Cross-Browser Event Handling Using Plain ole JavaScript

scriptjunkie{ }™

Juriy Zaytsev | June 7, 2010

Cross browser event handling is not a trivial task. Fortunately, most of the popular Javascript libraries abstract this process away. But what's hidden behind the scenes is a wonderful bouquet of cross-browser inconsistencies, together with workarounds needed to get event handling to work. Let's take a look at what it takes to create a robust event handling abstraction. You might find this overview useful when building your own event-handling utilities, when assessing/optimizing existing ones, or just for educational purposes.

Quite naturally, the cornerstones of event handling are functions for adding and removing event listeners to/from elements. These functions are exactly what we'll be covering today. We will not touch upon abstraction for firing events, as that subject is a bit too involved to cover properly in this article.

DOM Level 2 defines addEventListener and removeEventListener methods as part of EventTarget interface.

`EventTarget` interface is usually implemented on objects implementing Node or Window interfaces. What this means is that DOM elements, as well as window objects, receive `addEventListener/removeEventListener` methods through which event listeners can be added or removed from the corresponding objects:

`removeEventListener` is pretty straight-forward as well (note that the event handler should reference the same function as the one used when adding listener to an element — `bodyHandler` in this example):

If we lived in a perfect world, these two functions would be all that's needed to get event handling to work. Alas, that's not the case, and a bit more work is needed to get to a finish line. The main obstacle on our way is a non-standard event-handling model used in Internet Explorer. Instead of `addEventListener/removeEventListener`, MSHTML DOM (Document Object Model used in Internet Explorer) defines attachEvent and detachEvent methods, accordingly. Both of these methods accept an `eventName` string as a first argument, and an event handler function object as a second argument.

As you can see, `attachEvent` and `detachEvent` slightly differ from their standard counterparts. The name of event is always prepended with "on" while the third argument, useCapture, is missing. There are other deviations, which we can't see immediately, but which we'll be taking a closer look at later on.

Armed with this knowledge, it should now be trivial to create an abstraction of these two different models. Let's create the following wrapping methods, `addListener` and `removeListener`:

```
1.  function addListener(element, eventName, handler) {
2.    if (element.addEventListener) {
3.      element.addEventListener(eventName, handler, false);
4.    }
5.    else if (element.attachEvent) {
6.      element.attachEvent('on' + eventName, handler);
7.    }
8.    else {
9.      element['on' + eventName] = handler;
10.   }
11. }
12.
13.
14. function removeListener(element, eventName, handler) {
15.   if (element.addEventListener) {
16.     element.removeEventListener(eventName, handler, false);
17.   }
```

```
18.    else if (element.detachEvent) {
19.      element.detachEvent('on' + eventName, handler);
20.    }
21.    else {
22.      element['on' + eventName] = null;
23.    }
24.  }
```

To ensure future compatibility and interoperability, it's important to **always first test for existence of standard method** (e.g. `addEventListener`), and only then try proprietary ones (e.g. `attachEvent`).

Note that besides `addEventListener` and `attachEvent`, there's also a third branch, for when an element is missing both `addEventListener` and `attachEvent` methods. In such case, we fall back on non-standard event-handler property assignment.

Optimizing Performance

Notice how every time `addListener/removeListener` are evaluated, there's a code branching to be done. Methods are accessed on elements, and further actions are determined based on their existence. We can eliminate all this unnecessary work by defining different functions at "load time", not at "run time". For example, when declaring `addListener`, we can check for the existence of `addEventListener` or `attachEvent` on some DOM element that we have an access to at that time. One such elements is the root element in a document, which is represented as `<html>...</html>` in markup and is accessible via `document.documentElement` in the DOM. Contrary to the body element (accessible via `document.body` in DOM), `document.documentElement` exists even when document is not "ready". This obviously makes it perfect for tests that are performed before the page has finished loading. Another option for such a "test element" is any element created dynamically, such as `document.createElement('div')`.

Let's take a look at how load-time branching can be done:

```
1.  /* first, declare variables to assign functions to */
2.  var addListener, removeListener,
3.
4.
5.      /* test element */
6.      docEl = document.documentElement;
7.
8.
9.  if (docEl.addEventListener) {
10.
11.
12.    /* if `addEventListener` exists on test element, define function to use `addEventListener` */
13.    addListener = function (element, eventName, handler) {
14.      element.addEventListener(eventName, handler, false);
15.    };
16.  }
17.  else if (docEl.attachEvent) {
18.
19.
20.    /* if `attachEvent` exists on test element, define function to use `attachEvent` */
21.    addListener = function (element, eventName, handler) {
22.      element.attachEvent('on' + eventName, handler);
23.    };
24.  }
25.  else {
26.
```

```
27.
28.    /* if neither methods exists on test element, define function to fallback strategy */
29.    addListener = function (element, eventName, handler) {
30.      element['on' + eventName] = handler;
31.    };
32.  }
```

The same goes for `removeListener`; the function is defined at load time, based on presence of methods in question.

It's worth mentioning that such optimization has certain downsides. Besides increased code size, there's now a weaker, unrelated inference being used. What this means is that instead of checking for the existence of a method directly on a target element, we test for the existence of a method on a test element and **assume** that the method **exists on the target element as well**. As with any assumption, there's a risk of failure. Theoretically, `document.documentElement` can implement `addEventListener`, but another element, the one passed to `addListener`, will not have it. Practice shows that when it comes to `addEventListener/removeEventListener`, such inference is generally safe, as long as we test for the method on same type of object, for example, **element** object and not `window` or `document`.

Hardening inference

Speaking of unrelated inference, there's one little thing we can do to lower the chance of failure. We can test for the existence of a method on the window object as well. After all, `addListener` will often be passed window object (to listen to events like "load", "resize", "scroll", etc.):

Implementations that lack `addEventListener` or `attachEvent` on `window` objects will not jeopardize `addListener` anymore. From the existence of the method on `document.documentElement` we assume that the method is present on all objects implementing `Node` interface (such as DOM elements that implement `Element` interface, or documents that implement `Document` interface). And from the existence of the method on `window`, we assume that all `window` objects have this method.

Fail-safe feature testing

The way we've been testing methods on DOM elements (and other host objects, such as `window`) is by performing boolean type-conversion on them. Boolean type-conversion is what happens to a value when it's evaluated as expression in an if statement (e.g.: `if (docEl.addEventListener) { ... }`). The problem with such conversion is that it's **known to blow up in some implementations**:

In Internet Explorer, certain host objects are implemented as ActiveX objects, and their type-conversion results in errors. What's interesting is that the type of such objects (as returned by `typeof` operator) is usually "unknown". If you think this is a weird type, that's because it is. But as per specification (ECMA-262, 3rd ed.), the `typeof` operator can return any kind of value when given a host object (e.g: "foo", "bar", "number", "undefined" or even an empty string).

So how do we deal with these quirky objects? Since we don't know exactly which ones they are, the safest strategy is to **avoid type-conversion of any host object** at all. Instead, we can infer method existence or callability by inspecting its type (as returned by `typeof` operator). If type is "object" or "function", we assume that object is callable. If the type is "unknown", we assume that the object is also callable, it's just represented as an ActiveX object but could still be called.

To abstract all this type checking, some libraries use so-called `isHostMethod` (popularized by David Mark, now used in FuseJS and My Library):

We can now replace all host objects' type-conversion -based tests with safer, isHostMethod -based ones:

Implementations in which `addEventListener` or `attachEvent` methods of `document.documentElement` are "quirky" are now handled without errors.

Normalizing event handler in IE

I mentioned before that the MSHTML event model deviates from the standard one in few other areas. One such difference is in what `this` references during event handler invocation. The handler initialized via `addEventListener` is always invoked in the context of the element that the listener was attached to. In the case of `document.body.addEventListener(...)`, the event handler is called in the context of `document.body`. In case of `window.addEventListener('...')`, the event handler is called in the context of `window` object, etc.

`attachEvent`, however, behaves differently and always invokes the event handler in context of windowobject:

As a result, our `addListener`, as it stands now, is inconsistent with regards to the context of the event handler. If you're planning to use this in it, it's a good idea to eliminate this dangerous inconsistency. Let's see how this can be done:

Instead of passing the event handler directly to `attachEvent`, we pass it a wrapping function which in turn invokes the event handler in context of an element. Now `this` always references an element that the listener was attached to.

You might have noticed that not only is the event handler invoked in the context of an element, but it's also being passed a `window.event` object as a first argument — `handler.call(element, window.event)`. This is done to work around another difference in MSHTML event model; lack of an `event` object as first argument of the event handler. Whereas `addEventListener` ensures the event handler is being given an event object, Internet Explorer makes the event object accessible via the global `window.event` property. Nothing is passed to the event handler.

By explicitly passing `window.event` to the event handler via `handler.call(element, window.event)`, we make sure it always has the proper event object accessible as the first argument.

Cleaning up memory leaks in IE

While the previous solution solves the problem of thisand the eventobject, it introduces another, rather sneaky issue of memory leaks. The annoyance with memory leaks in Internet Explorer has been described at great length in other places, so we won't repeat it here. However, let's take a quick look at what exactly causes the leak in this particular case:

When `addListener` function is executed, a **circular reference is formed**. An element has a reference to an event handler, and the event handler has a reference to an element through its scope chain. The following pattern is a classical memory leak trigger and is something we need to take care of.

So how do we fix the leak? Well, one of the easiest ways is to break these circular references on page unload. The idea is simple: when the window's "unload" event fires, iterate over all existing event listeners and clean them up. By clean-up, we mean detaching event and nullifying its reference. This effectively breaks a circular reference.

Let's take a look at one possible implementation of such system:

```
1.  function wrapHandler(element, handler) {
2.    return function (e) {
3.      return handler.call(element, e || window.event);
4.    };
5.  }
6.
7.
8.  function createListener(element, eventName, handler) {
9.    return {
10.     element: element,
11.     eventName: eventName,
12.     handler: wrapHandler(element, handler)
13.   };
14. }
15.
```

```
16.
17.  function cleanupListeners() {
18.    for (var i = listenersToCleanup.length; i--; ) {
19.      var listener = listenersToCleanup[i];
20.      litener.element.detachEvent(listener.eventName, listener.handler);
21.      listenersToCleanup[i] = null;
22.    }
23.    window.detachEvent('onunload', cleanupListeners);
24.  }
25.
26.
27.  var listenersToCleanup = [ ];
28.
29.
30.  if (isHostMethod(docEl, 'addEventListener')) {
31.    /* ... */
32.  }
33.  else if (isHostMethod(docEl, 'attachEvent')) {
34.    addListener = function (element, eventName, handler) {
35.
36.
37.      var listener = createListener(element, eventName, handler);
38.      element.attachEvent('on' + eventName, listener.handler);
39.      listenersToCleanup.push(listener);
40.    };
41.
42.
43.    window.attachEvent('onunload', cleanupListeners);
44.  }
45.  else {
46.    /* ... */
47.  }
```

During declaration of `addListener`, we also attach an "unload" listener to `window`. That method goes over all listeners stored in the `listenersToCleanup` array, and invokes the `detachEvent` on each one of them, followed by nullifying of a reference.

During execution of `addListener`, we create a listener abstraction, an object which encapsulates element, eventName, and normalized handler in one place, and push that listener into a `listenersToCleanup` array for later cleanup.

Avoiding memory leaks in IE

The cleanup procedure we wrote in previous section is great at keeping memory leaks at bay. However, another sneaky problem is rearing its ugly head — **presence of "unload" listener kills page cache** — an amazing feature implemented in majority of modern browsers. Also known as bfcache, page cache is a feature that provides immediate page display when navigating via back/forward buttons, all without sending request to a server. Unfortunately, page cache is being disabled as soon as an "unload" listener is added to a `window`. The reason page cache is disabled in cases like that is due to potential document modifications performed from within an unload handler. It is no longer safe to display a cached version. Instead, the document has to be re-requested from a server.

So we can't use the unload listener since it disables page cache. And we don't want to leave circular references either since they leak memory. Then how to take care of this tricky problem?

The solution is surprisingly trivial; **do not create circular references** in the first place!

The reason we needed to clean-up in the previous snippet is due to a somewhat hard-to-find circular reference. Let's try to find that sneaky code:

Note how `addListener` attaches `listener.handler` to an element via `attachEvent`. `listener.handler` in turn, is created via `wrapHandler(element, handler)`. If we look at `wrapHandler`, it becomes clear that returned function (the one that becomes an event handler) closes over an element in question. Once again, an element has a reference to an event handler and the event handler has a reference to an element. There is a circular reference in all its glory.

So how is it possible to avoid creation of circular references? Let's take a look at one possible implementation.

The goal of the following snippet is to avoid referencing an `element` from within an event handler. This is the key to eliminating those nasty circular references. Instead of passing an element to a handler maker, we pass it a unique id by which we can then retrieve an element. And how does an element get associated with id in the first place? By being assigned a unique id in the beginning of `addListener`.

So to sum it up, the main steps of this implementation of `addListener` are:

1. Get unique id of an element
2. Create a listener that holds a wrapped (normalized) handler. The wrapped handler never receives an actual element, to avoid creation of circular reference.
3. The newly created listener is associated with the element's unique id and event type.
4. A wrapped (normalized) handler is attached to an element (via `attachEvent`).

Circular reference is successfully avoided.

```
var getUniqueId = (function () {
  if (typeof document.documentElement.uniqueID != 'undefined') {
    return function (element) {
      return element.uniqueID;
    };
  }
  var uid = 0;
  return function (element) {
    return element.__uniqueID || (element.__uniqueID = 'uniqueID__' + uid++);
  };
})();


var getElement, setElement, listeners = { };


(function () {
  var elements = { };
  getElement = function (uid) {
    return elements[uid];
  };
  setElement = function (uid, element) {
    elements[uid] = element;
  };
})();


function createListener(uid, handler) {
```

```
29.     return {
30.       handler: handler,
31.       wrappedHandler: createWrappedHandler(uid, handler)
32.     };
33.   }
34.
35.
36.   function createWrappedHandler(uid, handler) {
37.     return function (e) {
38.       handler.call(getElement(uid), e || window.event);
39.     };
40.   }
41.
42.
43.   var addListener = function (element, eventName, handler) {
44.     var uid = getUniqueId(element);
45.     setElement(uid, element);
46.
47.
48.     if (!listeners[uid]) {
49.       listeners[uid] = { };
50.     }
51.     if (!listeners[uid][eventName]) {
52.       listeners[uid][eventName] = [ ];
53.     }
54.     var listener = createListener(uid, handler);
55.     listeners[uid][eventName].push(listener);
56.     element.attachEvent('on' + eventName, listener.wrappedHandler);
57.   };
58.
59.
60.   var removeListener = function (element, eventName, handler) {
61.     var uid = getUniqueId(element), listener;
62.     if (listeners[uid] && listeners[uid][eventName]) {
63.       for (var i = 0, len = listeners[uid][eventName].length; i < len; i++) {
64.         listener = listeners[uid][eventName][i];
65.         if (listener && listener.handler === handler) {
66.           element.detachEvent('on' + eventName, listener.wrappedHandler);
67.           listeners[uid][eventName][i] = null;
68.         }
69.       }
70.     }
71.   };
```

At the moment of this writing, only very few Javascript libraries employ event handling abstractions that do not assign unload listener. FuseJS is one of them. Others work around this problem by assigning an unload listener only "in IE", where "in IE" is determined either by sniffing userAgent string or based on some kind of inference test (jQuery 1.4.2, for example, assigns unload only when `window.attachEvent` is present, and `window.addEventListener` is not). Object inference (such as the one used in jQuery) is a safer choice compared to the fragile and unpredictable userAgent string. But as any other inference, it has a chance to produce false positives. Implementations that avoid attaching unload listener in the first place are safe from any potential "failures".

Fixing DOM Lo branch

We have normalized `this` and `event` in IE branch and did it in such way as to avoid creating memory leaks. Not only did we avoid memory leaks, but also left the page cache mechanism intact. We used robust feature testing and strong inference. We optimized performance by forking methods at load time. What else is left?

Unfortunately, the third branch of our implementation, the one that serves as a fallback for when neither addEventListener nor attachEvent are available, is somewhat defunct as of now.

If you look at it closely, the problem should become obvious.

Every time `addListener` is executed, a new handler overwrites an existing one! Moreover, `removeListener`, as it stands now, completely removes the only existing handler of an element! This is, of course, absolutely inconsistent with the rest of the implementation. We need to either remove third branch, or fix it.

Let's first take a look at how we could go about removing third branch. Some of the older browsers (e.g. Netscape Navigator 4, Opera 6) lack both `addEventListener` and `attachEvent`, so for those browsers the third branch is the only option. However, since those browsers are most likely extinct by now, more relevant are mobile browsers, which are known to lack even basic DOM methods. If we don't want to make the third branch functional, we should at least prevent failures and notify user of this deficiency in the implementation.

Note how we **still define`addListener`function** but make it essentially a no-op. We mark it as "defunct" by assigning "defunct" property with the value of true. Client code can now find if `addListener` is functional or not. If method is defunct, the application can degrade gracefully.

If, on the other hand, the goal is to support browsers without `addEventListener / attachEvent`, it's a good idea to make DOM Lo -based implementation consistent.

Before we look at one such implementation, here's a summary of the course of events. When `addListener` is called, a unique id is queried for an element. We use same helper as the one from the `attachEvent` branch. Next, this id is associated with an object to hold all handlers for this element by type. Next, an event handler is added to a queue of all event handlers for this particular element and this particular type. Finally, the existing event handler is replaced with a dispatcher, whose job is to iterate over all event handlers for this particular element/type combination and execute each one of them in proper context (element) and with proper arguments (event).

```
1.  ...
2.  else {
3.
4.
5.    var createDispatcher = function (uid, eventName) {
6.      return function (e) {
7.        if (handlers[uid] && handlers[uid][eventName]) {
8.          var handlersForEvent = handlers[uid][eventName];
9.          for (var i = 0, len = handlersForEvent.length; i < len; i++) {
10.           handlersForEvent[i].call(this, e || window.event);
11.         }
12.       }
13.     };
14.   };
15.
16.   var handlers = { };
17.
18.
19.   addListener = function (element, eventName, handler) {
```

```
20.     var uid = getUniqueId(element);
21.     if (!handlers[uid]) {
22.       handlers[uid] = { };
23.     }
24.     if (!handlers[uid][eventName]) {
25.       handlers[uid][eventName] = [ ];
26.       var existingHandler = element['on' + eventName];
27.       if (existingHandler) {
28.         handlers[uid][eventName].push(existingHandler);
29.       }
30.       element['on' + eventName] = createDispatcher(uid, eventName);
31.     }
32.     handlers[uid][eventName].push(handler);
33.   };
34.
35.
36.   removeListener = function (element, eventName, handler) {
37.     var uid = getUniqueId(element);
38.     if (handlers[uid] && handlers[uid][eventName]) {
39.       var handlersForEvent = handlers[uid][eventName];
40.       for (var i = 0, len = handlersForEvent.length; i < len; i++) {
41.         if (handlersForEvent[i] === handler) {
42.           handlersForEvent.splice(i, 1);
43.         }
44.       }
45.     }
46.   };
47. }
```

DOM Lo branch is now consistent with the rest of an implementation.

What wasn't taken care of.

Even though the final implementation turned out to be quite robust and complete, there are few things we haven't taken care of. One of them is **support for the capturing phase** in Internet Explorer. Lack of support for the capturing phase in the MSHTML event model is the reason `addListener` doesn't allow us to specify the `useCapture` argument (even though the standard `addEventListener` can handle it).

Another inconsistency is **the order of event handler execution** in Internet Explorer. While DOM L2 Events module doesn't specify in which order event handlers are to be fired, most of the browsers follow FIFO (First In, First Out) order, not LIFO (Last In, First Out) as it is in MSHTML event model. The DOM Level 3 Events module (currently draft) actually specifies the order as FIFO, saying that *"all event listeners that have been registered on the current target in their order of registration"*

We also haven't made the dispatcher in DOM L0 branch foolproof against **errors in standalone handlers**. Generally, it's a good idea to ensure none of the event handlers is affected by any failures in other handlers.

When retrieving a unique ID for an element, we didn't take care of cases when element is actually not an element, but something like a `window` object.

We can take a look at solving all of these "issues" next time. For now, they are left as an exercise to a reader. Finally, here's a complete implementation of `addListener / removeListener` at only about ~150 lines of code.

Final implementation

```
(function(global){


  function areHostMethods(object) {
    var methodNames = Array.prototype.slice.call(arguments, 1),
        t, i, len = methodNames.length;


    for (i = 0; i < len; i++) {
      t = typeof object[methodNames[i]];
      if (!(/^(?:function|object|unknown)$/).test(t)) return false;
    }
    return true;
  }


  var getUniqueId = (function () {
    if (typeof document.documentElement.uniqueID !== 'undefined') {
      return function (element) {
        return element.uniqueID;
      };
    }
    var uid = 0;
    return function (element) {
      return element.__uniqueID || (element.__uniqueID = 'uniqueID__' + uid++);
    };
  })();


  var getElement, setElement;
  (function () {
    var elements = { };
    getElement = function (uid) {
      return elements[uid];
    };
    setElement = function (uid, element) {
      elements[uid] = element;
    };
  })();


  function createListener(uid, handler) {
    return {
      handler: handler,
      wrappedHandler: createWrappedHandler(uid, handler)
    };
  }


  function createWrappedHandler(uid, handler) {
    return function (e) {
```

```
52.       handler.call(getElement(uid), e || window.event);
53.     };
54.   }
55.
56.   function createDispatcher(uid, eventName) {
57.     return function (e) {
58.       if (handlers[uid] && handlers[uid][eventName]) {
59.         var handlersForEvent = handlers[uid][eventName];
60.         for (var i = 0, len = handlersForEvent.length; i < len; i++) {
61.           handlersForEvent[i].call(this, e || window.event);
62.         }
63.       }
64.     };
65.   }
66.
67.
68.   var addListener, removeListener,
69.
70.       shouldUseAddListenerRemoveListener = (
71.         areHostMethods(document.documentElement, 'addEventListener', 'removeEventListener') &&``
72.         areHostMethods(window, 'addEventListener', 'removeEventListener')),
73.
74.
75.       shouldUseAttachEventDetachEvent = (
76.         areHostMethods(document.documentElement, 'attachEvent', 'detachEvent') &&
77.         areHostMethods(window, 'attachEvent', 'detachEvent')),
78.
79.
80.       // IE branch
81.       listeners = { },
82.
83.
84.       // DOM L0 branch
85.       handlers = { };
86.
87.
88.   if (shouldUseAddListenerRemoveListener) {
89.
90.
91.     addListener = function (element, eventName, handler) {
92.       element.addEventListener(eventName, handler, false);
93.     };
94.
95.
96.     removeListener = function (element, eventName, handler) {
97.       element.removeEventListener(eventName, handler, false);
98.     };
99.
100.
101.  }
102.  else if (shouldUseAttachEventDetachEvent) {
```

```
103.
104.
105.      addListener = function (element, eventName, handler) {
106.        var uid = getUniqueId(element);
107.        setElement(uid, element);
108.
109.
110.        if (!listeners[uid]) {
111.          listeners[uid] = { };
112.        }
113.        if (!listeners[uid][eventName]) {
114.          listeners[uid][eventName] = [ ];
115.        }
116.        var listener = createListener(uid, handler);
117.        listeners[uid][eventName].push(listener);
118.        element.attachEvent('on' + eventName, listener.wrappedHandler);
119.      };
120.
121.
122.      removeListener = function (element, eventName, handler) {
123.        var uid = getUniqueId(element), listener;
124.        if (listeners[uid] && listeners[uid][eventName]) {
125.          for (var i = 0, len = listeners[uid][eventName].length; i < len; i++) {
126.            listener = listeners[uid][eventName][i];
127.            if (listener && listener.handler === handler) {
128.              element.detachEvent('on' + eventName, listener.wrappedHandler);
129.              listeners[uid][eventName][i] = null;
130.            }
131.          }
132.        }
133.      };
134.    }
135.    else {
136.
137.
138.      addListener = function (element, eventName, handler) {
139.        var uid = getUniqueId(element);
140.        if (!handlers[uid]) {
141.          handlers[uid] = { };
142.        }
143.        if (!handlers[uid][eventName]) {
144.          handlers[uid][eventName] = [ ];
145.          var existingHandler = element['on' + eventName];
146.          if (existingHandler) {
147.            handlers[uid][eventName].push(existingHandler);
148.          }
149.          element['on' + eventName] = createDispatcher(uid, eventName);
150.        }
151.        handlers[uid][eventName].push(handler);
152.      };
153.
```

```
154.
155.     removeListener = function (element, eventName, handler) {
156.       var uid = getUniqueId(element);
157.       if (handlers[uid] && handlers[uid][eventName]) {
158.         var handlersForEvent = handlers[uid][eventName];
159.         for (var i = 0, len = handlersForEvent.length; i < len; i++) {
160.           if (handlersForEvent[i] === handler) {
161.             handlersForEvent.splice(i, 1);
162.           }
163.         }
164.       }
165.     };
166.   }
167.
168.
169.   /* export as global properties */
170.   global.addListener = addListener;
171.     global.removeListener = removeListener;
172.
173.
174.  })(this);
```

About the Author

Juriy Zaytsev, otherwise known as "kangax", is a front end web developer based in New York. Most of his work involves exploring and taming various aspects of Javascript. He blogs about some of his findings at http://perfectionkills.com/. Juriy has been contributing to various projects ranging from libraries and frameworks, to articles and books.

Find Juriy on:

- Twitter - @kangax
- Juriy's Blog