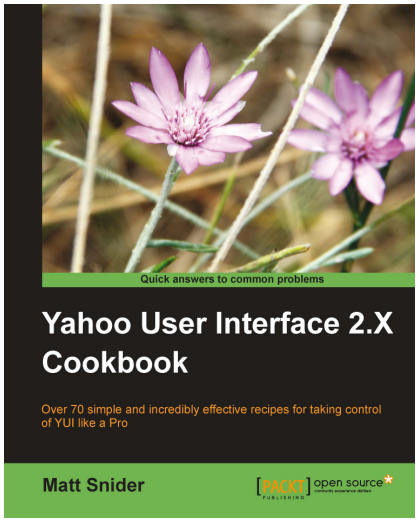
[PACKT] open source
community experience distilled
PUBLISHING

Yahoo User Interface 2.X Cookbook

Matt Snider



Chapter No. 3
"Using Event Component"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Using Event Component"

A synopsis of the book's content

Information on where to buy this book

About the Author

Matt Snider is a software engineer who has been working on the Web for over seven years. His focus is JavaScript engineering and he has contributed to both YUI 2 and 3. He was the first engineer hired at `Mint.com`, has maintained a web development blog since 2007 at `MattSnider.com`, and is currently the first engineer at `Votizen.com`. In his limited spare time he enjoys working on open source projects, playing, and collecting video games, extreme sports, and backpacking.

I greatly appreciate, my wonderful girlfriend, Janey Ly, for her support and understanding through all the late nights. Many thanks to my family, for their patience during the holidays and special occasions. Thanks to my co-workers at `Mint.com`, for their support to my YUI community contributions. And to the great engineers at YUI, for their work in building and maintaining the best JavaScript framework.

Yahoo User Interface 2.X Cookbook

Welcome to Yahoo User Interface 2.x Cookbook. The Yahoo! User Interface (YUI) Library is a set of utilities and controls, written in JavaScript, for building richly interactive web applications using techniques such as DOM scripting, DHTML, and AJAX. Although you can create stylish Internet applications by modifying its default components, even advanced users find it challenging to create impressive feature-rich Internet applications using YUI.

This book will help you learn how to use YUI 2.x to build richer, more interactive web applications that impress clients and wow your friends. It starts by explaining the core features of YUI 2.x, the utilities that the rest of the library depends on and that will make your life easier. It then explains how to build UI components and make AJAX requests using the YUI framework. Each recipe will cover the most common ways to use a component and how to configure it, and then explain any other features that may be available. We wrap things up by looking at some of the recent beta components and explain how to use them, and how they may be useful on your web application.

What This Book Covers

Chapter 1: Using YUI 2.x, teaches various techniques for including YUI into your project, and how to use the Get, Cookie, and JSON utilities.

Chapter 2: Using Dom and Selector Components, teaches how to use the DOM and Selector components to fetch and manipulate DOM elements.

Chapter 3: Using Event Component, teaches how to attach events to DOM elements and how to use the powerful custom event system to create your own events.

Chapter 4: Using Connection Component, teaches how to make AJAX requests with YUI and subscribe to related events.

Chapter 5: Using DataSource Component, teaches how to abstract away sources of data with a powerful and easy to use interface.

Chapter 6: Using Logger and YUI Test Components, teaches how to use YUI to log JavaScript and then how to test your JavaScript applications.

Chapter 7: Using Element and Button Components, teaches how to leverage Element to simplify working with DOM elements, and Button to simplify working with Button elements.

Chapter 8: Using Menu Component, teaches how to create and manage Menu objects.

Chapter 9: Using Animation Component, teaches how to use YUI to animate DOM elements.

Chapter 10: Using Drag and Drop Component, teaches how to use the Drag and Drop utility.

Chapter 11: Using Container Component, teaches all about the container stack and the various widgets that make use of it.

Chapter 12: Using DataTable Component, teaches how to use DataTable to build more useful and interactable tables.

Chapter 13: Using TreeView Component, teaches how to use TreeView to organize hierarchical data.

Chapter 14: Other Useful Components, teaches the basics about some of the other well established widgets, such as Autocomplete and Slider.

Chapter 15: Some Interesting Beta Components, teaches the basics about some of the latest widgets, such as Storage and Charts.

3
Using Event Component

In this chapter, we will cover:

- Using YUI to attach Javascript event listeners
- Event normalization functions
- Removing event listeners
- Listening for key events
- Using special YUI-only events
- Using YUI helper functions
- Using custom events
- The simple way to add custom events to classes

Introduction

In this chapter, you will learn how to use YUI to handle JavaScript events, what special events YUI has to improve the functionality of some JavaScript events, and how to write custom events for your own application.

Using YUI to attach JavaScript event listeners

When attaching events in JavaScript most browsers use the `addEventListener` function, but the developers of IE use a function called `attachEvent`. Legacy browsers do not support either function, but instead require developers to attach functions directly to element objects using the 'on' + eventName property (for example `myElement.onclick=function(){...}`). Additionally, the execution context of event callback functions varies depending on how the event listener is attached. The Event component normalizes all the cross-browser issues, fixes the execution context of the callback function, and provides additional event improvements. This recipe will show how to attach JavaScript event listeners, using YUI.

How to do it...

Attach a `click` event to an element:

```
var myElement = YAHOO.util.Dom.get('myElementId');
var fnCallback = function(e) {
  alert("myElementId was clicked");
};
YAHOO.util.Event.addListener(myElement, 'click', fnCallback);
```

Attach a `click` event to an element by its ID:

```
var fnCallback = function(e) {
  alert("myElementId was clicked");
};
YAHOO.util.Event.addListener('myElementId','click',fnCallback)
```

Attach a `click` event to several elements at once:

```
var ids = ["myElementId1", "myElementId2", "myElementId3"];
var fnCallback = function(e) {
  var targ = YAHOO.util.Event.getTarget(e);
  alert(targ.id + " was clicked");
};
YAHOO.util.Event.addListener(ids, 'click', fnCallback);
```

When attaching event listeners, you can provide an object as the optional fourth argument, to be passed through as the second argument to the callback function:

```
var myElem = YAHOO.util.Dom.get('myElementId');
var fnCallback = function(e, obj) {
  alert(obj);
};
var obj = "I was passed through.";
YAHOO.util.Event.addListener(myElem,'click',fnCallback,obj);
```

When attaching event listeners, you can change the execution context of the callback function to the fourth argument, by passing `true` as the optional fifth argument:

```
var myElement = YAHOO.util.Dom.get('myElementId');
var fnCallback = function(e) {
  alert('My execution context was changed.');
};
var ctx = {
  /* some object to be the execution context of callback */
};
YAHOO.util.Event.addListener(
  myElement, 'click', fnCallback, ctx, true);
```

How it works...

The `addListener` function wraps the native event handling functions, normalizing the cross-browser differences. When attaching events, YUI calls the correct browser specific function, or defaults to legacy event handlers. Before executing the callback function, the Event component must (in some browsers) find the event object and adjust the execution context of the callback function. The callback function is normalized by wrapping it in a closure function that executes when the browser event fires, thereby allowing YUI to correct the event, before actually executing the callback function.

In legacy browsers, which can only have one callback function per event type, YUI attaches a callback function that iterates through the listeners attached by the `addListener` function.

There's more...

The `addListener` function returns `true` if the event listener is attached successfully and `false` otherwise.

If the element to listen on is not available when the `addListener` function is called, the function will poll the DOM and wait to attach the listener when the element becomes available.

Additionally, the Event component also keeps a list of all events that it has attached. This list is maintained to simplify removing events listeners, and so that all event listeners can be removed when the end-user leaves the page.

Find all events attached to an element:

```
var listeners = YAHOO.util.Event.getListeners('myElementId');
for (var i=0,j=listeners.length; i<j; i+=1) {
    var listener = listeners[i];
    alert(listener.type); // event type
    alert(listener.fn); // callback function
    alert(listener.obj); // second argument of callback
    alert(listener.adjust); // execution context
}
```

Find all events of a certain type attached to an element:

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

The garbage collector in JavaScript does not always do a good job cleaning up event handlers. When removing nodes from the DOM, remember to remove events you may have added as well.

More on YAHOO.util.Event.addListener

The `addListener` function has been aliased by the shorter `on` function:

```
var myElement = YAHOO.util.Dom.get('myElementId');
var fnCallback = function(e) {
  alert("myElementId was clicked");
};
YAHOO.util.Event.on(myElement, 'click', fnCallback);
```

By passing an object in as the optional fifth argument of `addListener`, instead of a Boolean, you can change the execution context of the callback to that object, while still passing in an another object as the optional fourth argument:

```
var myElement = YAHOO.util.Dom.get('myElementId');
var fnCallback = function(e, obj) {
  // this executes in the context of 'ctx'
  alert(obj);
};
var obj = "I was passed through.";
var ctx = {
  /* some object to be the execution context of callback */
};
YAHOO.util.Event.addListener(
  myElement,'click',fnCallback,obj, ctx);
```

Lastly, there is an optional Boolean value that can be provided as the sixth argument of `addListener`, which causes the callback to execute in the event capture phase, instead of the event bubbling phase. You probably won't ever need to set this value to `true`, but if you want to learn more about JavaScript event phases see:

`http://www.quirksmode.org/js/events_order.html`

See also

Refer to Removing event listeners, to learn about how to remove event callbacks when they are not longer needed.

Event normalization functions

The event object, provided as the first argument of the callback function, contains a variety of values that you may need to use (such as the target element, character code, etc.). YUI provides a collection of static functions that normalizes the cross-browser variations of these values. Before trying to use these properties, you should read this recipe, as it walks you through each of those functions.

How to do it...

Fetch the normalized `target` element of an event:

```
var fnCallback = function(e) {
  var targetElement = YAHOO.util.Event.getTarget(e);
  alert(targetElement.id);
};
YAHOO.util.Event.on('myElementId', 'click', fnCallback);
```

Fetch the character code of a key event (also known as the key code):

```
var fnCallback = function(e) {
  var charCode = YAHOO.util.Event.getCharCode(e);
  alert(charCode);
};
YAHOO.util.Event.on('myElementId', 'keypress', fnCallback);
```

Fetch the x and y coordinates of a mouse event:

```
var fnCallback = function(e) {
  var x = YAHOO.util.Event.getPageX(e);
  var y = YAHOO.util.Event.getPageY(e);
  alert("x-position=" + x + " and x-position= " + y);
};
YAHOO.util.Event.on('myElementId', 'click', fnCallback);
```

Fetch both the x and y coordinates of a mouse event, using:

```
var fnCallback = function(e) {
  var point = YAHOO.util.Event.getXY(e);
  alert("x-position="+point[0]+" and x-position= "+point[1]);
};
YAHOO.util.Event.on('myElementId', 'click', fnCallback);
```

Fetch the normalized related target element of an event:

```
var fnCallback = function(e) {
  var targetElement = YAHOO.util.Event.getRelatedTarget(e);
  alert(targetElement.id);
};
YAHOO.util.Event.on('myElementId', 'click', fnCallback);
```

Fetch the normalized time of an event:

```
var fnCallback = function(e) {
  var time = YAHOO.util.Event.getTime(e);
  alert(time);
};
YAHOO.util.Event.on('myElementId', 'click', fnCallback);
```

Stop the default behavior, propagation (bubbling) of an event, or both:

```
var fnCallback = function(e) {
  // prevents the event from bubbling up to ancestors
  YAHOO.util.Event.stopPropagation(e);
  // prevents the event's default
  YAHOO.util.Event.preventDefault(e);
  // prevents the event's default behavior and bubbling
  YAHOO.util.Event.stopEvent(e);
};
YAHOO.util.Event.on('myElementId', 'click', fnCallback);
```

How it works...

All of these functions test to see if a value exists on the event for each cross-browser variation of a property. The functions then normalize those values and return them. The `stopPropogation` and `preventDefault` functions actually modify the equivalent cross-browser property of the event, and delegate the behavior to the browsers.

See also

Refer to Using YUI helper functions, for information on event delegation, which can be used to improve the performance of events.

Removing event listeners

As with attaching listeners, there are two browser variations for removing events and legacy browser considerations as well. Additionally, when using the native event removal functions, you must pass in exactly the same arguments as you passed to the native add event functions. YUI not only handles the cross-browser variations, but ensures that you pass the correct arguments when removing a function. This recipe shows several ways to remove event listeners from elements.

How to do it...

Remove a callback function directly, by passing in the exact same arguments you used when calling `addListener`:

```
YAHOO.util.Event.removeListener('myElementId', 'click',
  fnCallback);
```

Removing all events of a specific type does not require the arguments from `addListener`:

```
// removes all 'click' type events from 'myElementId'
YAHOO.util.Event.removeListener('myElementId', 'click');
```

Remove all listeners from an element:

```
YAHOO.util.Event.purgeElement('myElementId');
```

Remove all listeners from an element and its descendant elements recursively:

```
YAHOO.util.Event.purgeElement('myElementId', true);
```

Remove all events of a specific type from an element and its descendant elements recursively:

```
// removes all 'click' type events
YAHOO.util.Event.purgeElement('myElementId', true, 'click');
```

How it works...

In most cases YUI calls the native browser remove listener function, with the proper arguments from the internal array of listeners, and removes the listeners from its internal array. In legacy browsers, YUI simply deletes the function that was added to the event.

There's more...

Like the `addListener` function, the `removeListener` functions can accept either an element object or the `id` attribute of an element as its first parameter. If the element cannot be found, YUI fails silently.

To prevent memory leaks and improve site performance, YUI attaches an `unload` event listener, which removes all event listeners when the page unloads. When removing nodes from the DOM, you should call `purgeElement` on any element that might (or whose descendant elements might) have events attached to them.

Listening for key events

The native browser support for key events is identical to other events, except in addition to the other properties, the character code is attached to the event object. There are several browser variations to where the character code is stored and some value discrepancies. YUI handles these cross-browser issues, as well as adding the ability to test for key combinations (such as 'ctrl' plus another key), and a simplified interface. This recipe explains how to listen for key events, and how to leverage the more powerful YUI key management functions.

How to do it...

Listen for the `keypress` event and find the character code:

```
var fnCallback = function(e) {
  var charCode = YAHOO.util.Event.getCharCode(e);
  alert('Key pressed with character code ' + charCode);
};
YAHOO.util.Event.on('myElementId', 'keypress', fnCallback);
```

The event callback has some properties which are `true` when the *Alt*, *Control*, or *Shift* keys are pressed along with the triggering key:

```
var fnCallback = function(e) {
  if (e.altKey) {
    alert("alt key was pressed as well");
  }
  if (e.ctrlKey) {
    alert("control key was pressed as well");
  }
  if (e.shiftKey) {
    alert("shift key was pressed as well");
  }
};
YAHOO.util.Event.on('myElementId', 'keypress', fnCallback);
```

Use the `KeyListener` function to effortlessly attach key events:

```
var fnCallback = function(e) {
  alert("The enter key was pressed");
};
var keyObj = {keys:13};
var keyHandle =
  new YAHOO.util.KeyListener('myElemId', keyObj, fnCallback);
keyHandle.enable();
```

Use the `KeyListener` function to listen for several key events when *Ctrl* is also pressed:

```
var fnCallback = function(e) {
  alert("Up, Right, Down, or Left and Control was pressed");
};
var keyObj = {
  alt: false,
  ctrl: true,
  shift: false,
  keys: [37,38,39,40]
};
var keyHandle =
  new YAHOO.util.KeyListener('myElemId', keyObj, fnCallback);
keyHandle.enable();
```

The `KeyListener` function returns an object with two functions, `enable` and `disabled`, which allow you to turn the event listener on and off:

```
// turn off the key listener
keyHandle.disable();
// turn on the key listener
keyHandle.enable();
```

How it works...

When subscribing to any key event directly, YUI behaves as it does with other events, wrapping the native event and handling-cross browser variations. The `getCharCode` function checks the event object for the browser variations of character code, and normalizes the value.

The `KeyListener` function wraps a `keydown` event with an instantiatable object that checks the character codes before executing the callback. The `disable` and `enable` functions of the returned object respectively call `removeListener` and `addListener` for you. You must call the `enable` function before the instantiated `KeyListener` to work.

There's more...

The second argument of the `KeyListener` function is called the `keyData` object, and requires the `keys` property, but the Boolean values `alt`, `ctrl`, and `shift` are optional. The `keys` property can either be a single or an array of integer character codes.

By default the `KeyListener` function listens for the `keydown` event, but you can also specify the `keyup` event by passing it as a fourth optional argument:

```
var fnCallback = function(e) {
  alert("The escape key was keyed up");
};
var keyObj = {keys:27};
var KeyListener = YAHOO.util.KeyListener;
var keyHandle =
  new KeyListener('myElemId', keyObj, fnCallback, 'keyup');
keyHandle.enable();
```

Also, common key codes are available for you on a static object:

```
YAHOO.util.KeyListener.KEY = {
    ALT           : 18,
    BACK_SPACE    : 8,
    CAPS_LOCK     : 20,
    CONTROL       : 17,
    DELETE        : 46,
    DOWN          : 40,
    END           : 35,
    ENTER         : 13,
    ESCAPE        : 27,
    HOME          : 36,
    LEFT          : 37,
    META          : 224,
    NUM_LOCK      : 144,
    PAGE_DOWN     : 34,
    PAGE_UP       : 33,
    PAUSE         : 19,
    PRINTSCREEN   : 44,
    RIGHT         : 39,
    SCROLL_LOCK   : 145,
    SHIFT         : 16,
    SPACE         : 32,
    TAB           : 9,
    UP            : 38
};
```

Using special YUI only events

The Event component has two special events to bubble the `focus` and `blur` events, and two events that emulate IE's `mouseenter` and `mouseleave` events. This recipe explains how to use each of these events.

Getting ready

Use the following DOM for the examples in this recipe:

```
<div id="myElementId">
  <h3 id="myHeaderId">Example Title</h3>
  <input name="myTextInput1" type="text'/>
  <input name="myTextInput2" type="text'/>
  <input name="myTextInput3" type="text'/>
</div>
```

You will need to include the `event-mouseenter` optional component for the mouse events in this recipe:

```
<script src="pathToBuild/event-mouseenter/event-mouseenter-min.js"
type="text/javascript"></script>
```

How to do it...

Use the special `focusin` event to handle bubbling `focus` events:

```
// the focus of all 3 inputs can be handled by one callback
YAHOO.util.Event.on("myElementId", "focusin", function(e) {
  var targ = YAHOO.util.Event.getTarget(e);
  alert("focused on target: " + targ.name);
});
```

Use the special `focusout` event to handle bubbling `blur` events:

```
// the blur of all 3 inputs can be handled by one callback
YAHOO.util.Event.on("myElementId", "focusout", function(e) {
  var targ = YAHOO.util.Event.getTarget(e);
  alert("blurred on target: " + targ.name);
});
```

Use the `mouseenter` event to fire an event once when the mouse first enters an element:

```
// this callback fire once when the mouse enters the p tag
YAHOO.util.Event.on('myHeaderId', 'mouseenter', function(e){
    alert("Mouse entered: " + this.id);
});
```

Use the `mouseleave` event to fire an event once when the mouse first leaves an element:

```
// this callback fire once when the mouse enters the p tag
YAHOO.util.Event.on('myHeaderId', 'mouseleave', function(e){
    YAHOO.log("Mouse left: " + this.id);
});
```

How it works...

The `focusin` and `focusout` are special-cased events that leverage the capture phase of JavaScript events to emulate bubbling of `blur` and `focus` events.

The `mouseenter` and `mouseleave` events use IE's native event of the same name, but in other browsers YUI adds listeners to the `mouseover` and `mouseout` events of an element to emulate the behavior. YUI removes or re-attaches these events as necessary, to minimize the number of event subscribers.

Using YUI helper event functions

YUI events have four useful helpers functions: `delegate`, `onAvailable`, `onContentReady`, and `onDOMReady`. These functions augment how developers interact with the DOM. This recipe explains how they work.

Getting ready

To use the delegate function in this recipe, you will need to include the `event-delegate` and Selector components. Use the following DOM for the examples in this recipe:

```
<div id="myElementId">
    <ul>
        <li><a href="#" id="link1">Item Type One</a></li>
        <li><a href="#" id="link2">Item Type Two</a></li>
        <li><a href="#" id="link3">Item Type Three</a></li>
    </ul>
</div>
```

How to do it...

Use event delegation to have a single handler for all three anchors:

```
YAHOO.util.Event.delegate("myElementId", "click",
  function(e, matchedElement, container) {
    // The matchedElement is the default scope
    alert("Default scope id: " + this.id);
```

```
    alert("Clicked link id: " + matchedElement.id);
    /* The actual click target, which could be the matched item or a
descendant of it. */
    alert("Event target: " + YAHOO.util.Event.getTarget(e));
    //  The delegation container is passed as a third argument
    alert("Delegation container: " + container.id);
}, "li a");
```

Remove event delegation with the `removeDelegate` function:

```
YAHOO.util.Event.removeDelegate("myElementId", "click",
  fnCallback);
```

Use `onAvailable` to fire a function as soon as an element is detected in the DOM:

```
var fnCallback = function(obj) {
  alert('myElementId is now available');
};
var obj = {/* optional pass-through object */};
YAHOO.util.Event.onAvailable('myElementId', fnCallback, obj);
```

Use `onContentReady` to fire a function as soon as an element and its `nextSibling` are detected in the DOM:

```
var fnCallback = function(obj) {
  // the execution context was changed to obj
  alert('myElementId is now ready');
};
var obj = {/* optional pass-through object */};
YAHOO.util.Event.onContentReady('myElementId', fnCallback,
  obj, true);
```

Fire a callback function as soon as the DOM has finished loading:

```
var fnCallback = function(obj) {
  // the execution context was changed to window
  alert('The DOM is now available');
};
var obj = {/* optional pass-through object */};
YAHOO.util.Event.onDOMReady(fnCallback, obj, window);
```

How it works...

Event delegation works by attaching one event listener, of the specified type, to the container element (argument one). When the specified event type fires, anywhere on the container element, the DOM path from the event target to the container element is compared against the selector specified as the fourth argument of the `delegate` function. If the selector matches, then the callback function is executed.

The `onAvailable` and `onContentReady` functions poll the DOM until the element is available by `document.getElementById` or the window `load` event is fired. The `onContentReady` function only differs from `onAvailable`, in that it waits until the `nextSibling` of the element is also available to ensure that its descendants have completely loaded.

There's more...

The `onAvailable` and `onContentReady` functions have signatures exactly like the `addListener` function, except without the event type. You can use both functions after the window `load` event fires, and they will poll the DOM for up to 10 seconds.

The `onDOMReady` function also shares a signature with `addListener`, except without the element and event type. The recipes for these three functions show the variations you could use for the optional fourth and fifth arguments.

Attaching fewer events to the DOM improves the performance of your site. Use the event delegation function as much as possible to reduce the number of event listeners.

See also

Refer to Using YUI to attach JavaScript event listeners, for more information on the optional arguments for `onAvailable` and `onContentReady`.

Using custom events

YUI provides a framework for publishing and firing arbitrary events that you create. This simple feature drives much of the framework, and is the basis for most of the more complicated widgets covered in this book. This recipe shows you how to create, subscribe to, and fire custom events.

How to do it...

Create a basic custom event:

```
var myEvent = new YAHOO.util.CustomEvent('myEvent');
```

Create a custom event, whose callback has the specified execution context:

```
var ctx = {};
var myEvent = new YAHOO.util.CustomEvent('myEvent', ctx);
```

Subscribe to a custom event:

```
var myCallback = function(eventName, fireArgs, obj) {
  /* … */
};
myEvent.subscribe(myCallback);
```

Fire a custom event:

```
myEvent.fire();
```

Unsubscribe one callback from a custom event:

```
// the signature is the same as the subscribe function
if (myEvent.unsubscribe(myCallback)) {
  alert('callback unsubscribed successfully');
}
```

Unsubscribe all callbacks from a custom event:

```
myEvent.unsubscribeAll();
```

How it works...

The custom event framework maintains an array of all instantiated custom events, and an object to map the event names to the custom event object. Custom event objects contain a list of subscriber callback functions, logic to handle scope normalization, and logic for passing parameters from its constructor, `subscribe`, and `fire` functions to each subscriber function. The `unsubscribe` function removes the specified function by comparing the provided function against each subscriber function, and `unsubscribeAll` will truncate the array of subscriber functions.

There's more...

When calling the `fire` function of a custom event (and using the default callback signature), you can pass any number of arguments into the `fire` function, and they will be passed as an array to the second argument of each subscriber function.

When calling the `unsubscribe` function you should pass it the same signature as well calling the `subscribe` function to ensure the removal of the correct subscriber function.

The following subsections explain optional arguments that can be provided for the functions discussed in this recipe:

Instantiation function arguments

When instantiating a `CustomEvent` object, you need to only provide the first argument, which is the event name, but you can provide up to four additional arguments:

Argument	Explanation
type	A string value that is the unique name for this custom event.
context	Any object to be the execution context of the subscriber functions; this default to a window if no value is provided.
silent	A Boolean value indicating YUI should not log errors; by default this value is `true`.
signature	This can be one of two values that govern the function signature for the subscriber callback functions. The default value is `YAHOO.util.CustomEvent.LIST`, but you may also use `YAHOO.util.CustomEvent.FLAT`. More information is available in the *Subscriber callback function* section.
fireOnce	A Boolean value indicating the subscriber functions should only execute once, no matter how many times the `fire` function is called; this is `false` by default. Additionally, if the event has already fired, any new subscriber functions will execute immediately when they are assigned as callbacks.

```
var myEvent = YAHOO.util.CustomEvent('myEvent', this, false,   YAHOO.
util.CustomEvent.LIST, false);
```

Subscribe function

When calling the subscribe function you must pass a required callback function as the first argument, but can also pass up to two additional arguments:

Argument	Explanation
callback	The callback function. This is executed when the `fire` function is called.
obj	Any object to be passed through to the subscriber function.
context	Any object to be used as the execution context of the callback functions; this will override the execution context specified in the custom event constructor function.

```
var newContext = { /* … */ };
var obj = { /* … */ };
var myCallback = function(type, fireArgs, obj) {
  /* … */
}
myEvent.subscribe(myCallback, obj, newContext);
```

Subscriber callback functions

When the signature is `CustomEvent.LIST` (the default value) the following arguments will be passed to the subscriber callback functions when the custom event is fired:

Arguments	Explanation
type	The string name of the custom event.
fireArgs	The array of the arguments passed into the `fire` function.
obj	An object provided as the second argument when calling the `subscribe` function.

When the signature is `CustomEvent.FLAT` the following arguments will be passed:

Arguments	Explanation
arg	The first argument passed into the `fire` function. If you need multiple values, then use an array or object.
obj	An object provided as the second argument when calling the `subscribe` function.

See also

Refer to Apply EventProvider to manage custom events on objects for information on how to more effectively use custom events with objects.

The simple way to add custom events to classes

Since custom events drive much of the YUI framework, the developers have created the `YAHOO.util.EventProvider` class to simplify interacting with custom events on classes. This recipe explains how to apply `EventProvider` to your classes and leverage its subscription model.

How to do it...

Apply `EventProvider` to your class:

```
var MyClass = function() {/* … */};
MyClass.prototype = {/* … */};
YAHOO.lang.augment(MyClass, YAHOO.util.EventProvider);
var myClassInst = new MyClass();
```

Create an event on `myClassInst`:

```
myClassInst.createEvent('myEvent');
```

Subscribe to an event on `myClassInst`:

```
var myCallback = function() {/* … */};
myClassInst.subscribe('myEvent', myCallback);
```

Test if an event has been created on `myClassInst`:

```
if (myClassInst .hasEvent('myEvent')) {
  alert('myEvent has been created');
}
```

Fire an event on `myClassInst`:

```
myClassInst.fireEvent('myEvent');
```

Unsubscribe a callback from an event on `myClassInst`:

```
myClassInst.unsubscribe('myEvent', myCallback);
```

Unsubscribe from all callbacks from an event on `myClassInst`:

```
myClassInst.unsubscribeAll('myEvent');
```

Unsubscribe all events on `myClassInst`:

```
myClassInst.unsubscribeAll();
```

How it works...

When creating an event using the `EventProvider` framework, YUI prefaces the event name with a unique ID, so that custom event names are unique per object. Thus firing a custom event called `myEvent` on object A, does not execute callbacks subscribing to `myEvent` on object B. Otherwise, the functions work like their counterparts from the previous recipe, except they require you to specify the event name as the first argument. The only exception to this rule is the `fireEvent` function, which calls the subscriber functions using the `CustomEvent.FLAT` argument signature, instead of `CustomEvent.LIST`.

There's more...

The `createEvent` function will return the `CustomEvent` object that it creates or a previously existing one with the same name. The `createEvent` function instantiates the custom event object and hence you have less control over the instantiation arguments. However, you can pass a configuration object as the second optional argument of `createEvent`, with the following properties:

Property name	Explanation
scope	Any object to be the execution context of the subscriber functions; this default to a window if no value is provided.
silent	A Boolean value indicating YUI should not log errors; by default this value is `true`.
fireOnce	A Boolean value indicating the subscriber functions should only execute once, no matter how many times the `fire` function is called; this is `false` by default. Additionally, if the event has already fired, any new subscriber functions will execute immediately when they are assigned as callbacks.
onSubscribeCallback	A function that will be called each time a new subscriber is added to the event.

Any subscriber functions applied using the `subscribe` function before calling the `createEvent` function will be lost. Make sure, when instantiating classes you made, that you create all events the class may use.

See also

Using custom events, for more information on optional arguments that can be passed to the event functions.

The extending JavaScript objects, the YUI way, for more information on how to create your own extendable JavaScript objects.

Where to buy this book

You can buy Yahoo User Interface 2.X Cookbook from the Packt Publishing website: `https://www.packtpub.com/yahoo-user-interface-2-x-cookbook/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/yahoo-user-interface-2-x-cookbook/book