

Introducing jQuery Part 2

Welcome to Part 2 of the jQuery introductory series. In Part 1 we looked at some of the most foundational aspects of the library including how basic element selection is carried out and how jQuery methods can be used on the fundamental jQuery objects created following element selection.

In this part of the tutorial we're going to look at how events are handled within the jQuery framework and how we can detect and react to them in our own web applications and the methods available to you to build an effective event solution.

The main Event

A whole component of the jQuery library is dedicated to the detection and reaction of standard browser events which are likely to occur frequently (during every visit to our page) when visitors view our pages. We can even define our own custom events that have never been seen or heard of before and add code routines to handle these as well. Events are a key foundational part of any web application and allow you to define and react to interactions between your site and your visitors.

One event that is used almost every time the jQuery library is engaged is the `ready` event. This can be used in either the full `$(document).ready(function(){});` or the shorthand `$(function(){});` formats, and will execute the code within its curly braces when the document is ready, which is typically once the page has finished loading (except on pages with a lot of images, where the DOM may be ready before all of the images have finished loading). It's an extremely useful construct and something I have used in every single one of my jQuery implementations.

First of all, let's look at a brief example of how a basic, standard event can be handled using the jQuery library. In a blank page in your text editor, start off with the following very basic page:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
  <head>
    <title>jQuery Event Example</title>
    <script type="text/javascript" src="jquery/jquery-1.2.1.min.js"></script>
  </head>
  <body>
    <button id="button1">Click Me!</button>
  </body>
</html>
```

Now add the document `ready` and the `click` event handlers in their own `<script>` tag in the `<head>` of the page:

```
<script type="text/javascript">
$(function(){
  $("#button1").click(function() {
    alert("You clicked " + $(this).attr("id"));
  });
});
</script>
```

The `.click()` method, known as an Event Helper, is attached directly to the jQuery object representing the element from the DOM with an `id` of `button1`. It contains an anonymous function which in this example simply alerts a message containing the button's `id` attribute.

A wide range of different events can be handled in this way, and it is a perfect system for reacting to events which are only triggered by small number of elements. To extend the example slightly, add the following code to the existing `<script>` tag, ensuring that it is within the outer anonymous function executed when the page is ready:

```
$("#button1").keypress(function() {  
    alert("You used the keyboard to interact with " + $(this).attr("id"));  
});
```

This time we bind our anonymous function to the `keypress` event using the `.keypress()` method. For information, the `keypress` event is a combination of both `keydown` and `keyup` events on the same key. With the button focused (we could add an event handler for this too!), pressing a key on the keyboard will produce the alert.

At this stage, we don't know which button was used to interact with the button, but an important feature of the event model created by jQuery is that the anonymous function is automatically passed an event object which contains various details about the event. We can examine this object to find out which key was pressed. Alter the function so that it appears as follows:

```
$("#button1").keypress(function(e) {  
    var code = e.which;  
    var key = String.fromCharCode(code);  
    alert("You used the + key + " key to interact with " + $(this).attr("id"));  
});
```

The `which` property of the event object contains the Unicode representation of the key that was used on the button. The standard JavaScript flavour `String.fromCharCode()` method translates the Unicode into a letter representing the key that was pressed. Not all of the keyboard keys are represented in this way; some don't work at all and others, like the space bar or enter key for example, don't have a string equivalent, so if these keys are used the `key` variable, and the space it is added to the alert, will be empty.

The event object can also contain other properties depending on the event it represents; click events, for example, contain the `.pageX` and `.pageY` properties so that the position of the cursor on the page when the click occurs can be determined. We could easily code to handle either the space bar or enter key being pressed using the following conditional `if` statement instead of the `key` variable declaration:

```
if (code == "32") {  
    var key = "space";  
} else if (code == "13") {  
    var key = "enter";  
} else {  
    var key = String.fromCharCode(code);  
}
```

One thing to note at this stage is known as the default browser action; some events trigger a default browser action that is common across browsers; clicking a link for example, will navigate the user to a new page. The default browser action is fine in most situations, although there may be times when we wish to cancel it.

You may have noticed that when you use the enter key to interact with the button in our example, it generates both the `keydown` and `click` alert messages. This is because the default browser action when using the enter key to press a button is to simulate a click event.

To cancel the default browser action, all we need to do is add `return false;` before the final closing bracket of the `keydown` anonymous function. Once this has been done, the enter key will only generate one alert message in our example.

Additional Event Handling Methods

Our example code is fine in the context it is currently in, although if we had twenty buttons on the page instead of just one, it would not be very efficient to add an Event helper for each button. In this situation, we could use the jQuery `.bind()` method instead to share an event handler between elements. Add some more buttons to the page, ten or so should do. Once that has been done, remove the existing Event Helper methods and replace them with the following code:

```
$("#button").bind("click", function() {  
    alert("You clicked " + $(this).attr("id"));  
});
```

The `.bind()` method takes a minimum of two arguments, the first is the event to bind to the selected elements and the second argument in this example is the anonymous function to be executed when the bound event is detected. Sometimes it is not possible to use an anonymous function when binding events, in these situations the `.bind()` method can easily be extended so that the function is defined elsewhere and called using an argument. Data can be passed to the function using the optional second argument of the method, and the function call then becomes the third argument.

The `.one()` method is almost exactly the same as the `.bind()` method, except for the fact that the `.one()` method will only respond to the bound event once. Other than that, it is used in exactly the same way and can be used with the same arguments as `.bind()`.

If you want to remove a bound event handler, all you need to do is called the `.unbind()` method. It can be used to unbind all event listeners for a matched element, or using one of its optional arguments, a specific event can be unbound.

Custom Events

The `.bind()` method can also be useful when working with custom events that you yourself have defined. Using the `.bind()` method in conjunction with the `.trigger()` or `.triggerHandler()` methods ensures that your custom events are listened for and reacted to in the same way as standard events. Let's create a new `onTenClicks` event that will be triggered after ten clicks of any button on our example page. Alter your code so that it appears as follows:

```
$("#button").bind("onTenClicks", function(e){  
    alert("Ten clicks detected!");  
});  
var clicks = 1;  
$("#button").bind("click", function(){  
    if (clicks < 10) {  
        clicks++;  
    } else {  
        $("#button").triggerHandler("onTenClicks");  
    }  
});
```

We use a simple control variable to count the number of click events that occur, which is handled in the normal way using the second `.bind()` method. On each click event, the `clicks` variable is incremented by one. When the control variable reaches 10, we use the `.triggerHandler()` method to trigger our custom `ontenClicks` event, which then generates the alert.

Interaction Helpers

As well as the Event Helpers we looked at earlier, another extremely useful feature of the events component of jQuery are the Interaction Helpers; there are two of them at present - the `.hover()` method and the `.toggle()` method. Both interaction Helpers specify two functions to be executed; the `.hover()` method allows you to define anonymous functions to be executed on `mouseover` and `mouseout` events, so both related events can be wrapped up in one method. The `.toggle()` method simply allows two functions to be executed on alternative `click` events, so clicking on something once will execute the first function, clicking the same element a second time will then trigger the second function.

Let's have a look at how the `.hover()` method can be used in place of CSS roll-overs. Create the following new page in your text editor:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
  <head>
    <title>jQuery Event Example</title>
    <script type="text/javascript" src="jquery/jquery-1.2.1.min.js"></script>
    <script type="text/javascript">
      $(function(){
        });
      </script>
    <style>
      #faces {
        position:absolute;
        top:100px;
        left:100px;
      }
      #speeches {
        position:absolute;
        left:50px;
      }
      #speech1 {
        display:none;
      }
      #speech2 {
        position:absolute;
        left:220px;
        display:none;
      }
    </style>
  </head>
  <body>
    <div id="speeches">
      
      
    </div>
    <div id="faces">
```

```


</div>
</body>
</html>
```

So we have a few images on the page, which have been positioned with some CSS. A couple of images have also been hidden at this point. We also have the `$(function() {})` handler ready to be filled with code. The following methods will complete the example:

```
$("#face1").hover(
function() {
$("#speech1").css({display:"block"});
},
function() {
$("#speech1").css({display:"none"});
}
);
```

When you roll over the element with an id of 'face1', the first anonymous function within the `.hover()` method is executed, showing the image that has an id of 'speech1'. When the mouse rolls back off of the image, the second method is executed, hiding the image once more. We can do the same for the second face as well:

```
$("#face2").hover(
function() {
$("#speech2").css({display:"block"});
},
function() {
$("#speech2").css({display:"none"});
}
);
```

Figure 1 below shows how the page should roughly look:

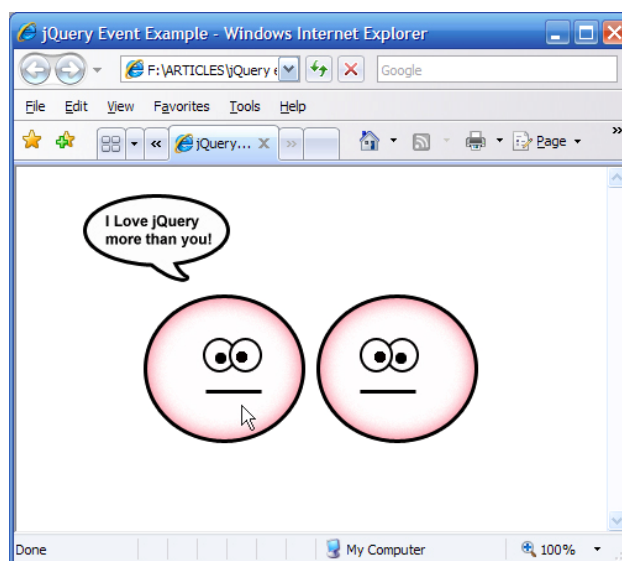


Figure 1 – the Arguing Faces

The `.toggle()` method I mentioned earlier is coded in exactly the same way; we could use this method instead of the `.hover()` method in the previous example and it would be exactly the same except that the speech bubbles would appear on clicks instead of hovers.

Summary

Events are the cornerstone of any web 2.0 application and allow you to react to different interactions that take place between your visitors and your application. jQuery makes this task easier for you, and presents a unified event model in which each browser works with events in the same way and has equal access to the event object where it is needed.

We can set and remove event listeners with the `.bind()` and `.unbind()` methods, react to almost any event with the Event Helpers, trigger events and custom events with `.trigger()` and `.triggerHandler()`, and produce interesting hover and click functions with the `.hover()` and `.toggle()` methods.

This brings us to the end of part 2 of the introduction to jQuery series. Over the course of this part we have looked in detail at the event component of the jQuery library and investigated how each of its methods can be used. I hope to see you when this series continues with Part 3, when we will be looking in detail at the stunning and ultra-useful CSS and Effects components of jQuery.