

5

Event Methods

The jQuery library allows us to observe user and browser behavior, and react accordingly. In this chapter, we'll closely examine each of the available event methods in turn. These methods are used to register behaviors to take effect when the user interacts with the browser, and to further manipulate those registered behaviors.



Some of the examples in this chapter use the `$.print()` function to print results to the page. This is a simple plug-in, which will be discussed in Chapter 10, *Plug-in API*.

Event handler attachment

The following methods are the building blocks of jQuery's event handling module.

.bind()

Attach a handler to an event for the elements.

```
.bind(eventType[, eventData], handler)
```

Parameters

- `eventType`: A string containing one or more JavaScript event types such as `click`, `submit`, or custom event names
- `eventData` (optional): A map of data that will be passed to the event handler
- `handler`: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

Description

The `.bind()` method is the primary means of attaching behavior to a document. All JavaScript event types such as `focus`, `mouseover`, and `resize` are allowed for `eventType`.

The jQuery library provides shortcut methods for binding the standard event types such as `.click()` for `.bind('click')`. Details of each event type can be found in the *Description* of its shortcut method.

Any string is legal for `eventType`. If the string is not the name of a native JavaScript event, then the handler is bound to a **custom event**. These events are never called by the browser, but may be triggered manually from other JavaScript code using `.trigger()` or `.triggerHandler()`.

If the `eventType` string contains a period (`.`) character, then the event is **namespaced**. The period character separates the event from its namespace. For example, in the call `.bind('click.name', handler)`, the string `click` is the event type and the string `name` is the namespace. Namespacing allows us to unbind or trigger some events of a type without affecting others. See the *Description* of `.unbind()` for more information.

When an event reaches an element, all handlers bound to that event type for the element are fired. If there are multiple handlers registered, they will always execute in the order in which they were bound. After all handlers have executed, the event continues along the normal event propagation path.



For an in-depth description of event propagation, see Chapter 3 of the book *Learning jQuery 1.3*.

A basic usage of `.bind()` is:

```
$('#foo').bind('click', function() {  
    alert('User clicked on "foo."');  
});
```

This code will cause the element with an ID of `foo` to respond to the `click` event. When a user clicks inside this element thereafter, the alert will be shown.

Event handlers

The `handler` parameter takes a callback function, as previously shown. Within the handler, the keyword `this` refers to the DOM element to which the handler is bound. To make use of the element in jQuery, it can be passed to the normal `$()` function. For example:

```
$('#foo').bind('click', function() {  
    alert($(this).text());  
});
```

After this code is executed, whenever the user clicks inside the element with an ID of `foo`, its text contents will be shown as an alert.

The event object

The handler callback function can also take parameters. When the function is called, the JavaScript event object will be passed to the first parameter.

The event object is often unnecessary and the parameter is omitted, as sufficient context is usually available when the handler is bound to know exactly what needs to be done when the handler is triggered. However, at times it becomes necessary to gather more information about the user's environment at the time the event was initiated. JavaScript provides information such as `.shiftKey` (whether the *Shift* key was held down at the time), `.offsetX` (the *x* coordinate of the mouse cursor within the element), and `.type` (to determine which kind of event this is).

Some of the event object's attributes and methods are not available on every platform. If the event is handled by a jQuery event handler, however, the library standardizes certain attributes so that they can be safely used on any browser.

The following attributes are standardized in particular:

- `.target`: This attribute represents the DOM element that initiated the event. It is often useful to compare `event.target` to `this` in order to determine if the event is being handled due to event bubbling.
- `.relatedTarget`: This attribute represents the other DOM element involved in the event, if any. For `mouseout`, it indicates the element being entered, and for `mouseover`, it indicates the element being exited.
- `.which`: For the key or button events, this attribute indicates the specific key or button that was pressed.
- `.pageX`: This attribute contains the *x* coordinate of the mouse cursor relative to the left edge of the page.
- `.pageY`: This attribute contains the *y* coordinate of the mouse cursor relative to the top edge of the page.

- `.result`: This attribute contains the last value returned by an event handler that was triggered by this event, unless the value was undefined.
- `.timeStamp`: This attribute returns the number of milliseconds since January 1, 1970 when the event is triggered. It can be useful for profiling the performance of certain jQuery functions.
- `.preventDefault()`: If this method is called, the default action of the event will not be triggered. For example, clicked anchors will not take the browser to a new URL. We can use `.isDefaultPrevented()` to determine if this method has been called by an event handler that was triggered by this event.
- `.stopPropagation()`: This method prevents the event from bubbling up the DOM tree looking for more event handlers to trigger. We can use `.isPropagationStopped()` to determine if this method has been called by an event handler that was triggered by this event.

Returning `false` from a handler is equivalent to calling both `.preventDefault()` and `.stopPropagation()` on the event object.

Using the event object in a handler looks like this:

```
$(document).ready(function() {
    $('#foo').bind('click', function(event) {
        alert('The mouse cursor is at ('
            + event.pageX + ', ' + event.pageY + ')');
    });
});
```

Note the parameter added to the anonymous function. This code will cause a click on the element with ID `foo` to report the page coordinates of the mouse cursor at the time of the click.

Passing event data


The optional `eventData` parameter is not commonly used. When provided, this argument allows us to pass additional information to the handler. One handy use of this parameter is to work around issues caused by **closures**. For example, suppose we have two event handlers that both refer to the same external variable.

```
var message = 'Spoon!';
$('#foo').bind('click', function() {
    alert(message);
});
message = 'Not in the face!';
$('#bar').bind('click', function() {
    alert(message);
});
```


As the handlers are closures having `message` in their environment, both will display the message **Not in the face!** when triggered. The variable's value has changed. To sidestep this, we can pass the message in using `eventData`.

```
var message = 'Spoon!';
$('#foo').bind('click', {msg: message}, function(event) {
    alert(event.data.msg);
});
message = 'Not in the face!';
$('#bar').bind('click', {msg: message}, function(event) {
    alert(event.data.msg);
});
```

This time the variable is not referred to directly within the handlers. Instead, the variable is passed in *by value* through `eventData`, which fixes the value at the time the event is bound. The first handler will now display **Spoon!**, while the second will alert **Not in the face!**.

 Note that objects are passed to functions *by reference*, which further complicates this scenario. An in-depth description of closures can be found in Appendix C of the book *Learning jQuery 1.3*.

If `eventData` is present, it is the second argument to the `.bind()` method. If no additional data needs to be sent to the handler, then the callback is passed as the second and final argument.

 See the `.trigger()` method reference for a way to pass data to a handler at the time the event happens rather than when the handler is bound.

.unbind()

Remove a previously-attached event handler from the elements.

```
.unbind([eventType[, handler]])
.unbind(event)
```

Parameters (first version)

- `eventType`: A string containing a JavaScript event type such as `click` or `submit`
- `handler`: The function that is to be no longer executed

Parameters (second version)

- `event`: A JavaScript event object as passed to an event handler

Return value

The jQuery object, for chaining purposes.

Description

Any handler that has been attached with `.bind()` can be removed with `.unbind()`. In the simplest case with no arguments, `.unbind()` removes all handlers attached to the elements.

```
$('#foo').unbind();
```

This version removes the handlers regardless of type. To be more precise, we can pass an event type.

```
$('#foo').unbind('click');
```

By specifying the `click` event type, only handlers for that event type will be unbound. However, this approach can still have negative ramifications if other scripts might be attaching behaviors to the same element. Robust and extensible applications typically demand the two-argument version for this reason.

```
var handler = function() {  
    alert('The quick brown fox jumps over the lazy dog.');
```

```
};  
$('#foo').bind('click', handler);  
$('#foo').unbind('click', handler);
```

By naming the handler, we can be assured that no other functions are caught in the crossfire. Note that the following will *not* work:

```
$('#foo').bind('click', function() {  
    alert('The quick brown fox jumps over the lazy dog.');
```

```
});  
  
$('#foo').unbind('click', function() {  
    alert('The quick brown fox jumps over the lazy dog.');
```

```
});
```

Even though the two functions are identical in content, they are created separately and so JavaScript is free to keep them as distinct function objects. To unbind a particular handler, we need a reference to that function and not a different one that happens to do the same thing.

Using namespaces

Instead of maintaining references to handlers in order to unbind them, we can **namespace** the events and use this capability to narrow the scope of our unbinding actions. As shown in the *Description* for the `.bind()` method, namespaces are defined by using a period (`.`) character when binding a handler.

```
$('#foo').bind('click.myEvents', handler);
```

When a handler is bound in this fashion, we can still unbind it the normal way.

```
$('#foo').unbind('click');
```

However, if we want to avoid affecting other handlers, we can be more specific.

```
$('#foo').unbind('click.myEvents');
```

If multiple namespaced handlers are bound, we can unbind them at once.

```
$('#foo').unbind('click.myEvents.yourEvents');
```

This syntax is similar to that used for CSS class selectors; they are not hierarchical. This method call is thus the same as the following:

```
$('#foo').unbind('click.yourEvents.myEvents');
```

We can also unbind all of the handlers in a namespace, regardless of event type.

```
$('#foo').unbind('.myEvents');
```

It is particularly useful to attach namespaces to event bindings when we are developing plug-ins or otherwise writing code that may interact with other event-handling code in the future.

Using the event object

The second form of the `.unbind()` method is used when we wish to unbind a handler from within itself. For example, suppose we wish to trigger an event handler only three times:

```
var timesClicked = 0;
$('#foo').bind('click', function(event) {
    alert('The quick brown fox jumps over the lazy dog.');
```

```
    timesClicked++;
    if (timesClicked >= 3) {
        $(this).unbind(event);
    }
});
```

The handler, in this case, must take a parameter so that we can capture the event object and use it to unbind the handler after the third click. The event object contains the context necessary for `.unbind()` to know which handler to remove.

This example is also an illustration of a **closure**. As the handler refers to the `timesClicked` variable, which is defined outside the function, incrementing the variable has an effect even between invocations of the handler.

.one()

Attach a handler to an event for the elements. The handler is executed at most once.

```
.one(eventType[, eventData], handler)
```

Parameters

- `eventType`: A string containing a JavaScript event type such as `click` or `submit`
- `eventData` (optional): A map of data that will be passed to the event handler
- `handler`: A function to execute at the time the event is triggered

Return value

The jQuery object, for chaining purposes.

Description

This method is identical to `.bind()`, except that the handler is unbound after its first invocation. For example:

```
$('#foo').one('click', function() {  
    alert('This will be displayed only once.');
```

After the code is executed, a click on the element with ID `foo` will display the alert. Subsequent clicks will do nothing.

This code is equivalent to the following:

```
$('#foo').bind('click', function(event) {  
    alert('This will be displayed only once.');
```

In other words, explicitly calling the `.unbind()` from within a regularly bound handler has exactly the same effect.

.trigger()

Execute all handlers and behaviors attached to the matched elements for the given event type.

```
.trigger(eventType[, extraParameters])
```

Parameters

- **eventType**: A string containing a JavaScript event type such as `click` or `submit`
- **extraParameters**: An array of additional parameters to pass along to the event handler

Return value

The jQuery object, for chaining purposes.

Description

Any event handlers attached with `.bind()` or one of its shortcut methods are triggered when the corresponding event occurs. However, they can be fired manually with the `.trigger()` method. A call to `.trigger()` executes the handlers in the same order they would be if the event were triggered naturally by the user.

```
$('#foo').bind('click', function() {  
    alert($(this).text());  
});  
$('#foo').trigger('click');
```

While `.trigger()` simulates an event activation, complete with a synthesized event object, it does not perfectly replicate a naturally occurring event.

When we define a custom event type using the `.bind()` method, the second argument to `.trigger()` can become useful. For example, suppose we have bound a handler for the custom event to our element instead of the built-in `click` event as done previously:

```
$('#foo').bind('custom', function(event, param1, param2) {  
    alert(param1 + "\n" + param2);  
});  
$('#foo').trigger('custom', ['Custom', 'Event']);
```

The event object is always passed as the first parameter to an event handler. However, if additional parameters are specified during a `.trigger()` call as they are here, these parameters will be passed along to the handler as well.

Note the difference between the extra parameters we're passing here and the `eventData` parameter to the `.bind()` method. Both are mechanisms for passing information to an event handler. However, the `extraParameters` argument to `.trigger()` allows information to be determined at the time the event is triggered, while the `eventData` argument to `.bind()` requires the information to be already computed at the time the handler is bound.

.triggerHandler()

Execute all handlers attached to an element for an event.

```
.triggerHandler(eventType[, extraParameters])
```

Parameters

- `eventType`: A string containing a JavaScript event type such as `click` or `submit`
- `extraParameters`: An array of additional parameters to pass along to the event handler

Return value

The return value of the triggered handler, or `undefined` if no handlers are triggered.

Description

The `.triggerHandler()` method behaves similarly to `.trigger()`, with the following exceptions:

- The `.triggerHandler()` method does not cause the default behavior of an event to occur (such as a form submission)
- While `.trigger()` will operate on all elements matched by the jQuery object, `.triggerHandler()` only affects the first matched element
- Events created with `.triggerHandler()` do not bubble up the DOM hierarchy; if they are not handled by the target element directly, they do nothing
- Instead of returning the jQuery object (to allow chaining), `.triggerHandler()` returns whatever value was returned by the last handler it caused to be executed

For more information on this method, see the *Description* for `.trigger()`.

.live()

Attach a handler to the event for all elements that match the current selector, now or in the future.

```
.live(eventType, handler)
```

Parameters

- `eventType`: A string containing a JavaScript event type such as `click` or `keydown`
- `handler`: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

Description

This method is a variation on the basic `.bind()` method for attaching event handlers to elements. When `.bind()` is called, the elements that the jQuery object refers to get the handler attached; however, the elements that get introduced later do not, so they would require another `.bind()` call. For instance, consider the following HTML code:

```
<body>
  <div class="clickme">
    Click here
  </div>
</body>
```

We can bind a simple click handler to this element.

```
$('.clickme').bind('click', function() {
  $.print('Bound handler called.');
```

When the element is clicked, the message gets printed. However, suppose that another element is added after this.

```
$('body').append('<div class="clickme">Another target</div>');
```

This new element also matches the selector `.clickme`, but since it was added after the call to `.bind()`, clicks on it will do nothing.

The `.live()` method provides an alternative to this behavior. Suppose we bind a click handler to the target element using this method.

```
$('.clickme').live('click', function() {  
    $.print('Live handler called.');
```

```
});
```

And then we add a new element to this.

```
$('#body').append('<div class="clickme">Another target</div>');
```

Then clicks on the new element will also trigger the handler.

Event delegation

The `.live()` method is able to affect elements that have not yet been added to the DOM through the use of **event delegation**—a handler bound to an ancestor element is responsible for events that are triggered on its descendants. The handler passed to `.live()` is never bound to an element; instead, `.live()` binds a special handler to the root of the DOM tree. In our example, when the new element is clicked, the following steps occur:

1. A click event is generated and passed to the `<div>` for handling.
2. No handler is directly bound to the `<div>`, so the event bubbles up the DOM tree.
3. The event bubbles up until it reaches the root of the tree, which is where `.live()` always binds its special handlers.
4. The special click handler bound by `.live()` executes.
5. This handler tests the `target` of the event object to see whether it should continue. This test is performed by checking if `$(event.target).closest('.clickme')` is able to locate a matching element.
6. If a matching element is found, the original handler is called on it.

As the test in step 5 is not performed until the event occurs, elements can be added at any time and still respond to events.

Caveats

The `.live()` technique is useful. However, due to its special approach, it cannot be simply substituted for `.bind()` in all cases. Specific differences include the following:

- Not all event types are supported. Only custom events and the following JavaScript events may be bound with `.live()`:
 - `click`
 - `dblclick`
 - `keydown`
 - `keypress`
 - `keyup`
 - `mousedown`
 - `mousemove`
 - `mouseout`
 - `mouseover`
 - `mouseup`
- DOM traversal methods are not fully supported for finding elements to send to `.live()`. Rather, the `.live()` method should always be called directly after a selector, as in the previous example.
- To stop further handlers from executing after one bound using `.live()`, the handler must return `false`. Calling `.stopPropagation()` will not accomplish this.

See the *Description* for `.bind()` for more information on event binding.

.die()

Remove an event handler previously attached using `.live()` from the elements.

```
.die(eventType[, handler])
```

Parameters

- `eventType`: A string containing a JavaScript event type such as `click` or `keydown`
- `handler` (optional): The function that is to be no longer executed

Return value

The jQuery object, for chaining purposes.

Description

Any handler that has been attached with `.live()` can be removed with `.die()`. This method is analogous to `.unbind()`, which is used to remove handlers attached with `.bind()`.

See the *Description* of `.live()` and `.unbind()` for further details.

Document loading

These events deal with the loading of a page into the browser.

.ready()

Specify a function to execute when the DOM is fully loaded.

```
$(document).ready(handler)
$.ready(handler)
$(handler)
```

Parameters

- `handler`: A function to execute after the DOM is ready

Return value

The jQuery object, for chaining purposes.

Description

While JavaScript provides the `load` event for executing code when a page is rendered, this event does not get triggered until all assets, such as images, have been completely received. In most cases, the script can be run as soon as the DOM hierarchy has been fully constructed. The handler passed to `.ready()` is guaranteed to be executed after the DOM is ready, so this is usually the best place to attach all other event handlers and run other jQuery code. When using scripts that rely on the value of CSS style properties, it's important to reference external stylesheets or embed style elements before referencing the scripts.

In cases where code relies on loaded assets (for example, if the dimensions of an image are required), the code should be placed in a handler for the `load` event instead.



The `.ready()` method is generally incompatible with the `<body onload="">` attribute. If `load` must be used, either do not use `.ready()` or use jQuery's `.load()` method to attach `load` event handlers to the window or to more specific items such as images.

All three syntaxes provided are equivalent. The `.ready()` method can only be called on a jQuery object matching the current document. So, the selector can be omitted.

The `.ready()` method is typically used with an anonymous function:

```
$(document).ready(function() {
    $.print('Handler for .ready() called.');
```

With this code in place, the following message is printed when the page is loaded:

Handler for .ready() called.

If `.ready()` is called after the DOM has been initialized, the new handler passed in will be executed immediately.

Aliasing the jQuery namespace

When using another JavaScript library, we may wish to call `$.noConflict()` to avoid namespace difficulties. When this function is called, the `$` shortcut is no longer available, forcing us to write `jQuery` each time we would normally write `$`. However, the handler passed to the `.ready()` method can take an argument, which is passed the global jQuery object. This means we can rename the object within the context of our `.ready()` handler without affecting other code.

```
jQuery(document).ready(function($) {
    // Code using $ as usual goes here.
});
```

.load()

Bind an event handler to the `load` JavaScript event.

```
.load(handler)
```

Parameters

- **handler:** A function to execute when the event is triggered

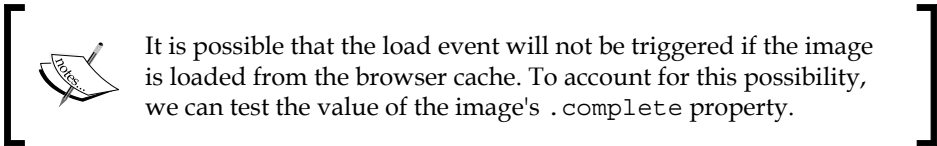
Return value

The jQuery object, for chaining purposes.

Description

This method is a shortcut for `.bind('load', handler)`.

The `load` event is sent to an element when it and all its sub-elements have been completely loaded. This event can be sent to any element associated with a URL such as images, scripts, frames, and the body of the document itself.



For example, consider a page with a simple image as follows:

```

```

The event handler can be bound to the image.

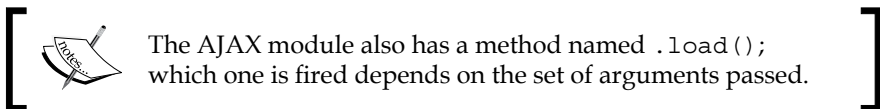
```
$('#book').load(function() {  
    $.print('Handler for .load() called.');
```

```
});
```

Now as soon as the image has been loaded, the following message is displayed:

Handler for .load() called.

In general, it is not necessary to wait for all images to be fully loaded. If code can be executed earlier, it is usually best to place it in a handler sent to the `.ready()` method.



.unload()

Bind an event handler to the unload JavaScript event.

```
.unload(handler)
```

Parameters

- **handler:** A function to execute when the event is triggered

Return value

The jQuery object, for chaining purposes.

Description

This method is a shortcut for `.bind('unload', handler)`.

The unload event is sent to the window element when the user navigates away from the page. This could mean one of many things. The user could have clicked on a link to leave the page, or typed in a new URL in the address bar. The forward and back buttons will trigger the event. Closing the browser window will cause the event to be triggered. Even a page reload will first create an unload event.



The exact handling of the unload event has varied from version to version of browsers. For example, some versions of Firefox trigger the event when a link is followed, but not when the window is closed. In practical usage, behavior should be tested on all supported browsers and contrasted with the proprietary `beforeunload` event.

Any unload event handler should be bound to the window object.

```
$(window).unload(function() {
    alert('Handler for .unload() called.');
```

```
});
```

After this code executes, the alert will be displayed whenever the browser leaves the current page.

It is not possible to cancel the unload event with `.preventDefault()`. This event is available so that scripts can perform cleanup when the user leaves the page.

.error()

Bind an event handler to the `error` JavaScript event.

```
.error(handler)
```

Parameters

- `handler`: A function to execute when the event is triggered

Return value

The jQuery object, for chaining purposes.

Description

This method is a shortcut for `.bind('error', handler)`.

The `error` event is sent to elements such as images, which are referenced by a document and loaded by the browser. It is called if the element was not loaded correctly.

For example, consider a page with a simple image as follows:

```

```

The event handler can be bound to the image.

```
$('#book').error(function() {  
    $.print('Handler for .error() called.');
```

```
});
```

If the image cannot be loaded (for example, because it is not present at the supplied URL), the following message is displayed:

Handler for .error() called.



This event may not be correctly fired when the page is served locally. As `error` relies on normal HTTP status codes, it will generally not be triggered if the URL uses the `file:` protocol.

Mouse events

These events are triggered by mouse movement and button presses.

.mousedown()

Bind an event handler to the mousedown JavaScript event, or trigger that event on an element.

```
.mousedown(handler)
.mousedown()
```

Parameters (first version)

- **handler**: A function to execute each time the event is triggered.

Return value

The jQuery object, for chaining purposes.

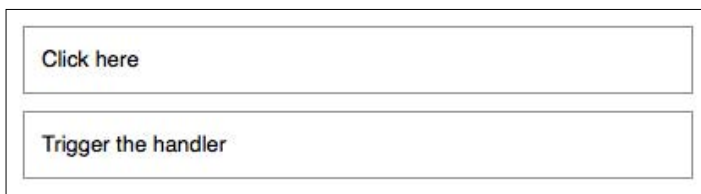
Description

This method is a shortcut for `.bind('mousedown', handler)` in the first variation and `.trigger('mousedown')` in the second.

The mousedown event is sent to an element when the mouse pointer is over the element and the mouse button is pressed. Any HTML element can receive this event.

For example, consider the following HTML code:

```
<div id="target">
  Click here
</div>
<div id="other">
  Trigger the handler
</div>
```



The event handler can be bound to any `<div>`.

```
$('#target').mousedown(function() {
  $.print('Handler for .mousedown() called.');
```

```
});
```

Now if we click on this element, the following message is displayed:

Handler for `.mousedown()` called.

We can also trigger the event when a different element is clicked.

```
$('#other').click(function() {  
    $('#target').mousedown();  
});
```

After this code executes, clicks on **Trigger the handler** will also display the same message.

The `mousedown` event is sent when any mouse button is clicked. To act only on specific buttons, we can use the event object's `.which` property. Not all browsers support this property (Internet Explorer uses `.button` instead), but jQuery normalizes the property so that it is safe to use in any browser. The value of `.which` will be 1 for the left button, 2 for the middle button, and 3 for the right button.

This event is primarily useful for ensuring that the primary button was used to begin a drag operation. If it is ignored, strange results can occur when the user attempts to use a context menu. While the middle and right buttons can be detected with these properties, this is not reliable. In Opera and Safari, for example, right mouse button clicks are not detectable by default.

If the user clicks on an element, drags away from it, and releases the button, this is still counted as a `mousedown` event. This sequence of actions is treated as a "canceling" of the button press in most user interfaces. So, it is usually better to use the `click` event unless we know that the `mousedown` event is preferable for a particular situation.

`.mouseup()`

Bind an event handler to the `mouseup` JavaScript event, or trigger that event on an element.

```
.mouseup(handler)  
.mouseup()
```

Parameters (first version)

- `handler`: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

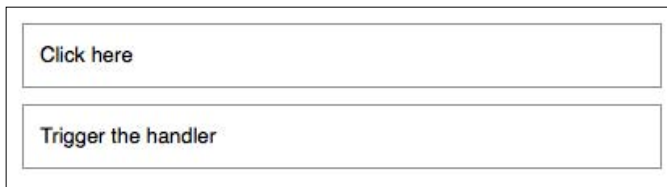
Description

This method is a shortcut for `.bind('mouseup', handler)` in the first variation, and `.trigger('mouseup')` in the second.

The `mouseup` event is sent to an element when the mouse pointer is over the element, and the mouse button is released. Any HTML element can receive this event.

For example, consider the following HTML code:

```
<div id="target">
  Click here
</div>
<div id="other">
  Trigger the handler
</div>
```



The event handler can be bound to any `<div>`.

```
$('#target').mouseup(function() {
  $.print('Handler for .mouseup() called.');
```

```
});
```

Now if we click on this element, the following message is displayed:

Handler for .mouseup() called.

We can also trigger the event when a different element is clicked.

```
$('#other').click(function() {
  $('#target').mouseup();
});
```

After this code executes, clicks on **Trigger the handler** will also display the same message.

If the user clicks outside an element, drags onto it, and releases the button, this is still counted as a `mouseup` event. This sequence of actions is not treated as a button press in most user interfaces, so it is usually better to use the `click` event unless we know that the `mouseup` event is preferable for a particular situation.

.click()

Bind an event handler to the `click` JavaScript event, or trigger that event on an element.

```
.click(handler)
.click()
```

Parameters (first version)

- `handler`: A function to execute each time the event is triggered.

Return value

The jQuery object, for chaining purposes.

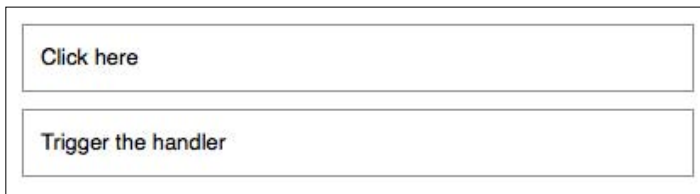
Description

This method is a shortcut for `.bind('click', handler)` in the first variation, and `.trigger('click')` in the second.

The `click` event is sent to an element when the mouse pointer is over the element and the mouse button is pressed and released. Any HTML element can receive this event.

For example, consider the following HTML code:

```
<div id="target">
  Click here
</div>
<div id="other">
  Trigger the handler
</div>
```



The event handler can be bound to any `<div>`.

```
$('#target').click(function() {
  $.print('Handler for .click() called.');
```

```
});
```

Now if we click on this element, the following message is displayed:

Handler for `.click()` called.

We can also trigger the event when a different element is clicked.

```
$('#other').click(function() {  
    $('#target').click();  
});
```

After this code executes, clicks on **Trigger the handler** will also display the same message.

The `click` event is only triggered after this exact series of events:

1. The mouse button is depressed while the pointer is inside the element.
2. The mouse button is released while the pointer is inside the element.

This is usually the desired sequence before taking an action. If this is not required, the `mousedown` or `mouseup` event may be more suitable.

`.dblclick()`

Bind an event handler to the `dblclick` JavaScript event, or trigger that event on an element.

```
.dblclick(handler)  
.dblclick()
```

Parameters (first version)

- `handler`: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

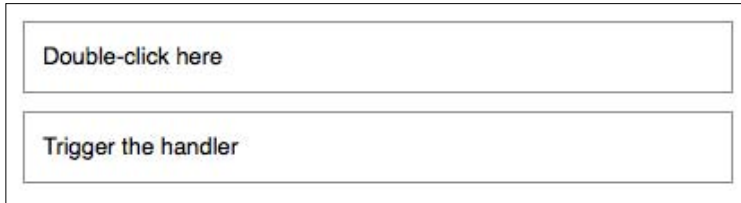
Description

This method is a shortcut for `.bind('dblclick', handler)` in the first variation, and `.trigger('dblclick')` in the second.

The `dblclick` event is sent to an element when the element is double-clicked. Any HTML element can receive this event.

For example, consider the following HTML code:

```
<div id="target">
  Double-click here
</div>
<div id="other">
  Trigger the handler
</div>
```



The event handler can be bound to any `<div>`.

```
$('#target').dblclick(function() {
  $.print('Handler for .dblclick() called.');
```

Now if we double-click on this element, the following message is displayed:

Handler for .dblclick() called.

We can also trigger the event when a different element is clicked.

```
$('#other').click(function() {
  $('#target').dblclick();
});
```

After this code executes, (single) clicks on **Trigger the handler** will also display the same message.

The `dblclick` event is only triggered after this exact series of events:

1. The mouse button is depressed while the pointer is inside the element.
2. The mouse button is released while the pointer is inside the element.
3. The mouse button is depressed again while the pointer is inside the element within a time window that is system-dependent.
4. The mouse button is released while the pointer is inside the element.

It is inadvisable to ever bind handlers to both the `click` and `dblclick` events for the same element. The sequence of events triggered varies from browser to browser with some receiving two `click` events and others only one. If an interface that reacts differently to single and double clicks cannot be avoided, then the `dblclick` event should be simulated within the `click` handler. We can achieve this by saving a timestamp in the handler, and then comparing the current time to the saved timestamp on subsequent clicks. If the difference is small enough, we can treat the click as a double-click.

.toggle()

Bind two or more handlers to the matched elements, to be executed on alternate clicks.

```
.toggle(handlerEven, handlerOdd[,  
    additionalHandlers...])
```

Parameters

- `handlerEven`: A function to execute every even time the element is clicked
- `handlerOdd`: A function to execute every odd time the element is clicked
- `additionalHandlers` (optional): Additional handlers to cycle through after clicks

Return value


The jQuery object, for chaining purposes.

Description

The `.toggle()` method binds a handler for the `click` event. So, the rules outlined for the triggering of `click` apply here as well.

For example, consider the following HTML code:

```
<div id="target">  
  Click here  
</div>
```



Event handlers can then be bound to the `<div>`.

```
$('#target').toggle(function() {  
    $.print('First handler for .toggle() called.');
```

```
    }, function() {  
        $.print('Second handler for .toggle() called.');
```

```
    });
```

As the element is clicked repeatedly, the messages alternate:

First handler for .toggle() called.

Second handler for .toggle() called.

First handler for .toggle() called.

Second handler for .toggle() called.

First handler for .toggle() called.

If more than two handlers are provided, `.toggle()` will cycle among all of them. For example, if there are three handlers, then the first handler will be called on the first click, the fourth click, the seventh click, and so on.

The `.toggle()` method is provided for convenience. It is relatively straightforward to implement the same behavior by hand, and this can be necessary if the assumptions built into `.toggle()` prove limiting. For example, `.toggle()` is not guaranteed to work correctly if applied twice to the same element. As `.toggle()` internally uses a `click` handler to do its work, we must unbind `click` to remove a behavior attached with `.toggle()` so that other `click` handlers can be caught in the crossfire. The implementation also calls `.preventDefault()` on the event. So links will not be followed and buttons will not be clicked if `.toggle()` has been called on the element.

.mouseover()

Bind an event handler to the `mouseover` JavaScript event, or trigger that event on an element.

```
.mouseover(handler)  
.mouseover()
```

Parameters (first version)

- `handler`: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

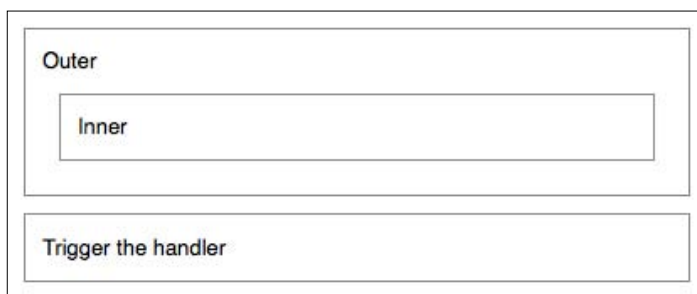
Description

This method is a shortcut for `.bind('mouseover', handler)` in the first variation, and `.trigger('mouseover')` in the second.

The `mouseover` event is sent to an element when the mouse pointer enters the element. Any HTML element can receive this event.

For example, consider the following HTML code:

```
<div id="outer">
  Outer
  <div id="inner">
    Inner
  </div>
</div>
<div id="other">
  Trigger the handler
</div>
```



The event handler can be bound to any element.

```
$('#outer').mouseover(function() {
  $.print('Handler for .mouseover() called.');
```

Now when the mouse pointer moves over the **Outer** `<div>`, the following message is displayed:

Handler for .mouseover() called.

We can also trigger the event when another element is clicked.

```
$('#other').click(function() {  
    $('#outer').mouseover();  
});
```

After this code executes, clicks on **Trigger the handler** will also display the same message.

This event type can cause many headaches due to event bubbling. For instance, when the mouse pointer moves over the **Inner** element in this example, a `mouseover` event will be sent to that, and then it will trickle up to **Outer**. This can trigger our bound `mouseover` handler at inopportune times. See the *Description* for `.mouseenter()` for a useful alternative.

.mouseout()

Bind an event handler to the `mouseout` JavaScript event, or trigger that event on an element.

```
.mouseout(handler)  
.mouseout()
```

Parameters (first version)

- `handler`: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

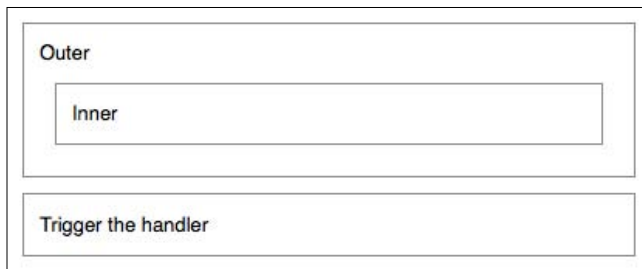
Description

This method is a shortcut for `.bind('mouseout', handler)` in the first variation, and `.trigger('mouseout')` in the second.

The `mouseout` event is sent to an element when the mouse pointer leaves the element. Any HTML element can receive this event.

For example, consider the following HTML code:

```
<div id="outer">
  Outer
  <div id="inner">
    Inner
  </div>
</div>
<div id="other">
  Trigger the handler
</div>
```



The event handler can be bound to any element.

```
$('#outer').mouseout(function() {
  $.print('Handler for .mouseout() called.');
```

Now when the mouse pointer moves out of the **Outer** <div>, the following message is displayed:

Handler for .mouseout() called.

We can also trigger the event when another element is clicked.

```
$('#other').click(function() {
  $('#outer').mouseout();
});
```

After this code executes, clicks on **Trigger the handler** will also display the same message.

This event type can cause many headaches due to event bubbling. For instance, when the mouse pointer moves out of the **Inner** element in this example, a `mouseout` event will be sent to that, and then it will trickle up to **Outer**. This can trigger our bound `mouseout` handler at inopportune times. See the *Description* for `.mouseleave()` for a useful alternative.

.mouseenter()

Bind an event handler to be fired when the mouse enters an element, or trigger that handler on an element.

```
.mouseenter(handler)
.mouseenter()
```

Parameters (first version)

- **handler**: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

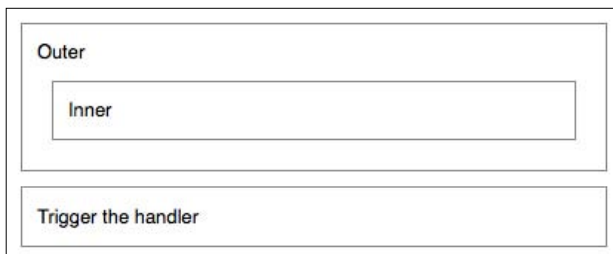
Description

This method is a shortcut for `.bind('mouseenter', handler)` in the first variation, and `.trigger('mouseenter')` in the second.

The `mouseenter` JavaScript event is proprietary to Internet Explorer. Due to the event's general utility, jQuery simulates this event so that it can be used regardless of browser. This event is sent to an element when the mouse pointer enters the element. Any HTML element can receive this event.

For example, consider the following HTML code:

```
<div id="outer">
  Outer
  <div id="inner">
    Inner
  </div>
</div>
<div id="other">
  Trigger the handler
</div>
```



The event handler can be bound to any element.

```
$('#outer').mouseenter(function() {  
    $.print('Handler for .mouseenter() called.');
```

Now when the mouse pointer moves over the **Outer** <div>, the following message is displayed:

Handler for .mouseenter() called.

We can also trigger the event when another element is clicked.

```
$('#other').click(function() {  
    $('#outer').mouseenter();  
});
```

After this code executes, clicks on **Trigger the handler** will also display the same message.

The `mouseenter` event differs from `mouseover` in the way it handles event bubbling. If `mouseover` was used in this example, then whenever the mouse pointer moved over the **Inner** element, the handler would be triggered. This is usually undesirable behavior. The `mouseenter` event, on the other hand, only triggers its handler when the mouse enters the element it is bound to (not a descendant). So in this example, the handler is triggered when the mouse enters the **Outer** element, but not the **Inner** element.

.mouseleave()

Bind an event handler to be fired when the mouse leaves an element, or trigger that handler on an element.

```
.mouseleave(handler)  
.mouseleave()
```

Parameters (first version)

- **handler**: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

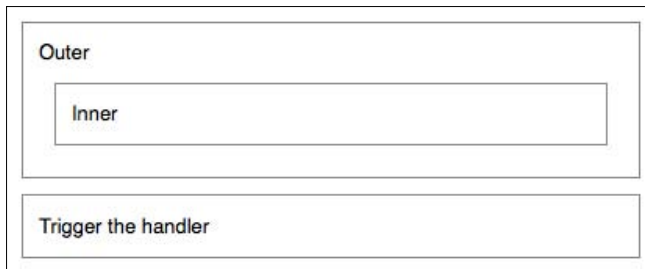
Description

This method is a shortcut for `.bind('mouseleave', handler)` in the first variation, and `.trigger('mouseleave')` in the second.

The `mouseleave` JavaScript event is proprietary to Internet Explorer. Because of the event's general utility, jQuery simulates this event so that it can be used regardless of the browser. This event is sent to an element when the mouse pointer leaves the element. Any HTML element can receive this event.

For example, consider the following HTML code:

```
<div id="outer">
  Outer
  <div id="inner">
    Inner
  </div>
</div>
<div id="other">
  Trigger the handler
</div>
```



The event handler can be bound to any element.

```
$('#outer').mouseleave(function() {
  $.print('Handler for .mouseleave() called.');
```

Now when the mouse pointer moves out of the **Outer** `<div>`, the following message is displayed:

Handler for .mouseleave() called.

We can also trigger the event when another element is clicked.

```
$('#other').click(function() {
  $('#outer').mouseleave();
});
```


After this code executes, clicks on **Trigger the handler** will also display the same message.

The `mouseleave` event differs from `mouseout` in the way it handles event bubbling. If `mouseout` were used in this example, then when the mouse pointer moved out of the **Inner** element, the handler would be triggered. This is usually undesirable behavior. The `mouseleave` event, on the other hand, only triggers its handler when the mouse leaves the element it is bound to (not a descendant). So in this example, the handler is triggered when the mouse leaves the **Outer** element, but not the **Inner** element.

.hover()

Bind two handlers to the matched elements, to be executed when the mouse pointer enters and leaves the elements.

```
.hover(handlerIn, handlerOut)
```

Parameters

- `handlerIn`: A function to execute when the mouse pointer enters the element
- `handlerOut`: A function to execute when the mouse pointer leaves the element

Return value

The jQuery object, for chaining purposes.

Description

The `.hover()` method binds handlers for both `mouseenter` and `mouseleave` events. We can use it to simply apply behavior to an element during the time the mouse is within the element.

Calling `$obj.hover(handlerIn, handlerOut)` is shorthand for the following:

```
$obj.mouseenter(handlerIn);  
$obj.mouseleave(handlerOut);
```

See the *Description* for `.mouseenter()` and `.mouseleave()` for more details.

.mousemove()

Bind an event handler to the `mousemove` JavaScript event, or trigger that event on an element.

```
.mousemove(handler)
.mousemove()
```

Parameters (first version)

- `handler`: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

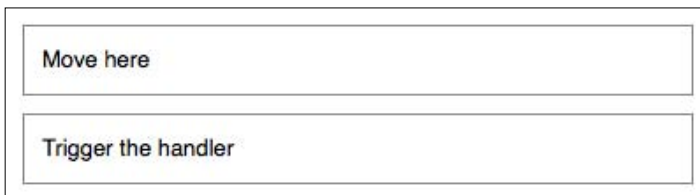
Description

This method is a shortcut for `.bind('mousemove', handler)` in the first variation, and `.trigger('mousemove')` in the second.

The `mousemove` event is sent to an element when the mouse pointer moves inside the element. Any HTML element can receive this event.

For example, consider the following HTML code:

```
<div id="target">
  Move here
</div>
<div id="other">
  Trigger the handler
</div>
```



The event handler can be bound to the target:

```
$('#target').mousemove(function(event) {
  $.print('Handler for .mousemove() called at ('
    + event.pageX + ', ' + event.pageY + ')');
});
```

Now when the mouse pointer moves within the target button, the following messages are displayed:

Handler for .mousemove() called at (399, 48)

Handler for .mousemove() called at (398, 46)

Handler for .mousemove() called at (397, 44)

Handler for .mousemove() called at (396, 42)

We can also trigger the event when the second button is clicked.

```
$('#other').click(function() {  
    $('#target').mousemove();  
});
```

After this code executes, clicks on the **Trigger the handler** button will also display the following message:

Handler for .mousemove() called at (undefined, undefined)

When tracking mouse movement, we clearly usually need to know the actual position of the mouse pointer. The event object that is passed to the handler contains some information about the mouse coordinates. Properties such as `.clientX`, `.offsetX`, and `.pageX` are available, but support for them differs between browsers. Fortunately, jQuery normalizes the `.pageX` and `.pageY` attributes so that they can be used in all browsers. These attributes provide the X and Y coordinates of the mouse pointer relative to the top-left corner of the page, as illustrated in the preceding example output.

We need to remember that the `mousemove` event is triggered whenever the mouse pointer moves, even for a pixel. This means that hundreds of events can be generated over a very small amount of time. If the handler has to do any significant processing, or if multiple handlers for the event exist, this can be a serious performance drain on the browser. It is important, therefore, to optimize `mousemove` handlers as much as possible and to unbind them as soon as they are no longer needed.

A common pattern is to bind the `mousemove` handler from within a `mousedown` handler, and to unbind it from a corresponding `mouseup` handler. If you're implementing this sequence of events, remember that the `mouseup` event might be sent to a different HTML element than the `mousemove` event was. To account for this, the `mouseup` handler should typically be bound to an element high up in the DOM tree such as `<body>`.

Form events

These events refer to `<form>` elements and their contents.

.focus()

Bind an event handler to the `focus` JavaScript event, or trigger that event on an element.

```
.focus(handler)
.focus()
```

Parameters (first version)

- `handler`: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

Description

This method is a shortcut for `.bind('focus', handler)` in the first variation, and `.trigger('focus')` in the second.

The `focus` event is sent to an element when it gains focus. This event is implicitly applicable to a limited set of elements such as form elements (`<input>`, `<select>`, and others) and links (`<a href>`). In recent browser versions, the event can be extended to include all element types by explicitly setting the element's `tabindex` property. An element can gain focus via keyboard commands such as the *Tab* key, or by mouse clicks on the element.

Elements with focus are usually highlighted in some way by the browser, for example with a dotted line surrounding the element. The focus is used to determine which element is the first to receive keyboard-related events.

For example, consider the following HTML code:

```
<form>
  <input id="target" type="text" value="Field 1" />
  <input type="text" value="Field 2" />
</form>
<div id="other">
  Trigger the handler
</div>
```

The event handler can be bound to the first input field.

```
$('#target').focus(function() {
    $.print('Handler for .focus() called.');
```

Now if we click on the first field, or tab to it from another field, the following message is displayed:

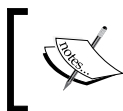
Handler for .focus() called.

We can trigger the event when another element is clicked.

```
$('#other').click(function() {
    $('#target').focus();
});
```

After this code executes, clicks on **Trigger the handler** will also display the same message.

The `focus` event does not bubble in Internet Explorer. Therefore, scripts that rely on **event delegation** with the `focus` event will not work consistently across browsers.



Triggering the focus on hidden elements causes an error in Internet Explorer. Take care to call `.focus()` without parameters only on elements that are visible.

.blur()

Bind an event handler to the `blur` JavaScript event, or trigger that event on an element.

```
.blur(handler)
.blur()
```

Parameters (first version)

- `handler`: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

Description

This method is a shortcut for `.bind('blur', handler)` in the first variation, and `.trigger('blur')` in the second.

The `blur` event is sent to an element when it loses focus. Originally, this event was only applicable to form elements such as `<input>`. In recent browsers, the domain of the event has been extended to include all element types. An element can lose focus via keyboard commands such as the *Tab* key, or by mouse clicks elsewhere on the page.

For example, consider the following HTML code:

```
<form>
  <input id="target" type="text" value="Field 1" />
  <input type="text" value="Field 2" />
</form>
<div id="other">
  Trigger the handler
</div>
```

The event handler can be bound to the first input field.

```
$('#target').blur(function() {
  $.print('Handler for .blur() called.');
```

Now if the first field has the focus and we click elsewhere, or tab away from it, the following message is displayed:

Handler for .blur() called.

We can trigger the event when another element is clicked.

```
$('#other').click(function() {
  $('#target').blur();
});
```

After this code executes, clicks on **Trigger the handler** will also display the same message.

The `blur` event does not bubble in Internet Explorer. Therefore, scripts that rely on **event delegation** with the `blur` event will not work consistently across browsers.

.change()

Bind an event handler to the change JavaScript event, or trigger that event on an element.

```
.change(handler)
.change()
```

Parameters (first version)

- **handler**: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

Description

This method is a shortcut for `.bind('change', handler)` in the first variation, and `.trigger('change')` in the second.

The change event is sent to an element when its value changes. This event is limited to `<input type="text">` fields, `<textarea>` boxes, and `<select>` elements. For select boxes, the event is fired immediately when the user makes a selection with the mouse. However, for the other element types, the event is deferred until the element loses focus.

For example, consider the following HTML code:

```
<form>
  <input class="target" type="text" value="Field 1" />
  <select class="target">
    <option value="option1" selected="selected">Option 1</option>
    <option value="option2">Option 2</option>
  </select>
</form>
<div id="other">
  Trigger the handler
</div>
```

The event handler can be bound to the text input and the select box.

```
$('.target').change(function() {
  $.print('Handler for .change() called.');
```

```
});
```

Now when the second option is selected from the dropdown, the following message is displayed:

Handler for `.change()` called.

It is also displayed if we change the text in the field and then click away. If the field loses focus without the contents having changed, though, the event is not triggered. We can trigger the event manually when another element is clicked.

```
$('#other').click(function() {  
    $('#target').change();  
});
```

After this code executes, clicks on **Trigger the handler** will also display the same message. The message will be displayed twice because the handler has been bound to the change event on both of the form elements.

The change event does not bubble in Internet Explorer. Therefore, scripts that rely on **event delegation** with the change event will not work consistently across browsers.

`.select()`

Bind an event handler to the `select` JavaScript event, or trigger that event on an element.

```
.select(handler)  
.select()
```

Parameters (first version)

- `handler`: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

Description

This method is a shortcut for `.bind('select', handler)` in the first variation, and `.trigger('select')` in the second.

The `select` event is sent to an element when the user makes a text selection inside it. This event is limited to `<input type="text">` fields and `<textarea>` boxes.

For example, consider the following HTML code:

```
<form>
  <input id="target" type="text" value="Hello there">
  Trigger the handler
</div>
```

The event handler can be bound to the text input.

```
$('#target').select(function() {
  $.print('Handler for .select() called.');
```

Now when any portion of the text is selected, the following message is displayed:

Handler for .select() called.

Merely setting the location of the insertion point will not trigger the event. We can trigger the event manually when another element is clicked.

```
$('#other').click(function() {
  $('#target').select();
});
```

After this code executes, clicks on the **Trigger the handler** button will also display the same message.

In addition, the default `select` action on the field will be fired, so the entire text field will be selected.



The method for retrieving the current selected text differs from one browser to another. For a simple cross-platform solution, use the **fieldSelection** jQuery plug-in.

.submit()

Bind an event handler to the `submit` JavaScript event, or trigger that event on an element.

```
.submit(handler)
.submit()
```

Parameters (first version)

- `handler`: A function to execute each time the event is triggered

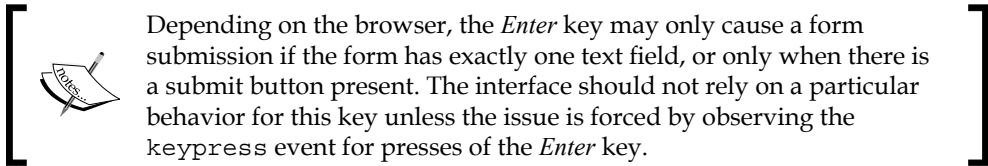
Return value

The jQuery object, for chaining purposes.

Description

This method is a shortcut for `.bind('submit', handler)` in the first variation, and `.trigger('submit')` in the second.

The `submit` event is sent to an element when the user is attempting to submit a form. It can only be attached to `<form>` elements. Forms can be submitted either by clicking an explicit `<input type="submit">` button, or by pressing *Enter* when a form element has focus.



Depending on the browser, the *Enter* key may only cause a form submission if the form has exactly one text field, or only when there is a submit button present. The interface should not rely on a particular behavior for this key unless the issue is forced by observing the `keypress` event for presses of the *Enter* key.

For example, consider the following HTML code:

```
<form id="target" action="destination.html">
  <input type="text" value="Hello there" />
  <input type="submit" value="Go" />
</form>
<div id="other">
  Trigger the handler
</div>
```

The event handler can be bound to the form.

```
$('#target').submit(function() {
  $.print('Handler for .submit() called.');
```

```
  return false;
});
```

Now when the form is submitted, the following message is displayed:

Handler for `.submit()` called.

This happens prior to the actual submission. Therefore, we can cancel the submit action by calling `.preventDefault()` on the event object or by returning `false` from our handler. We can trigger the event manually when another element is clicked.

```
$('#other').click(function() {
  $('#target').submit();
});
```

After this code executes, clicks on **Trigger the handler** will also display the same message.

In addition, the default `submit` action on the form will be fired, so the form will be submitted.

The `submit` event does not bubble in Internet Explorer. Therefore, scripts that rely on **event delegation** with the `submit` event will not work consistently across browsers.

Keyboard events

These events are triggered by the keys on the keyboard.

.keydown()

Bind an event handler to the `keydown` JavaScript event, or trigger that event on an element.

```
.keydown(handler)
.keydown()
```

Parameters (first version)

- `handler`: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

Description

This method is a shortcut for `.bind('keydown', handler)` in the first variation, and `.trigger('keydown')` in the second.

The `keydown` event is sent to an element when the user first presses a key on the keyboard. It can be attached to any element, but the event is only sent to the element that has the focus. Focusable elements can vary between browsers, but form elements can always get focus so are reasonable candidates for this event type.

For example, consider the following HTML code:

```
<form>
  <input id="target" type="text" value="Hello there" />
</form>
<div id="other">
  Trigger the handler
</div>
```

The event handler can be bound to the input field.

```
$('#target').keydown(function() {
  $.print('Handler for .keydown() called.');
```

Now when the insertion point is inside the field and a key is pressed, the following message is displayed:

Handler for .keydown() called.

We can trigger the event manually when another element is clicked.

```
$('#other').click(function() {
  $('#target').keydown();
});
```

After this code executes, clicks on **Trigger the handler** will also display the same message.

If key presses anywhere need to be caught (for example, to implement global shortcut keys on a page), it is useful to attach this behavior to the `document` object. Because of event bubbling, all key presses will make their way up the DOM to the document object unless explicitly stopped.

To determine which key was pressed, we can examine the event object that is passed to the handler function. While browsers use differing attributes to store this information, jQuery normalizes the `.which` attribute so we can reliably use it to retrieve the key code. This code corresponds to a key on the keyboard, including codes for special keys such as the arrow keys. For catching actual text entry, `.keypress()` may be a better choice.

.keypress()

Bind an event handler to the `keypress` JavaScript event, or trigger that event on an element.

```
.keypress(handler)
.keypress()
```

Parameters (first version)

- `handler`: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

Description

This method is a shortcut for `.bind('keypress', handler)` in the first variation, and `.trigger('keypress')` in the second.

The `keypress` event is sent to an element when the browser registers keyboard input. This is similar to the `keydown` event, except in the case of key repeats. If the user presses and holds a key, a `keydown` event is triggered once, but separate `keypress` events are triggered for each inserted character. In addition, modifier keys (such as *Shift*) cause `keydown` events, but not `keypress` events.

A `keypress` event handler can be attached to any element, but the event is only sent to the element that has the focus. Focusable elements can vary between browsers, but form elements can always get focus and so are reasonable candidates for this event type.

For example, consider the following HTML code:

```
<form>
  <input id="target" type="text" value="Hello there" />
</form>
<div id="other">
  Trigger the handler
</div>
```

The event handler can be bound to the input field.

```
$('#target').keypress(function() {
  $.print('Handler for .keypress() called.');
```

Now when the insertion point is inside the field and a key is pressed, the following message is displayed:

Handler for `.keypress()` called.

The message repeats if the key is held down. We can trigger the event manually when another element is clicked.

```
$('#other').click(function() {  
    $('#target').keypress();  
});
```

After this code executes, clicks on **Trigger the handler** will also display the same message.

If key presses anywhere need to be caught (for example, to implement global shortcut keys on a page), it is useful to attach this behavior to the `document` object. Because of event bubbling, all key presses will make their way up the DOM to the document object unless explicitly stopped.

To determine which character was entered, we can examine the event object that is passed to the handler function. While browsers use differing attributes to store this information, jQuery normalizes the `.which` attribute so we can reliably use it to retrieve the character code.

Note that `keydown` and `keyup` provide a code indicating which key is pressed, while `keypress` indicates which character was entered. For example, a lowercase "a" will be reported as 65 by `keydown` and `keyup`, but as 97 by `keypress`. An uppercase "A" is reported as 97 by all events. Because of this distinction, when catching special keystrokes such as arrow keys, `.keydown()` or `.keyup()` is a better choice.

.keyup()

Bind an event handler to the `keyup` JavaScript event, or trigger that event on an element.

```
.keyup(handler)  
.keyup()
```

Parameters (first version)

- `handler`: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

Description

This method is a shortcut for `.bind('keyup', handler)` in the first variation, and `.trigger('keyup')` in the second.

The `keyup` event is sent to an element when the user releases a key on the keyboard. It can be attached to any element, but the event is only sent to the element that has the focus. Focusable elements can vary between browsers, but form elements can always get focus so are reasonable candidates for this event type.

For example, consider the following HTML code:

```
<form>
  <input id="target" type="text" value="Hello there" />
</form>
<div id="other">
  Trigger the handler
</div>
```

The event handler can be bound to the input field.

```
$('#target').keyup(function() {
  $.print('Handler for .keyup() called.');
```

Now when the insertion point is inside the field and a key is pressed and released, the following message is displayed:

Handler for .keyup() called.

We can trigger the event manually when another element is clicked.

```
$('#other').click(function() {
  $('#target').keyup();
});
```

After this code executes, clicks on **Trigger the handler** will also display the same message.

If key presses anywhere need to be caught (for example, to implement global shortcut keys on a page), it is useful to attach this behavior to the `document` object. Because of event bubbling, all key presses will make their way up the DOM to the `document` object unless explicitly stopped.

To determine which key was pressed, we can examine the event object that is passed to the handler function. While browsers use differing attributes to store this information, jQuery normalizes the `.which` attribute so we can reliably use it to retrieve the key code. This code corresponds to a key on the keyboard, including codes for special keys such as the arrow keys. For catching actual text entry, `.keypress()` may be a better choice.

Browser events

These are events related to the entire browser window.

.resize()

Bind an event handler to the `resize` JavaScript event, or trigger that event on an element.

```
.resize(handler)
.resize()
```

Parameters (first version)

- `handler`: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

Description

This method is a shortcut for `.bind('resize', handler)` in the first variation, and `.trigger('resize')` in the second.

The `resize` event is sent to the window element when the size of the browser window changes.

```
$(window).resize(function() {
    $.print('Handler for .resize() called.');
```

Now whenever the browser window's size is changed, the following message is displayed:

Handler for .resize() called.

Code in a `resize` handler should never rely on the number of times the handler is called. Depending on implementation, the `resize` events can be sent continuously as the resizing is in progress (the typical behavior in Internet Explorer and WebKit-based browsers such as Safari and Chrome), or only once at the end of the resize operation (the typical behavior in Firefox).

.scroll()

Bind an event handler to the `scroll` JavaScript event, or trigger that event on an element.

```
.scroll(handler)
.scroll()
```

Parameters (first version)

- `handler`: A function to execute each time the event is triggered

Return value

The jQuery object, for chaining purposes.

Description

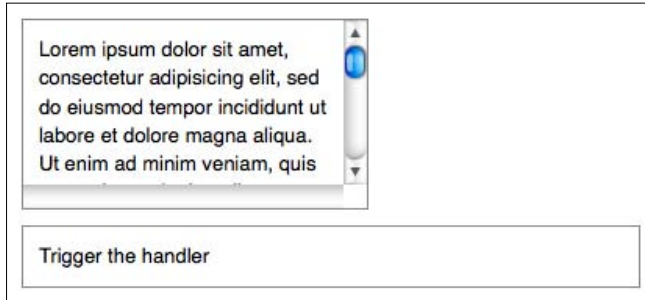
This method is a shortcut for `.bind('scroll', handler)` in the first variation, and `.trigger('scroll')` in the second.

The `scroll` event is sent to an element when the user scrolls to a different place in the element. It applies to window objects as well as to scrollable frames and elements with the `overflow` CSS property set to `scroll` (or `auto` when the element's explicit height is less than the height of its contents).

For example, consider the following HTML code:

```
<div id="target"
  style="overflow: scroll; width: 200px; height: 100px;">
  Lorem ipsum dolor sit amet, consectetur adipisicing elit,
  sed do eiusmod tempor incididunt ut labore et dolore magna
  aliqua. Ut enim ad minim veniam, quis nostrud exercitation
  ullamco laboris nisi ut aliquip ex ea commodo consequat.
  Duis aute irure dolor in reprehenderit in voluptate velit
  esse cillum dolore eu fugiat nulla pariatur. Excepteur
  sint occaecat cupidatat non proident, sunt in culpa qui
  officia deserunt mollit anim id est laborum.
</div>
<div id="other">
  Trigger the handler
</div>
```

The style definition is present to make the target element small enough to be scrollable.



The `scroll` event handler can be bound to this element.

```
$('#target').scroll(function() {  
    $.print('Handler for .scroll() called.');
```

```
});
```

Now when the user scrolls the text up or down, the following message is displayed:

Handler for .scroll() called.

We can trigger the event manually when another element is clicked.

```
$('#other').click(function() {  
    $('#target').scroll();  
});
```

After this code executes, clicks on **Trigger the handler** will also display the same message.

A `scroll` event is sent whenever the element's scroll position changes, regardless of the cause. Clicking on or dragging the scroll bar, dragging inside the element, pressing the arrow keys, or scrolling the mouse wheel could cause this event.