Catching Up with addEvent()

By Ryan Campbell

For most developers, myself included, `addEvent()` remains in their library because it is a widely recommended cross browser solution that "just works." And while it's usually a safe bet to follow the coding practices of those you admire, it's always worth questioning the origin and reasons for the existence of a function like addEvent() to find out where (if anywhere) something could be improved.

Due to the increasing popularity of JavaScript, developers have been searching for methods that efficiently separate structure (XHTML) from behavior (Client Side Scripting). Since this practice eliminates obtrusive code, it becomes necessary to find a way to attach events, such as onclick(), to elements in the Document Object Model. In 2001, a widely accepted solution was proposed by Scott Andrew LePera, and has been use by thousands of sites across the Internet. However, LePera's addEvent() was neither the beginning of event handling, nor does it seem to be the end.

Identifying The Problem

We'll start our journey with a review of the basic form of event handling. Event handling is the process of adding events to elements, and it can be done inline or unobtrusively. An inline event handler looks something like this:

```
<a href="somepage.html" onclick="executeFunction()">Click Me!</a>
```

When the user clicks on the link, the onclick event fires, which triggers executeFunction(). And we could create the same effect unobtrusively by writing only in the JavaScript:

```
element.onclick = executeFunction;
```

Basically, when the element is clicked by the user, executeFunction() will fire. This approach works fine, except for one major drawback—each element may have only one event attached to it. So, if you wrote the following script:

```
element.onclick = executeFunction1;
element.onclick = executeFunction2;
```

Both functions would not be executed in this scenario. In fact, the last function attached to the element is the one executed, and any

Event Registration :
For a more detailed explanation of why events override one another, I'll refer you to Peter-Paul Koch's article on the Traditional Event Registration Model.

previous assignments are ignored, which becomes particularly inconvenient for situations where programmers want to execute multiple functions at window.onload().

If you have two separate script files (a common practice with unobtrusive JavaScript), one of those files would overwrite the actions of the other. This is obviously unacceptable, since the inability to run multiple functions on an event prevents the developer from creating resuable code files—one of the major reasons for separating structure from behavior in the first place.

W3C Steps In

Given the deficiency mentioned above, in addition to more issues that will be introduced later, the W3C introduced the EventTarget interface in the DOM level two specifications, which described a very useful method called addEventListener(). Using this recommendation, unobtrusive event handling transforms into the following syntax:

```
element.addEventListener("click",executeFunction, false)
```

The method takes three parameters: the event, the function to be triggered, and a boolean representing useCapture. While this recommendation was exactly what was needed by the community, it was unfortunately introduced a bit too late. Prior to this, Microsoft already noticed the problem and implemented their own method.

Implemented before the W3C DOM working group honed the standard event model, the attachEvent() method is available for every HTML element object in the Windows version of IE5 and later.

Supporting Three Event Models at Once – ADC

Microsoft's method is called attachEvent() and it usedsa similar syntax. The first parameter is the event to listen for, and the second the function to trigger when the event is successfully fired. The method can be called as shown below:

```
element.attachEvent("onclick",executeFunction);
```

While both attachEvent() and addEventListener() are similar in concept, there are three differences to note:

- attachEvent() expects the event parameter to come from the DHTML Event List in which each event is preceded by "on." With addEventListener(), this is handled oppositely, so onclick becomes click.

- The third parameter in addEventListener(), boolean useCapture, is absent in Microsoft's rendition of the function.

- Different function names for different browsers means that object detection becomes required.

addEvent() is Born

An attempt to consolidate the difficulties associated with event listeners into one simple function spawned the first version of addEvent(), written by Scott Andrew LePera. In Crossbrowser DOM Scripting: Event Handlers, he addresses the three associated problems with event listeners and creates a function that is easy to implement, handles object detection, and comes partnered with a complimentary removeEvent() function. Here's the code in full:

```
function addEvent(obj, evType, fn, useCapture){
    if (obj.addEventListener){
        obj.addEventListener(evType, fn, useCapture);
        return true;
    } else if (obj.attachEvent){
        var r = obj.attachEvent("on"+evType, fn);
        return r;
    } else {
        alert("Handler could not be attached");
    }
}
```

After checking to make sure the object exists, this function appends "on" to the event and additionally sets useCapture if it is applicable. Another benefit of this function is that it funnels event handling through one function, so any tracking or resource management can be done from within. Unfortunately, no IE support for useCapture is possible through this version.

Problems Arise

The introduction of a cross browser unobtrusive solution to an accepted problem gave addEvent() a wide range of recognition. So much recognition that it is still being passed on as the de facto solution almost five years later. While the function deserves all of the credit given to it, there is still one limitation involving the *this* keyword. The this keyword is a reference to the currently active object. An example of using this with inline JavaScript looks like this:

```
<a href="#" onclick="executeFunction(this)">Click Me!</a>

function executeFunction(ctrl) {
      ctrl.style.display = "none";
}
```

When the link is clicked, it is passing this, which can be thought of as passing itself to the function to be modified like any other DOM element. This can also be handled unobtrusively:

```
element.onclick = executeFunction;
function executeFunction() {
      this.style.display = "none";
}
```

Note that you can directly use the *this* keyword in the unobtrusive example—it does not need to be passed to the function. Peter-Paul Koch provides an informative article covering this very concept in The *this* Keyword.

So what does the this keyword have to do with addEvent()? Well, earlier this year, Peter-Paul Koch published addEvent() considered harmful, in which he presented a test case showing how Internet Explorer's attachEvent() fails to maintain the this keyword. So, if you were to execute this script:

```
addEvent(element, "click", executeFunction);
function executeFunction() {
      this.style.display = "none";
}
```

addEvent Comparisons

Original addEvent()

Scott Andrew LePera's original version of this function satisfies both browsers by performing object detection, appending "on" to event names for Internet Explorer, and sending useCapture to browsers that support it.

```
function addEvent(obj, evType, fn, useCapture){
    if (obj.addEventListener){
        obj.addEventListener(evType, fn, useCapture);
        return true;
    } else if (obj.attachEvent){
        var r = obj.attachEvent("on"+evType, fn);
        return r;
    } else {
        alert("Handler could not be attached");
    }
}
```

Contest Winner

After QuirksMode held the addEvent() recoding contest, John Resig emerged as the victor with his rendition of addEvent(), shown below. Maintains *this* keyword.

```
function addEvent( obj, type, fn )
{
   if (obj.addEventListener)
      obj.addEventListener( type, fn, false );
   else if (obj.attachEvent)
   {
      obj["e"+type+fn] = fn;
      obj[type+fn] = function() { obj["e"+type+fn]( window.event ); }
      obj.attachEvent( "on"+type, obj[type+fn] );
   }
}
```

Dean Edwards Variation

Due to a desire to perfect event listeners, Dean Edwards took a different approach from John Resig. Instead of using browsers' built-in methods, like attachEvent(), custom functions were used to manage all of the events. Comments are compliment of Dean himself.

```
function addEvent(element, type, handler) {
    // assign each event handler a unique ID
    if (!handler.$$guid) handler.$$guid = addEvent.guid++;

    // create a hash table of event types for the element
    if (!element.events) element.events = {};

    // create a hash table of event handlers for each element/event pair
    var handlers = element.events[type];

    if (!handlers) {
        handlers = element.events[type] = {};
        // store the existing event handler (if there is one)
        if (element["on" + type]) {
            handlers[0] = element["on" + type];
        }
    }

    // store the event handler in the hash table
    handlers[handler.$$guid] = handler;

    // assign a global event handler to do all the work
    element["on" + type] = handleEvent;
};
```

The display of the window object would be set to none, not that of the element being clicked, but that's if Internet Explorer is being used.

Competition Forks Efforts

After Koch's test case was analyzed and deemed a problem that required solving, the addEvent() Recoding Contest was launched with Scott Andrew LePera, Dean Edwards and Peter-Paul Koch as judges. This call to arms challenged a wide audience to produce a superior replacement to addEvent().

The competition produced two lines of thinking. The problem could be approached similar to the old addEvent(), where the keyword this is maintained, but the code still adapts to different browsers' implementations. Or one could handle management and firing events through their own custom functions, and completely ignore the methods browsers have provided to attach events.

John Resig used the first approach and won the contest. His solution, Flexible JavaScript Events, is a 15-line function similar to the original addEvent, but maintains this keyword. While Resig's submission was declared the winner, it was generally accepted that the ideal solution had not been found. In the quest to find the ideal solution, one of the judges, Dean Edwards, wrote a variation of the function using the second line of thinking – to ignore the default browser methods and create everything from scratch.

His solution solved certain problems with Resig's entry and worked in more browsers. Tino Zijdel published an excellent summary detailing the differences between the two approaches. Additionally, Tino has been working on the script with Dean and you can follow their development on Tino's and Dean's web sites.

Where To Go From Here

Right now, either script will do justice in its current form. What's nice about Resig's script is in how lightweight and elegant it is and allows developers to use something a bit more solid (since he won't be releasing any more versions in the foreseeable future). Edward's script, however, is attempting to solve the various problems associated with Resig's and offers nice functionality, but it's still a work in progress. Regardless of your choice, remember to stay educated about the implications of your event-attaching implementations.

Ryan Campbell is a Treehouse editor and develops software for Particletree Inc. He can't seem to stop running and showering, even when there's a lot of stuff to be done around the house and work.