Yahoo! UI Library: Event Utility

The YUI Event Utility facilitates the creation of event-driven applications in the browser by giving you a simplified interface for subscribing to DOM events and for examining properties of the browser's Event object. The Event Utility package includes the Custom Event object; Custom Events allow you to "publish" the interesting moments or events in your own code so that other components on the page can "subscribe" to those events and respond to them. The Event Utility package provides the following features:

- A flexible method of attaching an event handler to one or more elements
- Automatic deferral of handler attachment for elements that are not yet available
- Automatic scope correction, optional scope assignment
- Automatic event object browser abstraction
- The ability to include an arbitrary object to be sent to the handler with the event
- Utility methods to access event properties that require browser abstraction
- Automatic listener cleanup
- A mechanism for creating and subscribing to custom events
- The ability to execute functions as soon as a DOM element is detected

On This Page:

Quick Links:

- Examples: Explore examples of the Event Utility in action.
- API Documentation: View the full API documentation for the Event Utility.
- Release Notes: Detailed change log for the Event Utility.
- License: The YUI Library is issued under a BSD license.
- Download: Download the Event Utility as part of the full YUI Library on SourceForge.

Getting Started

To use the Event and Custom Event Utilities, include the following source files in your web page with the **script** tag:

view plain | print | ?

```
<!-- Dependency -->
<script type="text/javascript" src="http://yui.yahooapis.com/2.3.1/build/yahoo/ya
 
<!-- Event source file -->
<script type="text/javascript" src="http://yui.yahooapis.com/2.3.1/build/event/ev
```

Where these files come from: The files included using the text above will be served from Yahoo! servers; see "Serving YUI Files from Yahoo!" for important information about this service. JavaScript files are minified, meaning that comments and white space have been removed to make them more efficient to download. To use the full, commented versions or the **-debug** versions of YUI JavaScript files, please download the library distribution and host the files on your own server.

Order matters: As is the case generally with JavaScript and CSS, order matters; these files should be included in the order specified above. If you include files in the wrong order, errors may result.

The Event and Custom Event components are defined by **YAHOO.util.Event** and **YAHOO.util.CustomEvent**, respectively.

Basic Events

To attach an event handler to the DOM, simply define your event handler and pass the event handler to the Event Utility along with a reference to the event for which you want to listen and the element to which you want attach the handler:

view plain | print | ?

```
var oElement = document.getElementById("elementid");
function fnCallback(e) { alert("click"); }
YAHOO.util.Event.addListener(oElement, "click", fnCallback);
```

These lines of code:

- Declare a variable **oElement** and assign a specific element in the DOM to that variable.
- Define a callback function, **fnCallback(e)**, to handle the specified event.
- Call the **addListener** method on the **YAHOO.util.Event** object to bind an event to the DOM element. The **addListener** method requires three arguments: the element the event is bound to (**oElement**), the event to bind (**"click"**, as a string), and the callback function (**fnCallback**).

To attach an event handler by element ID, use this code:

view plain | print | ?

```
function fnCallback(e) { alert("click"); }
YAHOO.util.Event.addListener("elementid", "click", fnCallback);
```

This example is similar to the first. However, in this case we are identifying the element by its HTML ID (**"elementid"** as a string) rather than by passing in a variable pointing to the element object. The Event Utility attempts to find the DOM element by this id value; should it fail to find the element immediately, it continues to seek the element for up to 15 seconds after the page has loaded. This "automatic deferral" enables you, in many cases, to write your event attachment code directly into your script rather than separating it out in a function that runs only after the page has loaded.

To attach an event handler to multiple elements, use this code:

view plain | print | ?

```
// array can contain object references, element ids, or both
var ids = ["el1", "el2", "el3"];
function fnCallback(e) { alert(this.id); }
YAHOO.util.Event.addListener(ids, "click", fnCallback);
```

These lines:

- Declare an array of ids corresponding to the HTML ID attributes of elements on the page. The array can contain HTML IDs as strings (as shown above); it can also accept variable object references.
- Define a callback function, **fnCallback(e)**, to handle the specified event.
- Call the **addListener** method of **YAHOO.util.Event** to bind an event to the DOM element. In this case the first argument to the **addListener** method is the ids array rather than a single element or ID.

See the examples in Using Event and Using CustomEvent or the API Documentation for more details. See also the first Event Utility example for a tutorial on how to attach events using **addListener**.

Note: Developers often wonder where they can find a comprehensive list of DOM events (e.g., "click", "mousemove", etc.) that shows in which browsers each event is supported. As far as we know, no perfect list exists. Danny Goodman's *DHTML: The Definitive Reference* may have the most comprehensive information of this kind; PPK's Event Compatibility Table on quirksmode may have the best compatibility assessment online. The Event Utility does not place any constraints on the events for which you attach handlers; it will attempt to attach listeners for any event name you provide. It's your responsibility to make sure that the event you're using is one that is supported in the browsers for which you're developing.

Using Event

This section describes several common features and uses of the Event Utility. It contains these sections:

- Handler Attachment Deferral
- Automatic Scope Correction
- Automatic Event Object Browser Abstraction
- Send an Arbitrary Object to the Event Handler

Handler Attachment Deferral

If you attempt to attach a handler to an element before the page is fully loaded, the Event Utility attempts to locate the element. If the element is not available, Event periodically checks for the element until the **window.onload** event is triggered. Handler deferral only works when attaching handlers by element id; if you attempt to attach to a DOM object reference that is not yet available, the component has no way of knowing what object you are trying to access.

Automatic Scope Correction

Event handlers added with Internet Explorer's **attachEvent** method are executed in the **window** scope, so the special variable **this** in your callback references the **window** object. This is not very useful. Even more vexing is the fact that the event object in Internet Explorer does not provide a reliable way of identifying the element on which the event was registered; standards-based browsers supply this as the **currentTarget** property, but this property is not present in IE.

By default, the Event Utility automatically adjusts the execution scope so that **this** refers to the DOM element to which the event was attached, conforming to the behavior of **addEventListener** in W3C-compliant browsers. Moreover, the event subscriber can override the scope so that **this** refers to a custom object passed into the **addListener** call.

Automatic Event Object Browser Abstraction

The first parameter your callback receives when the event fires is always the actual event object. There is no need to look at **window.event**.

Send an Arbitrary Object to the Event Handler

It is common in object-oriented JavaScript development to assign a custom object's member method to listen for an event, access internal properties and execute internal methods in response. Because the event handler is (by default) executed in the scope of the element, not in the scope of the listener method's parent object, the custom object's properties are not available through the **this** property as one might expect. You can work around this in a number of ways: (1) by creating closures or (2) creating circular references between your custom object and the element.

The Event Utility enables you to pass your custom objects directly to the event handler so you don't have to use any of these (potentially leaky) methods to gain access to that custom object. Pass your custom object as the fourth parameter to the **addListener** method, and that object is passed in as the second parameter to your callback function (the first is the event object itself):

view plain | print | ?

```
function MyObj(elementId, customProp, callback) {
   this.elementId = elementId;
   this.customProp = customProp;
   this.callback = callback;
}

MyObj.prototype.addClickHandler = function() {
   YAHOO.util.Event.addListener(this.elementId, "click", this.callback, this);
};

function fnCallback1(e, obj) {
  // the execution context is the html element ("myelementid")
  alert(this.id + " click event: " + obj.customProp);
}

function fnCallback2(e, obj) {
  // the execution context is the custom object
  alert("click event: " + this.customProp);
}

var myobj = new MyObj("myelementid", "hello world", fnCallback1);
var mydata = {id: 10 };

// One way to add the handler:
myobj.addClickHandler();

// This will do the same thing:
YAHOO.util.Event.addListener("myelementid", "click", fnCallback1, myobj);

// If we pass true as the final parameter, the custom object that is passed
// is used for the execution scope (so it becomes "this" in the callback).
YAHOO.util.Event.addListener("myelementid", "click", fnCallback2, myobj, true);


// Alternatively, we can assign a completely different object to be the
// execution scope:
YAHOO.util.Event.addListener("myelementid", "click", fnCallback2, mydata, myobj)
```

Removing Events

You can remove event listeners by calling **YAHOO.util.Event.removeListener** with the same event signature that you used to create the event.

view plain | print | ?

```
YAHOO.util.Event.removeListener("myelementid", "click", fnCallback1);
```

If it is not convenient to save a reference to the original callback you used to register the event, and you know you are the only listener to the event, you can call **removeListener** without the function argument. Doing so will remove all listeners added via **addListener** for the specified element and event type.

view plain | print | ?

```
YAHOO.util.Event.removeListener("myelementid", "click");
```

YAHOO.util.Event.getListeners lets you retrieve all of the listeners that were attached to an element via **addListener**. Optionally, you can retrieve all bound listeners of a given type:

view plain | print | ?

```
// all listeners
var listeners = YAHOO.util.Event.getListeners(myelement);
for (var i=0; i<listeners.length; ++i) {
    var listener = listeners[i];
    alert( listener.type   ); // The event type
    alert( listener.fn     ); // The function to execute
    alert( listener.obj    ); // The custom object passed into addListener
    alert( listener.adjust ); // Scope correction requested, if true, listener.ol
                              // is the scope, if an object, that object is the 
}

// only click listeners
var listeners = YAHOO.util.Event.getListeners(myelement, "click");
```

YAHOO.util.Event.purgeElement lets you remove all listeners that were registered via **addListener** from an element. Optionally, a specific type of listener can be specified. In addition, The element's children can also be purged.

view plain | print | ?

```
// all listeners
YAHOO.util.Event.purgeElement(myelement);
// all listeners and recurse children
YAHOO.util.Event.purgeElement(myelement, true);
// only click listeners
YAHOO.util.Event.purgeElement(myelement, false, "click");
```

Using the onAvailable and onContentReady Methods

onAvailable lets you define a function that will execute as soon as an element is detected in the DOM. The intent is to reduce the occurrence of timing issues when rendering script and html inline. It is not meant to be used to define handlers for elements that *may* eventually be in the document; it is meant to be used to detect elements you are in the process of loading.

The argument signature for **onAvailable** is similar to that of **addListener**, omitting only the event type.

view plain | print | ?

```
<script type="text/javascript">

function TestObj(id) {
  YAHOO.util.Event.onAvailable(id, this.handleOnAvailable, this);
}

TestObj.prototype.handleOnAvailable = function(me) {
  alert(this.id + " is available");
}

var obj = new TestObj("myelementid");
</script>

<div id="myelementid">my element</div>
```

The **onContentReady** method shares an identical syntax with **onAvailable**. The material difference between the two methods is that **onContentReady** waits until both the target element and its **nextSibling** in the DOM respond to **getElementById**. This guarantees that the target element's contents will have loaded fully (excepting any dynamic content you might add later via script). If **onContentReady** never detects a **nextSibling**, it fires with the **window.load** event.

Using the onDOMReady Method

onDOMReady lets you define a function that will execute as soon as the DOM is in a usable state. The DOM is is not deemed "usable" until it is structurally complete; a number of bugs, primarily in IE, can lead to the browser crashing or failing to load the page successfully if scripts attempt to insert information into the DOM prior to the DOM being in a complete state.

DOM readiness is achieved before images have finished loading, however, so **onDOMReady** is often an excellent alternative to using the **window** object's **load** event.

view plain | print | ?

```
<script type="text/javascript">

function init() {
   YAHOO.util.Dom.setStyle("hidden_element", "visibility", "");
}
YAHOO.util.Event.onDOMReady(init);

// As with addListener, onAvailable, and onContentReady, you can pass a data obj
// YAHOO.util.Event.onDOMReady(init, data, scope);

</script>
```

Using the CustomEvent Object

The CustomEvent object enables you to define and use events not available by default in the DOM — events that are specific to and of interest in your own application. This section describes several common uses of the CustomEvent component and provides some examples. It contains these sections:

- Defining a Custom Event
- Subscribing (Listening) to a Custom Event
- Creating a Callback
- Triggering the Event

Defining a Custom Event

To define a custom event type, create a new instance of CustomEvent:

view plain | print | ?

```
// custom object
function TestObj(name) {
    this.name = name;
    // define a custom event
    this.event1 = new YAHOO.util.CustomEvent("event1", this);
}
```

The CustomEvent constructor creates a new Custom Event; it takes one required parameter and three optional parameters:

- **type** — The type of event. This string is returned to listeners that receive this event so that they know what event occurred.
- **scope** — The scope in which listener methods should fire; if you do not specify a scope here, the default scope will be the **window** object.
- **silent** — false by default. If true, the activity for this event will not be logged when in debug mode.
- **signature** — specifies the signature for the event listeners. The choices are:
  - **YAHOO.util.CustomEvent.LIST** (the default):
    - **param1:** event name
    - **param2:** array of arguments sent to fire
    - **param3:** (optional) a custom object supplied by the subscriber
  - **YAHOO.util.CustomEvent.FLAT**
    - **param1:** the first argument passed to fire. If you need to pass multiple parameters, use and array or object literal
    - **param2:** a custom object supplied by the subscriber

The event subscriber can override the scope so that **this** refers to the custom object that was passed into the **subscribe** method.

Subscribing (Listening) to a Custom Event

To subscribe to a custom event, use its **subscribe** method:

view plain | print | ?

```
// a custom consumer object that will listen to "event1"
function Consumer(name, testObj) {
    this.name = name;
    this.testObj = testObj;
    this.testObj.event1.subscribe(this.onEvent1, this);
}
```

In this example, **event1** is the Custom Event object that was created in the previous section. Use the **subscribe** method to listen to that event. The **subscribe** method takes two parameters. The first is the callback; the second is a custom object you can define (see Send an Arbitrary Object to the Event Handler, earlier in this document). When the event is triggered, the callback is called and the custom object is passed to that callback as the third argument (when using the default argument signature; when using the flat signature, the custom object is the second argument).

Creating a Callback

To create a callback for a custom event:

view plain | print | ?

```
Consumer.prototype.onEvent1 = function(type, args, me) {
    alert(" this: " + this +
          "\n this.name: " + this.name +
          "\n type: " + type +
          "\n args[0].data: " + args[0].data +
          "\n me.name: " + me.name);
}
```

In this example the type argument is the event type (**"event1"** in this case), args is an array of all of the arguments that were passed to the Custom Event's **fire** method, and **me** is the custom object we passed in when we subscribed to the event.

Triggering the Event

To trigger or **fire** a custom event:

view plain | print | ?

```
// random test data to be used as an event argument
function TestData(data) {
    this.data = data;
}

// create an instance of our test object
var t1 = new TestObj("mytestobj1");

// create the event consumer, passing in the custom
// object so that it can subscribe to the custom event
var c1 = new Consumer("mytestconsumer1", t1);

// create a data object that will be passed to the consumer when the event fires
var d1 = new TestData("mydata1");

// fire the test object's event1 event, passing the data object as a parameter
t1.event1.fire(d1);
```

In this example **t1** is the test object we created, **event1** is the CustomEvent instance and **d1** is our test data. This example produces the following output:

view plain | print | ?

```
this: [object Object]
this.name: mytestobj1
type: event1
args[0].data: mydata1
me.name: mytestconsumer1
```

YUI on Mobile: Using Event Utility with "A-Grade" Mobile Browsers

About this Section: YUI generally works well with mobile browsers that are based on A-Grade browser foundations. For example, Nokia's N-series phones, including the N95, use a browser based on Webkit — the same foundation shared by Apple's Safari browser, which is found on the iPhone. The fundamental challenges in developing for this emerging class of full, A-Grade-derived browsers on handheld devices are:

- *Screen size:* You have a much smaller canvas;
- *Input devices:* Mobile devices generally do not have mouse input, and therefore are missing some or all mouse events (like mouseover);
- *Processor power:* Mobile devices have slower processors that can more easily be saturated by JavaScript and DOM interactions — and processor usage affects things like battery life in ways that don't have analogues in desktop browsers;
- *Latency:* Most mobile devices have a much higher latency on the network than do terrestrially networked PCs; this can make pages with many script, css or other types of external files load much more slowly.

There are other considerations, many of them device/browser specific (for example, current versions of the iPhone's Safari browser do not support Flash). The goal of these sections on YUI User's Guides is to provide you some preliminary insights about how specific components perform on this emerging class of mobile devices. Although we have not done exhaustive testing, and although these browsers are revving quickly and present a moving target, our goal is to provide some early, provisional advice to help you get started as you contemplate how your YUI-based application will render in the mobile world.

More Information:

- Challenges of Interface Design for Mobile Devices - YUI Blog article by Lucas Pettinati, Yahoo! Sr. Interaction Designer.

The Event Utility works in any browser that has DOM2 event support. However, the user interaction model of mobile browsers can make certain browser events behave in a different manner than expected, or not at all.

The iPhone's touch interface supports gestures that prevent certain mouse events from working correctly. For instance, the 'mousedown' event does not fire when the user initially touches the screen over an element. It only fires once the user's finger is removed (the mousedown, mouseup, and click events all fire at this moment). This makes is difficult or impossible to provide certain DHTML interactions that rely on these events, drag and drop being the most obvious.

Since the iPhone has a touch interface, there is no mouse cursor. This means that there are no hover states for elements, and no mouseover events.

Support & Community

The YUI Library and related topics are discussed on the on the ydn-javascript mailing list.

enter email address [Join Now]

In addition, please visit the YUIBlog for updates and articles about the YUI Library written by the library's developers.

Filing Bugs & Feature Requests

The YUI Library's public bug tracking and feature request repositories are located on the YUI SourceForge project site. Before filing new feature requests or bug reports, please review our reporting guidelines.

- Guidelines for YUI feature requests and bug reports.
- Review current bug list or file a new bug.
- Review current feature requests or file a new feature request.

Event Utility Cheat Sheet:

Download full set of cheat sheets.

Event Utility Examples:

- Simple Event Handling and Processing
- Using Custom Events
- Using onAvailable, onContentReady, and onDOMReady
- Using Event Utility and Event Delegation to Improve Performance

Other YUI Examples That Make Use of the Event Utility:

- Implementing Container Keyboard Shortcuts with KeyListener (included with examples for the Container Family)

More Reading about Dom Events and the YUI Event Utility:

- Event Compatibility Tables, by PPK (quirksmode); a list of basic DOM events that breaks out their compatibility by browser version
- Event-Driven Application Design, by Christian Heilmann
- Event Delegation versus Event Handling, by Christian Heilmann
- Forget addEvent, use Yahoo!'s Event Utility, by Dustin Diaz
- Agent YUI: Mission 1 — Attaching Events (the easy way), by Klaus Komenda

YUI Event on del.icio.us: