	`http://blog.nitrogenlabs.com/2009/03/json-plugin-for-jquery.html`
Download link	`http://giulio.hewle.com/gstuff/jquery/labs_json/`
Online demonstration	`http://giulio.hewle.com/gstuff/jquery/labs_json/demo.html`

Description

This plugin, created by Giraldo Rosales, quickly converts JavaScript objects to JSON strings and JSON strings back to objects.

It is particularly useful whenever we are about to pass data to other applications, so that we can count on a certain way in which all the information will be transferred. We need to develop the software necessary to decode (or encode) the received messages with absolute confidence that the standard is already set and shared between the two programs.

Synopsis

This plugin makes the exchange of JSON strings extremely easy, as is the following code that is needed to set up and call its functions:

```
 // ENCODE
$.json.encode(value[, replacer[, space]])

// DECODE
$.json.decode(text[, reviver]);
```

Arguments (encode):

Name	Type	Description
value	obj	Object to be encoded to JSON string
replacer	function	Function to be called for replacing key-value pairs
space	int/string	Number of spaces to indent the values with

Arguments (decode):

Name	Type	Description
text	string	JSON string to be converted back into object
reviver	function	Function to be optionally called at the end

Time for action – encoding and decoding JSON strings

Converting values to and from JSON strings, with just a few lines of code, is rather simple. Here is how we can approach the problem:

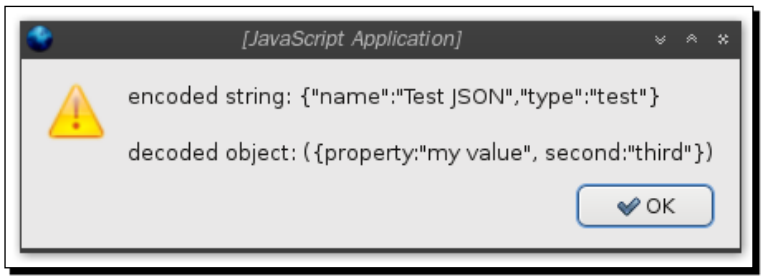
1. We can encode objects into strings using the encode function, after we have successfully created an object.

```
var obj = {};
obj.name = "Test JSON";
obj.type = "test";

$.json.encode(obj);
```

2. We then have the possibility to convert the encoded string back to an object, making use of the decode function.

3. To test the script, we will show a message box with both the results, right after the conversions are made:

```
<html>
<head>
  <script src="jquery.js" type="text/javascript"></script>
  <script src="labs_json.js" type="text/javascript"></script>
  <script type="text/javascript">
    $(document).ready(function() {
      var obj = {};
      obj.name = "Test JSON";
      obj.type = "test";

      alert("encoded string: " + $.json.encode(obj) +
             "\n\ndecoded object: " +
             $.json.decode('{"property":"my value",
              "second":"third"}').toSource());());
    });
  </script>
</head>
<body></body>
</html>
```



What just happened?

To better understand what we've just learned to do, we might want to know a bit more about the JSON standard.

JSON, acronym for JavaScript Object Notation, is a standard designed for exchanging data in a human-readable form.

It basically represents simple sets of associative arrays and data structures of key-value pairs that are, thanks to the JSON format, language independent and can be parsed by a variety of software developed, available for most programming languages.

JSON strings are thus very useful when we need to pass data to (or receive data from) other applications, typically in a server-web application transaction in place of XML.

Final thoughts

- ✓ A **quick** way to handle JSON strings and objects
- ✓ Some **interesting options** to make use of
- × Very **poorly documented**

notNow

Information regarding this plugin can be found at the following links:

Project page	`http://plugins.jquery.com/project/notNow`
Home page	None
Download link	`http://plugins.jquery.com/node/10023/release`
Online demonstration	`http://giulio.hewle.com/gstuff/jquery/notNow/demo.html`

Description

notNow is a jQuery plugin developed by Sergey Vasilianskiy that allows postponing a certain operation (function), for a certain period, once.

Synopsis

The function made available by the plugin can be called as follows:

```
$.notNow(timeOut, func);
```

Arguments:

Name	Type	Description
`timeOut`	`int`	Time out period in milliseconds
`func`	`function`	Function to be executed after timeout

Time for action – postponing a function using the notNow plugin

A sample application of the plugin follows. This should also make clear what the differences are compared with the built-in `setTimeout()` function.

1. From the very basic documentation that's provided, here is a sample application of this simple plugin that actually finds many applications in everyday usage:

```
<html>
```

```
  <head>
    <script type="text/javascript" src="jquery.js"></script>
    <script type="text/javascript"
        src="jquery.notnow.js"></script>
    <script type="text/javascript">
      $(document).ready(function() {
        var myFunc = function() {
          // do something
        }

        $.notNow(5000, myFunc);
      });
    </script>
  </head>
  <body></body>
</html>
```

2. Obviously, we can also use the anonymous function and handle everything without the need to define another named function.

```
$.notNow(2000, function() {
  // load image in 2 sec;
  var image = new Image();
  image.src = 'image.png';
});
```

What just happened?

The notNow plugin is, of course, extremely simple in terms of usage; but it allows for quite a lot of different applications when using it in the real world.

A common problem we may face in an ordinary task such as loading a web page is, in fact, executing a function, or even a set of several functions, after some time.

It's true that, making use of the callback facility jQuery provides, we can chain functions and methods so that one is run right after the previous function has finished. However, with a plugin that postpones this kind of activity and provides a quicker way to access the `setTimeout()` utility, everything is more nicely done.

Final thoughts

- ✓ **Very handy** in many situations
- ✓ Lots of possible uses
- × Once again, no documentation is available, but the plugin is so simple that everybody should be able to understand how it works.

Webcam

Information regarding this plugin can be found at the following links:

Project page	`http://mackers.com/projects/jquery.webcam/examples/basic/basic.htm`
Home page	`http://mackers.com/rant/2010/03/09/571-jquery-webcam`
Download link	`http://mackers.com/projects/jquery.webcam/examples/basic/`
Online demonstration	`http://mackers.com/projects/jquery.webcam/examples/basic/basic.htm`

Description

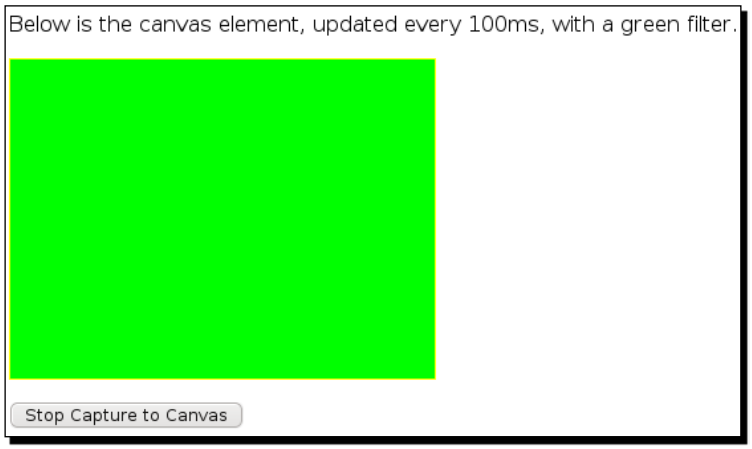
This plugin was written by **David McNamara** and allows jQuery to read data from a user's webcam or other video capture device.

Until browsers support native webcam capture, a Flash SWF is used for the actual capture.

The plugin can optionally prompt the user to allow Flash access to the webcam and writes the video to a canvas element.

It also supports callbacks and filters.

The image below shows the canvas element with a green filter applied. This is pretty much what the plugin will look like right before the webcam is turned on.



Synopsis

The plugin requires some additional code in order to produce the desired results, as a so-called 'canvas', in which the captured images will be displayed, is necessary.

HTML:

```
<canvas id="canvas"></canvas>
```

JavaScript:

```
// INIT
$.webcam.init(canvas[, flashcontainer[, properties]]);

// CALLBACKS ON IMAGE DATA
$.webcam.addCallback(function, interval);

// FILTERS
$.webcam.addFilter(function);

// START/STOP CAPTURE
$.webcam.startCapture();
$.webcam.stopCapture();
```

Arguments (init):

Name	Type	Description
canvas	string	Selector for the canvas element
flashcontainer	string	Selector for the Flash container
properties	obj	Options object

Properties

Name	Type	Description
width	int	Element width
height	int	Element height
interval	int	Update interval
dialogBody	string	Text displayed on dialog element
dialogProperties	obj	Dialog properties object

Time for action – setting up and using the webcam plugin

In order to make the plugin work as it's intended to, we first have to follow a series of steps to set up the script.

1. Firstly, we have to initialize the `webcam` object to an element of our choice, and optionally add a callback function to the image data.

```
$.webcam.init($('#canvas'));
```

2. The webcam can now start capture, which we can interrupt at any moment. We will bind the start and stop functions to buttons that we can easily manage and conveniently enable or disable.

```
$(document).ready(function() {
  $.webcam.init($('#canvas'));

  $("button#start").click(function() {
    $.webcam.startCapture();
    $(this).attr("disabled", true);
    $("#stop").attr("disabled", false);
  });

  $("button#stop").click(function() {
    $.webcam.stopCapture();
    $(this).attr("disabled", true);
    $("#start").attr("disabled", false);
  });
});
```

What just happened?

Setting up the plugin is one of the most important things to do in order to obtain good results, especially if the plugin in question provides lots of different ways to tweak and customize its functioning and appearance.

Required files for the plugin to work are:

- `jquerywebcamhelper.swf` (`http://github.com/mackers/jquery.webcam/tree/master/swf/`)
- jQuery.flash plugin (`http://jquery.lukelutman.com/plugins/flash/`)
- jQuery UI Dialog element and a theme (for the optional user prompt), which can be downloaded from `http://jqueryui.com`

Provided all the above files have been retrieved, an interesting feature the webcam plugin offers is filters.

Filters can be added though the `addFilter` method, which takes one argument (the filter object) and sets the filter color to the one we specified in RGB notation.

The `filter` object that we need to pass has the standard **r** (red), **g** (green), and **b** (blue) properties we can set according to our liking.

We can create and add a filter using this template code.

```
var filter = function(px) {
  px.red = 255;
  px.green = 255;
  px.blue = 255;

  return px;
};

$.webcam.addFilter(filter);
```

Final thoughts

- ✓ **Interesting** idea
- ✓ Great realization with many options and possibilities
- × No extensive documentation provided

Have a go hero – creating a green filter

Now that you know everything you need to know about filters, create a new page in which the webcam plugin will apply a green filter to the image.

This means only the green value must be set to 255, whereas the other values may, at your wish, either be set to 0 or remain untouched.

Quovolver

Information regarding this plugin can be found at the following links:

Project page	`http://plugins.jquery.com/project/Quovolver`
Home page	`http://sandbox.sebnitu.com/jquery/quovolver/`
Download link	`http://sandbox.sebnitu.com/jquery/quovolver/`
Online demonstration	`http://sandbox.sebnitu.com/jquery/quovolver/`

Description

Quovolver is a simple extension for jQuery written by **Sebastian Nitu** that takes a group of quotes and displays them on your page in an elegant way.

The plugin demonstration from the official website is as follows:

Synopsis

The plugin needs to be activated on a selection of elements (the quotes):

```
$('element').quovolver([speed[, delay]]);
```

Arguments:

Name	Type	Default	Description
`speed`	`int`	`500`	Execution speed in milliseconds
`delay`	`int`	`6000`	Delay between quotes in milliseconds

Time for action – putting Quovolver to work

Very easy to use, the first step consists in actually creating the group of quotes we'd like to cycle through.

1. We'll wrap the paragraphs into `blockquote` elements in order to obtain a standard HTML representation of quotations.

```
<html>
<head>
  <script src="jquery.js" type="text/javascript"></script>
```

```
  <script src="jquery.quovolver.js" type="text/javascript">
  </script>
  <script type="text/javascript">
    $(document).ready(function() {
      // code
    });
  </script>
</head>
<body>
  <blockquote>
    <p>
      This is a simple, short quote.
    </p>
    <cite>Author</cite>
  </blockquote>

  <blockquote>
    <p>
      A longer note is coming next. A longer note is coming next.
      A longer note is coming next.
    </p>
    <cite>Another author</cite>
  </blockquote>

  <blockquote>
    <p>
       This would be the longest quote of the set, with some text
       repeated over and over. This would be the longest quote of
       the set, with some text repeated over and over. This would
       be the longest quote of the set, with some text repeated
       over and over. This would be the longest quote of the set,
       with some text repeated over and over.
    </p>
    <cite>Last author</cite>
  </blockquote>
</body>
</html>
```

2. We can now add the JavaScript code to make the plugin work, which only requires a simple line, as follows:

```
$("blockquote").quovolver(200, 10000);
```

3. The first quote is then displayed for 10 seconds, then the animation (lasting only 200ms) takes place, and then the second quotation block appears.

   This happens until all the quotes have been displayed for ten seconds each, at which point the cycle starts over and the first quotation is displayed again.

What just happened?

Even though this plugin was originally thought up as a method to cycle through a set of quotations the principles lying behind Quovolver make it easy to apply the same plugin to other elements in a web page.

Instead of activating its main functionality by selecting a `blockquote` element, we could, in fact use the plugin to cycle through other kinds of media—such as images—to set up a very simple image gallery for our website.

Final thoughts

- ✓ Easy to use
- ✓ Some documentation (mostly examples) available

ScrollToElement

Information regarding this plugin can be found at the following links:

Download link	`http://giulio.hewle.com/gstuff/jquery/js/`
Online demonstration	`http://giulio.hewle.com/gstuff/jquery/scrollToElement/demo.html`

Description

The ScrollToElement plugin has been developed by Lauri Huovila and Neovica Oy.

> *This plugin allows you to scroll the current page; the selected element is shown at the top-left corner of the browser. By default, scrolling is smooth and takes 750ms.*

Synopsis

The plugin provides two functions, which need to be called in different ways and with different arguments.

```
// SCROLL TO SELECTED ELEMENT
$('#element').scrollTo([speed]);

// SCROLL TO ELEMENT
$.scrollToElement(element[, speed]);
```

Arguments:

Name	Type	Default	Description
`speed`	`int/string`	`'normal'`	Animation speed (text or number in ms)
`element`	`object`		Element to scroll to

Time for action – different ways of scrolling

The plugin provides two ways to scroll to an element, which we will analyze.

1. We can either select an element using jQuery built-in selectors and then scroll to it (calling the `.scrollTo()` method) or, using the `scrollToElement()` function, scroll to the selected element passed as an argument.

```
<html>
  <head>
    <script type="text/javascript" src="jquery.js"></script>
    <script type="text/javascript"
      src="jquery.scrolltoelement.js"></script>
    <script type="text/javascript">
      $(document).ready(function() {
        // code
      });
    </script>
  </head>
  <body>
    <button id="scroll1">scrollToElement</button><br />
    <button id="scroll2">scrollTo</button>

    <!-- page should be long enough to allow for scroll
    add <br /> tags as needed -->

    <p id="scrolltarget">scroll target is here</p>
  </body>
</html>
```

2. We can then bind the two buttons to a scrolling motion—the same result through a different process.

   For example, one of the two buttons can be bound to scrolling with the following code, which results in the function being called. The target element must be specified too.

```
$("button#scroll1").click(function() {
  $.scrollToElement($('#scrolltarget'));
});
```

   Alternatively, the method version can be used, and the code would read:

```
$("button#scroll2").click(function() {
  $('#scrolltarget').scrollTo();
});
```

What just happened?

This plugin is of a disarming simplicity, yet it also turns out to be extremely useful for some user experience tweaks to a web page.

Many website designers are used to adding a link to the top of the page (in the footer) or a link to the content (in the header).

In both cases, the "jump" is clear and noticeable; some may dislike this behavior.

On the other hand, using the ScrollToElement plugin, a simple animation can be set up so that the reaching of a determined element is so smooth that everybody enjoys it—and might possibly do it again just to check it out.

Final thoughts

- ✓ Smooth animation
- ✓ Great user experience add on
- × Poor documentation

Have a go hero – proceed a step further

Even though it's not much of a challenge, it's always useful to familiarize yourself and get used to how the plugin works.

In this case, the only other argument that can be passed is the animation length (speed). Everybody can pass another argument to a function. But this is not the most interesting feature to toy around with.

Instead, try to see what situations the method and function versions are best suited for. In what circumstances are methods more useful than functions? What about this plugin?

PassRoids

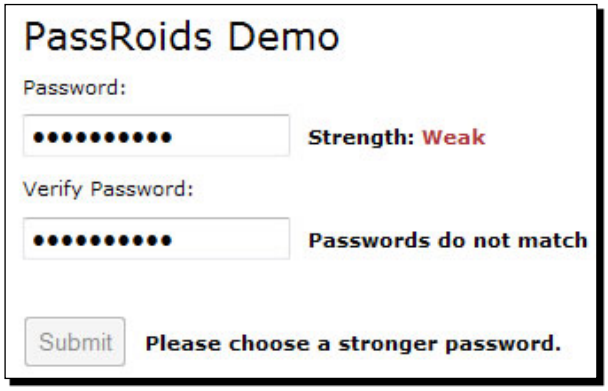
Information regarding this plugin can be found at the following links:

Project page	`http://plugins.jquery.com/project/PassRoids`
Home page	`http://plugins.jquery.com/project/PassRoids`
Download link	`http://plugins.jquery.com/node/9059/release`
Online demonstration	`http://giulio.hewle.com/gstuff/jquery/PassRoids/demo.html`

Description

> *The PassRoids jQuery plugin was designed by* **Patrick Keefe** *to allow simple integration of password strength measurement and verification and display it in a simple method to the end user. Installation requires minimal js and css code and can be setup in a matter of minutes.*
>
> *The script is only ~4K and has been tested and runs on IE6+, Firefox, Safari and Opera.*

A sample PassRoids application is seen in the following screenshot, which shows form and plugin element styling. The button is disabled to prevent the user from submitting data:



Synopsis

A simple line of code is enough to make the plugin work, but some options allow for an easy customization of the plugin functioning.

```
$('element').passroids(options);
```

Options:

Name	Type	Default	Description
main	string	'#password'	Selector for the field
verify	string	null	Selector for the confirmation field
button	string	null	Selector for the button
minimum	int	0	Minimum password strength required

Time for action – using the plugin

A sample application of the PassRoids plugin is shown in the following example, which gives a useful hint for those interested in an quick and easy way to check field values.

1. The basic layout for the page on which the plugin will work should look similar to the following:

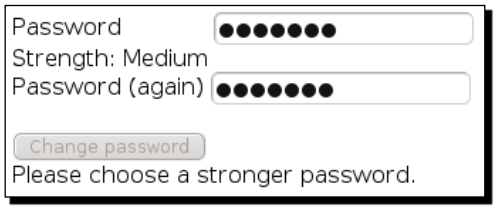
```
<html>
<head>
  <script src="jquery.js" type="text/javascript"></script>
  <script src="jquery.passroids.js" type="text/javascript">
  </script>
  <script type="text/javascript">
    $(document).ready (function () {
      // code
    });
  </script>
</head>
<body>
  <form>
    Password: <input type="password" name="pass_test" value=""
      id="pass_test" />
    Password (again): <input type="password"
      name="pass_test_verify" value="" id="pass_test_verify" />
    <input type="submit" name="pass_test_submit"
      value="Change password" id="pass_test_submit" />
  </form>
</body>
</html>
```

2. It is important to note that some elements must be present in order to use the plugin to its best. These elements are the password verification field (`#pass_test_verify`) and the submit button (`#pass_test_submit`). Of course, a password field (`#pass_test`) is needed to check the text input.

3. The plugin must be called on the `document` itself, and options can be passed all at once using the `options` object.

```
$(document).passroids ({ main: "#pass_test",
                         verify: "#pass_test_verify",
                         button: "#pass_test_submit",
                         minimum: 10
                       });
```

4. Our form will then look like the one in the following image:



Once we type in the password, the script will either enable the button or keep it disabled to prevent the form submission.

Also, a message reporting the password strength will be displayed.

What just happened?

A very strange thing we can notice is how the plugin needs to be called for it to work correctly.

The default value for `main` makes the plugin work by default on a `#password` field, and **not** on the field(s) that are selected using jQuery selectors and on which we actually expect the script to be used.

Needless to say, this unconventional approach makes it almost essential for us to pass options even for basic usage; and obviously makes everything worse and more difficult to use, resulting in an overall less user-friendly approach.

We can also style the plugin output, knowing that:

- `div#psr_score` contains strength information
- `div#psr_verify` is used to display whether passwords match
- `div#psr_strength_notice` shows password strength warnings

Moreover, the score level (Weak, Medium, and so on) is wrapped into a span element whose class is `psr_0` for Weak, `psr_1` for Medium, and so on.

Password strength is expressed by a number, which is calculated as the sum of all of the points obtained by meeting the following criteria:

- length
  - 4 or less: `3`
  - 5-7: `6`
  - 8-15: `12`
  - 16+: `18`
- letters
  - At least one lowercase: `1`
  - At least one uppercase: 5
- numbers
  - At least one: `5`
  - At least three: `5`
- special characters
  - At least one: `5`
  - At least two: `5`
- combos
  - Uppercase and lowercase letters: `4`
  - Letters and numbers: `4`
  - Letters, numbers, and special characters: `7`

Final thoughts

- ✓ Cross-browser
- ✓ Extremely lightweight script
- × No documentation available; a quick look at the code helps, though
- × Could have been easier to use

Virtual Keyboard Widget

Information regarding this plugin can be found at the following links.

Project page	`http://plugins.jquery.com/project/virtual_keyboard`
Home page	`http://snipplr.com/view/21577/virtual-keyboard-widget/`
Download link	`http://giulio.hewle.com/gstuff/jquery/virtual-keyboard/`
Online demonstration	`http://giulio.hewle.com/gstuff/jquery/virtual-keyboard/demo.html`

The following image represents what the virtual keyboard looks like (QWERTY layout, but this can be changed with ease):

Description

> *An on-screen virtual keyboard embedded within the browser window, which will pop up when a specified entry field is focused. The user can then type and preview their input before Accepting or Canceling.*

Developed by **Jeremy Satterfield**.

Synopsis

Basic code to call the method and provide options in the right way follows:

```
$('#input').keyboard(options);
```

Options:

Name	Type	Default	Description
`layout`	`string`	`'qwerty'`	Specify which keyboard layout to use
`customLayout`	`array`		Specify a custom layout (array of arrays)

Layout values accepted:

Name	Description
`qwerty`	Standard QWERTY layout (default)
`dvorak`	Simplified DVORAK layout
`alpha`	Alphabetical layout
`num`	Numerical (ten-keys) layout
`custom`	Uses custom layout defined by `customLayout`

Special/"Action" keys:

Name	Description
`{accept}`	Updates element value and closes keyboard
`{bksp}`	Backspace
`{cancel}`	Clears changes and closes keyboard
`{dec}`	Decimal for numeric entry, only allows one decimal
`{neg}`	Negative for numeric entry
`{return}`	Return/New Line
`{shift}`	Shift/Caps lock
`{sp:#}`	Adds # blank spaces (1 ~ width of one key)
`{space}`	Space bar

Time for action – using the virtual keyboard plugin

Creating an on-screen keyboard might sound as a rather difficult task, but thanks to this plugin it's as easy as writing a couple of lines of code.

1. Basic usage is really straightforward. Note that **jQuery UI** (version 1.7) is required for this plugin to work. It does not seem to work with versions 1.8 and higher. You can get it at `http://jqueryui.com`.

```
<html>
<head>
  <script src="jquery.js" type="text/javascript"></script>
  <script src="jquery-ui.js" type="text/javascript"></script>
  <script src="jquery.keyboard.js" type="text/javascript">
  </script>
  <script type="text/javascript">
    $(document).ready(function() {
      $('input[type=password]').keyboard ({
        layout:"qwerty"
      });
    });
  </script>
</head>
<body>
  <input type="password" name="pass" value="" id="pass" />
</body>
</html>
```

2. We then add some CSS code (that comes together with the plugin), to make the keyboard look better and spaces be of the exact width of keys.

```
  <style>
    .ui-keyboard{ position: absolute; z-index: 16000;}
    .ui-keyboard-button{height: 2em; width: 2em; margin: .1em;}
    .ui-keyboard-actionkey{width: 4em;}
    .ui-keyboard-space{width: 15em;}
    .ui-keyboard-preview{width: 100%; text-align: left;}
  </style>
```

3. To change the layout, a couple of additional lines can be added, defining our custom layout:

```
$('input[type=password]').keyboard({
  layout:"custom",
  customLayout:
    [["q w e r t y {bksp}","Q W E R T Y {bksp}"],
    ["s a m p l e {shift}","S A M P L E {shift}"],
    ["{accept} {space} {cancel}","{accept} {space} {cancel}"]]
});
```

4. The final result is a virtual keyboard with a definitely weird but customized layout:

What just happened?

Note that the CSS used in the examples is not required per se, but helps to make the entire virtual keyboard more consistent in its design.

As you have surely noticed, the `customLayout` option is really important in the creation of a custom keyboard layout. To clarify a little about this entity, we must remember that:

- It is an array of arrays
- Each internal array is a new keyboard row
- Each internal array can contain either one or two string elements (lowercase and uppercase respectively)
- Each string element must have each character or key separated by a space

Final thoughts

- ✓ Simple straightforward plugin of everyday usage for some
- ✓ Very customizable
- ✓ There is some documentation written as comments in the plugin file
- × No ready-to-download package
- × Requires jQuery UI

Sliding Doors

Information regarding this plugin can be found at the following links:

Project page	`http://plugins.jquery.com/project/slidingdoors`
Home page	`http://surreal-dreams.com/jquery-plugins-and-stuff/slidingdoors-plugin/`
Download link	`http://surreal-dreams.com/jquery-plugins-and-stuff/slidingdoors-plugin/`
Online demonstration	`http://surreal-dreams.com/jquery-plugins-and-stuff/slidingdoors-plugin/`

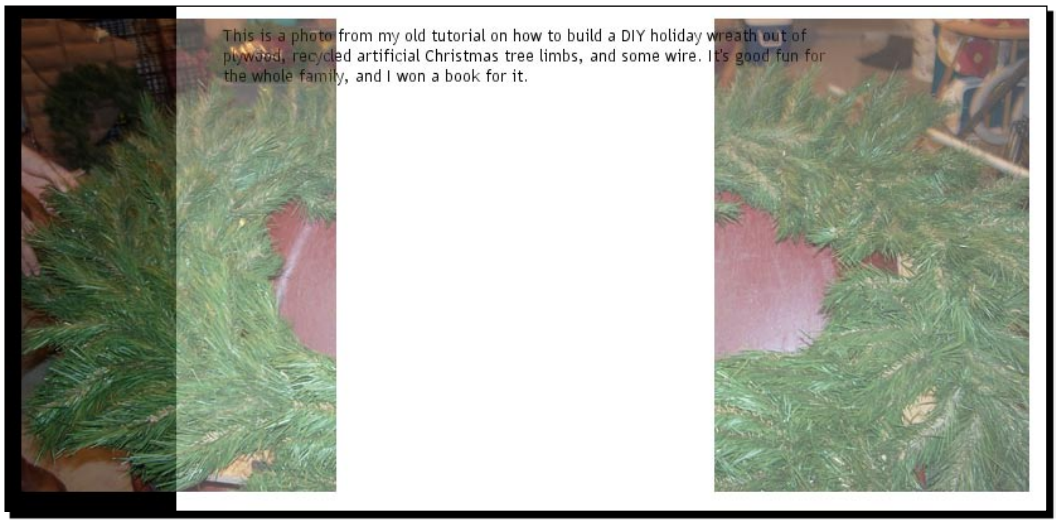
Description

This plugin, written by **Frank DeRosa***, will take a selected image or set of images and replace it with a container with two halves, which slide open and turn translucent on mouseover. The open halves reveal the alt tag text of the original image. Any links on the image are preserved along with their classes, and a configurable text label is added to the alt text. You can configure the time to open the "doors", close them, the opacity, the text indicating there's a link, and how far open the doors go (as a percentage).*

There is one known bug, which is that the original link is not removed and makes the entire structure a clickable link.

The image needs to have a set size or some browsers won't be able to figure out the size. Also, please ensure that your image is not scaled in HTML – the script isn't able to scale back the halves to match the original so it will look strange.

The screenshot represents how the image will look, after the two halves have been moved and the underlying content is visible:



Synopsis

JavaScript:

```
$('element').slidingDoor (options);
```

Options:

Name	Type	Default	Description
closeSpeed	int	500	Time for the doors to close, in ms
linkText	string	'click here'	If there's a link, this text appears
opacity	float	0.2	Opacity at the end of the animation
openSpeed	int	200	Time for the doors to open, in ms
openWidth	float	0.9	Percentage of total image width the doors will slide

Time for action – creating a sliding door

The sliding door technique, though highly effective and relatively difficult in its realization, is surprisingly easy to obtain using this plugin.

1. For example, the following code changes the open and close times and modifies the distance the "doors" slide, making sure the two parts of the image do not become extremely transparent at the end of the animation:

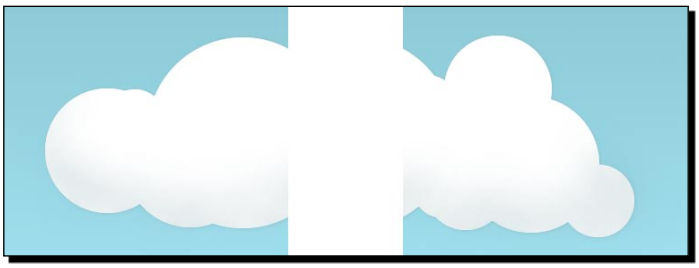
```
$(document).ready (function () {
  $('img').slidingDoor ({
    opacity :   .80,
    openWidth:  .2,
    openSpeed:  100,
    closeSpeed: 100
  });
});
```

2. Now the only thing we need to do in order to make the plugin work, is to add an image to the page.

```
<html>
<head>
  <script src="jquery.js" type="text/javascript"></script>
  <script src="jquery.slidingdoors.js" type="text/javascript">
  </script>
  <script type="text/javascript">
    $(document).ready (function () {
      $('img').slidingDoor ({
        opacity :   .80,
```

```
        openWidth:  .2,
        openSpeed:  100,
        closeSpeed: 100
      });
    });
  </script>
</head>
<body style="text-align: center">
  <><img src="cloud.png" alt="Underlying text"="" />
</body>
</html>
```

3. This is what we will see whenever we hover the image with our mouse pointer. The image looks like it has been split into two halves, and it actually does get split with a smooth animation lasting for a defined period of time.



What just happened?

When dealing with this plugin, an important thing we should bear in mind is the different use of the term "speed" plugin authors make in their works.

The word "speed" is generally used to identify the speed at which the plugin executes its task. In this case, however, both `openSpeed` and `closeSpeed` actually represent the *time* the plugin needs to open and close the sliding door.

Final thoughts

- ✓ Incredible results with minimum code
- ✓ Some documentation available; more is supposed to "come soon"
- × Minor improvements to code and final result still possible

idleTimer

Information regarding this plugin can be found at the following links:

Project page	`http://plugins.jquery.com/project/idleTimer`
Home page	`http://paulirish.com/2009/jquery-idletimer-plugin/`
Download link	`http://github.com/paulirish/jquery-idletimer/`
Online demonstration	`http://paulirish.com/demo/idle-timer`

Description

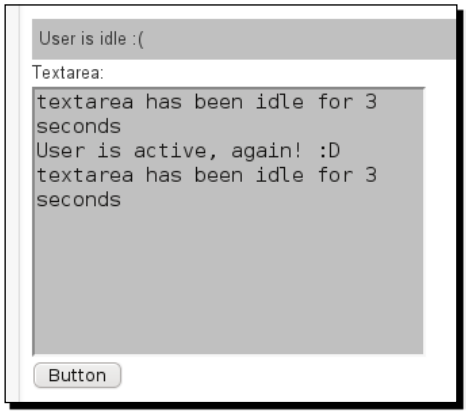
Paul Irish states there are a few cases where you want to know if the user is idle. Namely:

- You want to preload more assets
- You want to grab their attention to pull them back
- You want to close their banking session after 5 minutes of inactivity. (Jerk!)
- You want the site to sneak off the screen and see if they notice ;-)

The plugin has the following characteristics:

- Leveraged event namespaces for easy unbinding
- Considered mousewheel as activity, in addition to keyboard and mouse movement
- Multiple timers support

A simple way is to use the idleTimer plugin to check if the user is active on a web page or a specific element.



Synopsis

This is how we call the plugin on various elements:

```
// START TIMER
$(element).idleTimer([timeout]);

// ELAPSED TIME
$(element).idleTimer('getElapsedTime');

// STOP TIMER
$(element).idleTimer('destroy');

// USER ACTIVITY ('idle' or 'active')
$.data(element,'idleTimer')
```

Options:

Name	Type	Default	Description
`timeout`	`int`	`30000`	Timeout in milliseconds

Time for action – timing idle users

We can set the idle timer so that it checks whether the user is active on the entire document element (that is, the user moves the mouse) or not.

1. The first step is to set a `timeout` variable to which we can refer to, and then bind the functions we chose to the document element.

```
var timeout = 10000;

$(document).bind("idle.idleTimer", function() {
  // user is idle
});

$(document).bind("active.idleTimer", function() {
  // user is active
});
```

2. When the user is active (`active.idleTimer`) we may want to let everybody know. Obviously the same applies to the idle time (`idle.idleTimer`):

```
$(document).bind("idle.idleTimer", function() {
  $("#status").html("User is idle").css("backgroundColor",
    "silver");
});
```

```
$(document).bind("active.idleTimer", function() {
  $("#status").html("User is active").css("backgroundColor",
    "yellow");
});
```

3. Once everything has been correctly set up, we are ready to call the `$.idleTimer()` function and let the plugin do its job:

```
$.idleTimer(timeout);
```

What just happened?

Note that the preceding code is for Version 0.9 and above only.

If you're using the old `$.idleTimer` API, you should not do `$(document).idleTimer(...)`.

Element-bound timers will only watch for events inside of them. You may just want page-level activity, in which case you may set up your timers on `document`, `document.documentElement`, and `document.body`.

Final thoughts

- ✓ Useful for some web applications
- ✓ Some examples to refer to
- × Might be tricky to make the best use of it at first

Have a go hero – use multiple timers

The *idleTimer* plugin, in its most recent version, allows for several timers to be set at the same time, for different elements.

Try to apply two different `timeout` values to two timers that check whether the user is idle or not on certain elements.

Make sure that the functions do different things, or at least display different colored text.

Pop quiz

1. When do you use the Typesearch plugin?
   A. Whenever a search box needs to be added to a web page.
   B. When a certain consistency between browsers is required.
   C. When there's no other way to insert a search box in a webpage.
   D. None of the above.

2. What is JSON?
   A. JSON is a lightweight text-based open standard designed for human-readable data interchange, derived from JavaScript.
   B. JSON is a set of rules for encoding documents in machine-readable form. Tags need to be closed.
   C. JSON provides a means to create structured documents by denoting structural semantics for text.
   D. None of the above.

3. How can virtual keyboards serve as a method to protect users' privacy?
   A. They do not increase security.
   B. By reducing the risk of shoulder surfing.
   C. By reducing the risk of keystroke logging.
   D. None of the above.

4 How can the webcam plugin we used previously make use of our webcam?
   A. Using an external Flash wrapper to access the webcam device.
   B. JavaScript provides a built-in method to easily turn the camera on and start the capture.
   C. A core feature of jQuery is responsible for the webcam integration.
   D. None of the above.

5. Is the Quovolver plugin able to cycle through images as well as blockquotes?
   A. Yes. changing the selector in order to act on a set of images is enough for it to work in a different way.
   B. No, there is no way in which the plugin will be able to cycle through images.
   C. Yes, but images must be contained inside blockquote elements.
   D. None of the above.

Summary

By the end of our analysis, we can actually say we have covered quite a number of topics; and this last chapter has surely helped us understand what the real world for jQuery plugins is like.

Although very few developers care about good documentation and code "best practices", we have to admit they do offer us some really useful software to get our hands dirty with. Most of the plugins we have realized (and used in this chapter) are useful not only to us (developers who had a problem), but also to a lot of other people we don't even know exist—as strange as it sounds.

And if we ever decide to make our works available for others to download, read, modify, or whatever, we should not forget about writing a few lines to help newcomers understand what the whole code is about, what some code snippets do, and how the code does it.

Some useful links and offline resources for further reference can be found in Appendix A.

A
Tools, reference, and final recommendations

The following links, documents, and resources are a great starting point for learning about and expanding your knowledge on jQuery-related topics.

In fact, documentation plays an important role in a programmer's life. Documentation pages (are supposed to) provide exhaustive descriptions of how things work, best practices on how to make use of a particular feature, and finally examples and sample programs to better understand the methods functioning.

Anyway, here are a handful of interesting writings that could come in useful at any time.

Reference and bibliography

Some of the most useful sources of information related to jQuery and web development are as follows:

Official jQuery documentation

Of course, the official jQuery website provides a lot of useful information, featuring examples and user comments as a plus.

- `http://plugins.jquery.com`
- `http://docs.jquery.com`
- `http://docs.jquery.com/Plugins/Authoring`
- `http://docs.jquery.com/Tutorials#Plugin_Development`

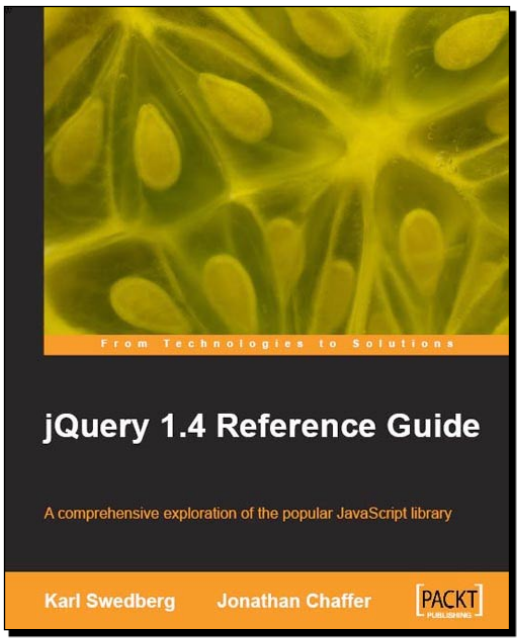
jQuery API browser

The API browser is intended to give the reader a good understanding of functions and methods that they can use and implement into their software.

`http://api.jquery.com`

jQuery 1.4 Reference Guide

Written by Jonathan Chaffer and Karl Swedberg, it is "a comprehensive exploration of the popular JavaScript library".



You probably won't read it from cover to cover, and it can be considered an offline version of the jQuery documentation. However, I like having a real book next to me, on which I can write, underline, and highlight—and turn pages.

Buy it now from `https://www.packtpub.com/jquery-1-4-reference-guide/book`.

Blogs to follow and websites to bookmark

Even online, there are loads of resources and websites or blogs dealing with jQuery. It's true that a vast majority of them actually deal with jQuery only (and not specifically with plugin development). However, there are many bits we can link and merge together to get the whole picture clearly sorted out.

jQuery blog

jQuery developers regularly post blog entries on the official jQuery blog. Needless to say, this is a must-read!

`http://blog.jquery.com`

jQuery UI blog

If you're interested in the jQuery UI library, its official blog is a good starting point to get to know a little better how the library works and why.

`http://blog.jqueryui.com`

John Resig

Though seldom updated, the site hosts many interesting posts and projects by the jQuery author himself.

`http://ejohn.org`

Learning jQuery

Learning jQuery is a multi-author weblog providing jQuery tutorials, demos, and announcements. The tutorials are for all skill levels, and each entry is categorized by level of difficulty.

`http://www.learningjquery.com`

Jörn Zaefferer (bassistance)

The author is keen on jQuery development and has created many quality jQuery plugins. As a side note he also blogs about music.

`http://bassistance.de`

jQuery for designers

The last post (at the time of writing) is dated back to July 2010, but the site has relatively big archives with some useful tutorials worth reading.

`http://jqueryfordesigners.com`

jQuery HowTo

jQuery HowTo is another great resource for tutorials and tips about jQuery. There are more than one hundred posts related to jQuery development.

`http://jquery-howto.blogspot.com`

On browsers: compatibility, comparisons, and plugins

When it comes to web development, it is best if we have some sort of tool that we can use in case we need to debug some code, or examine a web page.

Supported browsers

jQuery supports the following browsers:

- **Firefox** 2.0+
- **Internet Explorer** 6+
- **Safari** 3+
- **Opera** 9+
- **Chrome** 1+

The library should also work fine with Firefox 1.0.x and Konqueror. Further information can be found at `http://docs.jquery.com/Browser_Compatibility`.

Compatibility master table

A very useful resource if you want to learn more about browser compatibility in terms of features and standards supported. It is hosted on Quirksmode, courtesy of Peter-Paul Koch at `http://www.quirksmode.org/compatibility.html`.

Browser plugins

There are many plugins, extensions, and add-ons that are available for inspecting and debugging JavaScript and CSS/HTML code, some of which are extremely useful and we might not want to miss.

FireBug (Firefox)

Essential for the jQuery developer, Firebug integrates with Firefox to put a wealth of web development tools at your fingertips while you browse. You can edit, debug, and monitor CSS, HTML, and JavaScript live on any web page.

Get it from `http://getfirebug.com`.

Also, the possibility to write extensions for Firebug has contributed to the creation of FireQuery (`http://firequery.binaryage.com`), a Firebug extension for jQuery development. Some features are as follows:

- jQuery expressions are intelligently presented in the Firebug Console and DOM inspector.
- Attached jQuery data are first-class citizens.
- Elements in jQuery collections are highlighted on hover.
- jQuerify: Enables you to inject jQuery into any web page.
- jQuery Lint: Enables you to inject jQuery Lint into page being loaded automatically (great for ad hoc code validation).

Internet Explorer 8 Developer Tools

Microsoft published a guide on using Internet Explorer 8 built-in Developer Tools suite at `http://msdn.microsoft.com/en-us/library/dd565622%28v=VS.85%29.aspx`, which also explains how to debug JavaScript code with their debugging tool.

DebugBar (Internet Explorer)

The DebugBar (`http://debugbar.com`) provides a DOM inspector as well as a JavaScript console for debugging, especially useful for IE 6 and 7.

Also, from the same website we can access an extremely useful IE tester tool, which is a "Browser Compatibility Check for Internet Explorer Versions from 5.5 to 9".

Check it out at `http://www.my-debugbar.com/wiki/IETester/HomePage`.

Safari Web Inspector

Safari comes with a built-in Web Inspector tool to analyze DOM and JavaScript.

To enable the Web Inspector, open **Preferences**, go to the **Advanced** tab, and select the **Show Develop menu in the menu bar** item.

More information on Web Inspector is available at `http://trac.webkit.org/wiki/WebInspector`.

Dragonfly (Opera)

The built-in tool is very promising and allows for CSS, DOM inspection, and editing and JavaScript debugging.

You can read more about this tool at `http://www.opera.com/dragonfly/`.

Chrome Web Inspector

The Chrome Web Inspector, which is always enabled, makes it really easy to analyze and manipulate DOM and JavaScript.

More information is available at `http://www.google.com/chrome/intl/en/webmasters-faq.html#tools`.

Information on Web Inpsector can be found at: `http://trac.webkit.org/wiki/WebInspector`.

Cheatsheets

Cheatsheets are useful and practical references that contain API references with detailed description and some sample code: `http://www.cheat-sheets.org/#jQuery`.

jQuery plugin development checklist

Some basic key points to remember are as follows:

- jQuery documentation is your best friend. Always go back to the documentation pages when in trouble or in need of information. Reading it thoroughly wouldn't hurt, either.
- It might sound stupid, but always remember to link to the `jquery.js` file (containing the jQuery library) or, no matter what, we might spend hours looking for some error that justifies the script not running or working properly.
- Always prefer the document-ready statement to any other non-jQuery functions to check whether the page has already loaded or not. Enough said.
- This can never be stressed enough: plugins based on methods are completely different to plugins based on functions.
    - Method plugins extend the `jQuery.fn` object.
    - Function plugins extend the `jQuery` object directly.
    - Method plugins do support chainability.
    - Function plugins do not support chainability.
    - Method plugins should always return the `jQuery` object (`this`, in the code), to allow for chainability.

  Also refer back to Chapters 2 and 3 for more details.

As a plus, Keith Wood has written a very useful article about the plugin framework that he makes use of when developing plugins, which you may find interesting.

The article, which can be found at `http://keith-wood.name/pluginFramework.html`, is an extremely detailed dissertation covering all a developer needs to know about the jQuery plugin structure. It deals with singletons, event binding, settings, and destroy functionalities. It is a must-read for all jQuery plugin developers out there.

B
Pop Quiz Answers

Chapter 1: What is jQuery About?

Question number	Answer
1	C
2	D
3	A
4	C
5	C

Chapter 2: Plugins Basics

Question number	Answer
1	C
2	C
3	D
4	B
5	A
6	A
7	A
8	D
9	C
10	B

Chapter 3: Our First jQuery Plugin

Question number	Answer
1	D
2	A
3	C
4	C
5	B
6	D
7	D

Chapter 4: Media Plugins: Images Plugins

Question number	Answer
1	B
2	B
3	A

Chapter 5: Media Plugins: Audio Plugins

Question number	Answer
1	C
2	D
3	A
4	B
5	B

Chapter 6: Media Plugins: Video Plugins

Question number	Answer
1	D
2	C
3	A

Chapter 7: Form Plugins

Question number	Answer
1	C
2	C
3	A
4	D
5	B
6	A
7	D
8	C
9	B
10	B

Chapter 8: User Interface Plugins

Question number	Answer
1	A
2	B
3	C
4	D

Chapter 9: User Interface Plugins: Tooltip Plugins

Question number	Answer
1	C
2	B
3	D
4	A

Chapter 10: User Interface Plugins: Menu and Navigation Plugins

Question number	Answer
1	C
2	B
3	A

Chapter 11: Animation Plugins

Question number	Answer
1	B
2	A
3	C
4	D
5	A

Chapter 12: Utility Plugins

Question number	Answer
1	D
2	A
3	C
4	B

Chapter 13: Top jQuery Plugins

Question number	Answer
1	B
2	A
3	C
4	A
5	A

Index

Symbols

A

B

C

D

E

F

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Thank you for buying
jQuery 1.4 Plugin Development Beginner's Guide

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

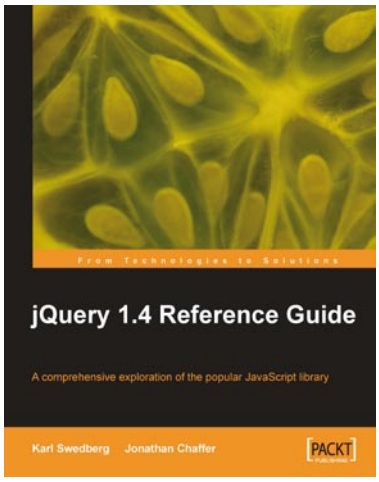
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



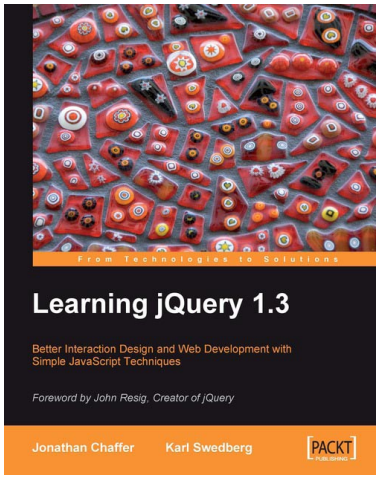
jQuery 1.4 Reference Guide

ISBN: 978-1-849510-04-2 Paperback: 336 pages

A comprehensive exploration of the popular JavaScript library

1. Quickly look up features of the jQuery library
2. Step through each function, method, and selector expression in the jQuery library with an easy-to-follow approach
3. Understand the anatomy of a jQuery script
4. Write your own plug-ins using jQuery's powerful plug-in architecture

---

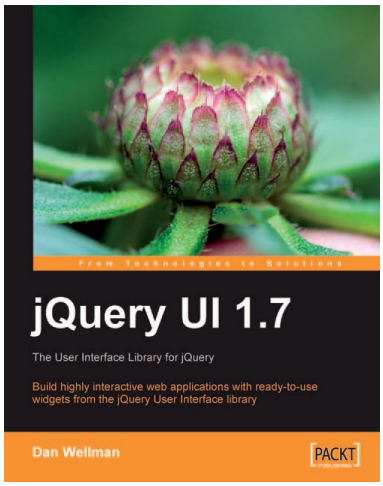


Learning jQuery 1.3

ISBN: 978-1-847196-70-5 Paperback: 444 pages

Better Interaction Design and Web Development with Simple JavaScript Techniques

1. An introduction to jQuery that requires minimal programming experience
2. Detailed solutions to specific client-side problems
3. For web designers to create interactive elements for their designs

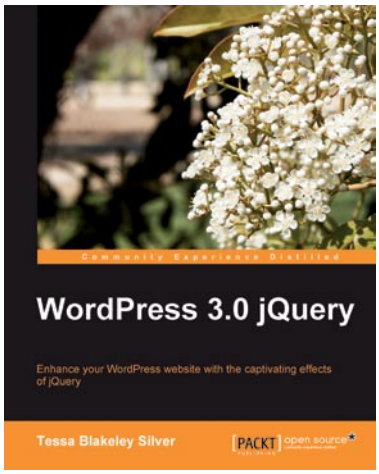


jQuery UI 1.7: The User Interface Library for jQuery

ISBN: 978-1-847199-72-0 Paperback: 392 pages

Build highly interactive web applications with ready-to-use widgets from the jQuery User Interface library

1. Organize your interfaces with reusable widgets: accordions, date pickers, dialogs, sliders, tabs, and more
2. Enhance the interactivity of your pages by making elements drag-and-droppable, sortable, selectable, and resizable
3. Revised and targeted at jQuery UI 1.7



WordPress 3.0 jQuery

ISBN: 978-1-849511-74-2 Paperback: 316 pages

Enhance your WordPress website with the captivating effects of jQuery

1. Enhance the usability and increase visual interest in your WordPress 3.0 site with easy-to-implement jQuery techniques
2. Create advanced animations, use the UI plugin to your advantage within WordPress, and create custom jQuery plugins for your site
3. Turn your jQuery plugins into WordPress plugins and share with the world