CHAPTER 10

Browser Events

Reacting to changes on a page is a big part of modern web application development. Although this has always been possible to *some* degree with anchor links and form submit buttons, the introduction of an events system made it possible to write code that adjusts to user input without the need to reload or changing the current page. You can probably see how this complements the ability to send AJAX requests as well. Since Internet Explorer version 4 (with the introduction of Microsoft's Trident layout engine) and Netscape Navigator version 2 (along with the first JavaScript implementation), it has been possible to listen for DOM events in order to provide a more dynamic user experience in the browser. The first implementation of this model was quite limited. But as the browser evolved via standardization, so did the events system. Today, we have a fairly elegant native API for listening to and triggering both standardized DOM events *and* custom events. You'll see throughout this chapter how you can make use of events in modern browsers to accomplish various tasks.

While the *modern* Web provides a powerful and intuitive set of methods for working with events, this was not always the case. This unfortunate point in history, along with lack of parity between Internet Explorer and the rest of the browsers in terms of the events API, made jQuery a great library for dealing with events across all popular browsers. In addition to normalizing events, jQuery also provides additional help through its support for event delegation and one-time event listener binding. jQuery is no longer needed to normalize significant event API implementation differences between browsers, but it still provides added convenience through extra-useful features. In this chapter, I also show you how to mirror the most important event-related jQuery features simply by relying on the native web API.

Throughout the following sections, you'll learn how to create, trigger, and listen for standard browser events *and* custom events. The jQuery method will be compared to the native web API approach, and you'll feel much more comfortable dealing with events yourself without using jQuery as a crutch. At the very least, even if you decide to continue using jQuery for event handling in your projects, this chapter will give you a comprehensive understanding of the event system provided by the browser. I will be focusing primarily on modern browsers in this chapter, but a section at the end will include some useful information that will help you understand how the events API worked in ancient browsers, and how this differs from the modern system.

How Do Events Work?

Before I cover *using* events to solve common problems, I think it's prudent to first outline how browser events "work." This event system follows the publish-subscribe pattern at a basic level, but there is much more to browser events than this. For starters, there are multiple classifications of browsers events. The two broadest categories are referred to as "custom" and "native" (by me).

"Native" browser events can be further assigned to sub-groups, such as mouse events, keyboard events, and touch events (to name a few). Apart from event types, browser events also have a unique attribute: the process by which they are dispersed to registered listeners. In fact, there are two distinct ways that a browser event can be broadcast across a page. Individual listeners can affect these events in various ways as well. In addition to event *types*, I also explain event *propagation* in this section. After completing this first section,

© Ray Nicholus 2016
R. Nicholus, *Beyond jQuery*, DOI 10.1007/978-1-4842-2235-5_10

you will have a good understanding of the core concepts surrounding browser events. This will allow you to effectively follow the subsequent sections that outline more specific uses of the events API.

Event Types: Custom and Native

To start off my comprehensive coverage of browser events, I'll now introduce you to the two high-level categories that all events fit into: "custom" and "native." *Native* events are those defined in one of the official web specifications, such as those maintained by WHATWG or W3C. A listing of most events can be found in the DOM Level 3 UI Events specification[1] maintained by the W3C. Please note that this is *not* an exhaustive list; it only contains a subset of the events available today. Some of the native events include "click", a mouse event triggered by the browser when a DOM element is activated via a pointing device or keyboard. Another common event is "load", which is fired when an `<img>`, document, `window`, or `<iframe>` (among others) has successfully loaded. There are quite a few other native events available. A good resource that provides a list of all currently available native DOM events can be seen on the Mozilla Developer Network events page.[2]

Custom events are, as you might expect, non-standard events that are created to be specific to a particular application or library. They can be created on-demand to support dynamic event-based workflows. For example, consider a file upload library that would like to fire an event whenever an file upload has started, and then another when the file upload is finished. Just after (or perhaps just before) the upload commences, the library might want to fire an "uploadStart" event, and then "uploadComplete" once the file is successfully on the server. It can even fire an "uploadError" event if a file upload ends prematurely. There really aren't any native events that provide the semantics needed for this situation, so custom events are the best solution. Luckily, the DOM API does provide a way to trigger custom events. Although triggering custom events has been a bit inelegant in some browsers without a polyfill, that is changing. More on that later.

A custom event created and triggered without jQuery can be observed and handled using jQuery's event API. However, there is an interesting limitation associated with jQuery when dealing with custom events, something you won't find anywhere in the jQuery documentation. A custom event created and triggered with jQuery's event API *cannot* be observed and handled without also using jQuery's event API. In other words, custom events created by jQuery are entirely proprietary and non-standard. The reason for this is actually quite simple. Although it is possible for jQuery to trigger all event handlers on a specific element for a native DOM event, it is *not* possible to do the same for custom events, nor is it possible to query a specific HTML element for its attached event listeners. For this reason, *jQuery* custom events are only usable by *jQuery* custom event handlers.

There is one object that ties custom and native events together. This central object is `the Event` object,[3] which is described in a specification curated by the W3C. Every DOM event, custom or native, is represented by an `Event` object, which itself has a number of properties and methods useful for identifying and controlling the event. For example, a `type` property makes the name of the custom or native event available. A "click" event has a `type` of 'click' on its corresponding `Event` object. That same `Event` object instance will also contain, among others, a `stopPropagation()` method, which can be called to prevent the click event from being further broadcasted to other listeners on the page.

Event Propagation: Bubbling vs. Capturing

In the early days of the Web, Netscape provided one way to disperse events throughout the DOM—event capturing—while Internet Explorer provided a contrasting process—event bubbling. Before standardization, browsers essentially made their own proprietary choices when implementing features, which is what led to these two divergent approaches. This all changed in 2000 when the W3C drafted the DOM Level 2 Events Specification.[4] This document describes an event model that includes both event capturing *and* bubbling.

[1] `www.w3.org/TR/DOM-Level-3-Events`
[2] `https://developer.mozilla.org/en-US/docs/Web/Events`
[3] `www.w3.org/TR/uievents/#h-event-interfaces`
[4] `www.w3.org/TR/DOM-Level-2-Events/`

All browsers that adhere to this specification follow this process of distributing events across the DOM. Currently, all modern browsers implement DOM Level 2 Events. Ancient browsers, which only support event bubbling, are covered at the end of this chapter.

In all modern browsers, per the DOM Level 2 Events spec, When a DOM event is created, the capturing phase begins. Assuming the event is not canceled at some point after it is triggered, it starts at `window`, followed by `document`, and propagates downwards, ending with the element that triggered the event. After the capturing phase is complete, the bubbling phase commences. Starting with this target element, the event "bubbles" up the DOM, hitting each ancestor, until the event is cancelled or until it hits `window` again.

If the description regarding the progress of an event is still a bit confusing, let me try to explain with a simple demonstration. Consider the following HTML document:

```html
<!DOCTYPE html>
<html>
<head>
  <title>event propagation demo</title>
</head>
<body>
  <section>
    <h1>nested divs</h1>
    <div>one
      <div>child of one
        <div>child of child of one</div>
      </div>
    </div>
  </section>
</body>
</html>
```

Suppose the `<div>child of child of one</div>` element is clicked. The click event takes the following path through the DOM:

Capturing Phase

1. `window`
2. `document`
3. `<html>`
4. `<body>`
5. `<section>`
6. `<div>one`
7. `<div>child of one`
8. `<div>child of child of one`

Bubbling Phase

9. `<div>child of child of one`
10. `<div>child of one`

11. `<div>one`
12. `<section>`
13. `<body>`
14. `<html>`
15. `document`
16. `window`

So, when is it appropriate to focus on the capturing phase instead of the bubbling phase, or vice versa? The most common choice is to intercept events in the bubbling phase. One reason for the overwhelming focus on the bubbling phase is due to historical reasons. Before Internet Explorer 9, this was the *only* phase available. This is no longer an obstacle with the advent of standardization and modern browsers. In addition to lack of support for capturing in ancient browsers, jQuery *also* lacks support for this phase. This is perhaps yet another reason why it is not particularly popular, and bubbling is the default choice. But there is no denying that the concept of event bubbling is more intuitive than capturing. Envisioning an event that moves up the DOM tree, starting with the element that created the event, seems to be a bit more sensible than an event that *ends* with the event that created it. In fact, capturing is rarely discussed when describing the browser event model. Attaching a handler to the event bubbling phase is also the default behavior when using the web API to listen for events.

Even though the event bubbling phase is often preferred, there *are* instances where capturing is the better (or only) choice. There appears to be a performance advantage to utilization of the capturing phase. Because event capturing occurs *before* bubbling, this seems to make sense. Basecamp, a web-based project-management application, has made use of event capturing to improve performance in their project,[5] for example. Another reason to make use of the capturing phase: event delegation for "focus"[6] and "blur"[7] events. Although these events do not bubble, handler delegation is possible by hooking into the capturing phase. I cover event delegation in detail later in this chapter.

Capturing can also be used to react to events that have been cancelled in the bubbling phase. There are a number of reasons to cancel an event—that is, to prevent it from reaching any subsequent event handlers. Events are almost always cancelled in the bubbling phase. I discuss this a bit more later in this chapter, but for now imagine a click event cancelled by a third-party library in your web application. If you still need access to this event in another handler, you can register a handler on an element in the capturing phase.

jQuery has made the unfortunate choice of artificially bubbling events. In other words, when an event is triggered via the library, it calculates the expected bubbling path and triggers handlers on each element in this path. jQuery does not make use of the native support for bubbling and capturing provided by the browser. This certainly adds complexity and bloat to the library and likely introduces performance consequences as well.

Creating and Firing DOM Events

To demonstrate firing DOM events with and without jQuery, let's use the following HTML fragment:

```html
<div>
  <button type="button">do something</button>
</div>

<form method="POST" action="/user">
```

[5]https://signalvnoise.com/posts/3137-using-event-capturing-to-improve-basecamp-page-load-times
[6]www.w3.org/TR/uievents/#event-type-focus
[7]www.w3.org/TR/uievents/#event-type-blur

```
  <label>Enter user name:
    <input name="user">
  </label>
  <button type="submit">submit</button>
</form>
```

Firing DOM Events with jQuery

jQuery's events API contains two methods that allow DOM events to be created and propagated throughout the DOM: `trigger` and `triggerHandler`. The `trigger` method is most commonly used. It allows an event to be created and propagated to all ancestors of the originating element via bubbling. Remember that jQuery artificially bubbles all events, and it does *not* support event capturing. The `triggerHandler` method differs from `trigger` in that it only executes event handlers on the element in which it is called; the event is *not* bubbled to ancestor elements. jQuery's `triggerHandler` is different from `trigger` in some other ways as well, but the definition I have provided is sufficient for this section.

In the next few listings, I use jQuery's `trigger` method to:

1. Submit the form in two ways.
2. Focus the text input.
3. Remove focus from the input element.

Listing 10-1. Triggering DOM Events: jQuery

```
// submits the form
$('FORM').trigger('submit');

// submits the form by clicking the button
$('BUTTON[type="submit"]').trigger('click');

// focuses the text input
$('INPUT').trigger('focus');

// removes focus from the text input
$('INPUT').trigger('blur');
```

To be fair, there is a second way to trigger the same events with jQuery, by using aliases for these events defined in the library's API.

Listing 10-2. Another Way of Triggering DOM Events: jQuery

```
// submits the form
$('FORM').submit();

// submits the form by clicking the button
$('BUTTON[type="submit"]').click();

// focuses the text input
$('INPUT').focus();

// removes focus from the text input
$('INPUT').blur();
```

What if we want to click the button that appears before the form, but we don't want to trigger any click handlers attached to ancestor elements? Suppose the parent `<div>` contains a click handler that we do not want to trigger in this instance. With jQuery, we can use `triggerHandler()` to accomplish this, as shown in Listing 10-3.

Listing 10-3. Triggering DOM Events without Bubbling: jQuery

```
// clicks the first button - the click event does not bubble
$('BUTTON[type="button"]').triggerHandler('click');
```

Web API DOM Events

There are two or three ways to trigger the same events just demonstrated *without* using jQuery. It's good to have choices (sometimes). Anyway, the easiest way to trigger the preceding events using the web API is to invoke corresponding native methods on the target elements. Listing 10-4 shows code that looks very similar to Listing 10-2.

Listing 10-4. Triggering DOM Events: Web API, All Modern Browsers, and Internet Explorer 8

```
// submits the form
document.querySelector('FORM').submit();

// submits the form by clicking the button
document.querySelector('BUTTON[type="submit"]').click();

// focuses the text input
document.querySelector('INPUT').focus();

// removes focus from the text input
document.querySelector('INPUT').blur();
```

The preceding code is only limited to IE8 and newer due to use of `querySelector()`, but this is far more than sufficient given the current year and state of browser support. The `click()`, `focus()`, and `blur()` methods are available on all DOM element objects that inherit from `HTMLElement`.[8] The `submit()` method is available to `<form>` elements only, as it is defined on the `HTMLFormElement` interface.[9] The "click" and "submit" events bubble when they are triggered with these methods. Per the W3C specification,[10] "blur" and "focus" events do *not* bubble, but they are available to event handlers that are coded to make use of the capturing phase.

The preceding events can also be created by using either the `Event()` constructor or the `createEvent()` method available on `document`.[11] The former is supported in all modern browsers, with the exception of any version of Internet Explorer. In the next code demonstration, I show you how to programmatically determine whether the `Event` constructor is supported and then fall back to the alternative path for triggering events. Perhaps you are wondering why you would even need to trigger events using an approach different from the simple one outlined here. If you want to alter the default behavior of an event in some way, construction of an `Event` object is required. For example, in order to mimic the behavior of jQuery's `triggerHandler()` method and prevent an event from bubbling, we must pass a specific configuration property to our "click" event when constructing it. You will see this at the end of Listing 10-5, which demonstrates a second approach to triggering events.

[8] www.w3.org/TR/html5/dom.html#htmlelement
[9] www.w3.org/TR/html5/forms.html#the-form-element
[10] www.w3.org/TR/DOM-Level-2-Events/
[11] www.w3.org/TR/DOM-Level-3-Events/#widl-DocumentEvent-createEvent

Listing 10-5. Triggering DOM Events without Bubbling: Web API, All Modern Browsers

```
var clickEvent;

// If the `Event` constructor function is not supported,
// fall back to `createEvent` method.
if (typeof Event === 'function') {
  clickEvent = new Event('click', {bubbles: false});
}
else {
    clickEvent = document.createEvent('Event');
    clickEvent.initEvent('click', false, true);
}

document.querySelector('BUTTON[type="button"]')
.dispatchEvent(clickEvent);
```

In the listing, when we must fall back to `initEvent()`, the second parameter is `bubbles`, which must be set to `false` if we do *not* want the event to bubble. The third parameter, which is set to true, indicates that this event is indeed cancelable. In other words, any default browser actions associated with this event may be prevented using the preventDefault() method on the Event object. I'll explain canceling events a bit more later on in this chapter. The `Event` constructor provides a more elegant way to set this and other options using a set of object properties. Once Internet Explorer 11 is dead and buried, we can focus exclusively on the `Event` constructor and forget `initEvent()` ever existed. But until then, the preceding check will allow you to choose the proper path if you must construct an event with special configuration options.

Creating and Firing Custom Events

Remember that custom events are those that are not standardized as part of an accepted web specification, such as those maintained by the W3C and the WHATWG. Let's imagine a scenario where we are writing a third-party library that handles adding and removing items from a gallery of images. When our library is integrated into a larger application, we need to provide a simple way to notify any listeners when an item is added or removed by our library. In this case, our library will wrap the gallery of images so we can signal a removal or addition simply by triggering an event that can be observed by an ancestor element. There aren't any standardized DOM events that are appropriate here, so we'll need to create our own event, a *custom* event. The custom event associated with removing an image will be aptly named "image-removed", and will need to include the ID of the removed image.

jQuery Custom Events

Let's first fire this event using jQuery. We'll assume that we already have a handle on an element controlled by our library. Our event will be triggered by this particular element:

```
// Triggers a custom "image-removed" element,
// which bubbles up to ancestor elements.
$libraryElement.trigger('image-removed', {id: 1});
```

This looks identical to the code used to trigger native DOM events, and it is. jQuery has a simple, elegant, and consistent API for triggering events of all types. But there is a problem here—the code that listens for this event, outside of our jQuery library, *must* also use jQuery to observe this event. This is a limitation of jQuery's custom event system. It matters little if the user of our library is using some other library or even if the user does not want to use jQuery outside of this library. Perhaps it is not clear to our user that jQuery *must* be used to listen for this event. They are forced to rely on jQuery and jQuery alone to accept messages from our library.

Firing Custom Events with the Web API

Triggering custom events with the web API is exactly like triggering native DOM events. The difference here is with *creating* custom events, although the process and API are still quite similar.

```
var event = new CustomEvent('image-removed', {
  bubbles: true,
  detail: {id: 1}
});
libraryElement.dispatchEvent(event);
```

Here, we can easily create our custom event, trigger it, and ensure it bubbles up to ancestor elements. So, our "image-removed" event is observable outside of our library. We've also passed the image ID in the `detail` property of the event payload. More on accessing this data later on. But there's a problem here: this will *not* work in any version of Internet Explorer. Unfortunately, as I mentioned in the last section, Explorer does not support the `Event` constructor. So, we must fall back to the following approach for cross-browser support:

```
var event = document.createEvent('CustomEvent');
event.initCustomEvent('image-removed', false, true, {id: 1});
libraryElement.dispatchEvent(event);
```

Instead of creating an 'Event,' as we did when attempting to trigger a native DOM event in Internet Explorer in the previous section, we must instead create a 'CustomEvent.' This exposes an `initCustomEvent()` method, which is defined on the `CustomEvent` interface. This special method allows us to pass custom data along with this event, such as our image ID.

The preceding code currently (as of mid-2016) works in all modern browsers, but this *may* change in the future once the `CustomEvent` constructor is supported in all browsers. It *may* be removed from any future browser version. To make our code future proof and still ensure it works in Internet Explorer, we need to use the check for the existence of the `CustomEvent` constructor, just as we did with the `Event` constructor in the previous section:

```
var event;

// If the `CustomEvent` constructor function is not supported,
// fall back to `createEvent` method.
if (typeof CustomEvent === 'function') {
  event = new CustomEvent('image-removed', {
    bubbles: true,
    detail: {id: 1}
  });
}
else {
    event = document.createEvent('CustomEvent');
    event.initCustomEvent('image-removed', false, true, {
      id: 1
    });
}

libraryElement.dispatchEvent(event);
```

After Internet Explorer fades into obsolesce and Microsoft Edge takes its place, you can use the `CustomEvent` constructor exclusively, and the preceding code will no longer be needed at all.

Listening (and Un-listening) to Event Notifications

Triggering events is *one* important part of passing messages around the DOM, but these events provide more value with corresponding listeners. In this section, I cover handling DOM and custom events. You may already be familiar with the process of registering event observers with jQuery, but I'll demonstrate how this is done first so that the differences are apparent when relying exclusively on the web API.

A `window` resize event handler may be important to make adjustments to a complex application as the view of your page is changed by the user. This "resize" event, which is triggered on the `window` when the browser is resized by the user, will provide us with a good way to demonstrate registering and unregistering event listeners.

jQuery Event Handlers

jQuery's `on` API method provides everything necessary to observe both DOM and custom events triggered on an element:

```
$(window).on('resize', function() {
  // react to new window size
});
```

If, at some point in the future, we no longer care about the size of the window, we can remove this handler using jQuery's appropriately named `off` handler, but that is less straightforward than adding a new listener. We have two options:

1. Remove *all* resize event listeners (easy).
2. Remove only our resize event listener (a bit harder).

Let's look at option 1 first:

```
// remove all resize listeners - usually a bad idea
$(window).off('resize');
```

The first option is quite simple, but we run the risk of causing problems for other code on the page that still relies on window resize events. In other words, option 1 is usually a poor choice. That leaves us with option 2, which requires that we store a reference to our handler function and provide that to jQuery so it can unbind only our listener. So, we will need to rewrite our event listener call from before such that we can easily unregister our listener later:

```
var resizeHandler = function() {
    // react to new window size
};

$(window).on('resize', resizeHandler);

// ...later
// remove only our resize handler
$(window).off('resize', resizeHandler);
```

Binding an event listener to a specific element and then removing it later when it is no longer needed is usually sufficient, but we may have a situation where an event handler is only ever needed once. After it is first executed, it can and should be removed. Perhaps we have an element that, once clicked, changes state in such a way that subsequent clicks are not prudent. jQuery provides a `one` API method for such cases:

```
$(someElement).one('click', function() {
  // handle click event
});
```

After the attached handler function is executed, the click event will no longer be observed.

Observing Events with the Web API

Since the beginning of time (almost), there have been two simple ways to attach an event handler to a specific DOM element, and both can be considered "inline." The first, shown in Listing 10-6, involves including the event handler function as the value of an attribute of the element in the document markup:

Listing 10-6. Inline Event Handler: Web API, All Browsers

```
<button onclick="handleButtonClick()">click me</button>
```

There are a couple problems with this approach. First, it requires a global `handleButtonClick()` function. If you have a number of buttons or other elements that require specific click handler functions, you will end with a cluttered and messy global namespace. Global variables and functions should always be limited to prevent conflicts and uncontrolled access to internal logic. This type of inline event handler is a step in the wrong direction for that reason.

A second reason why this is bad: it requires mixing your JavaScript and HTML in the same file. Generally speaking, that is discouraged because it goes against the principle of separation of concerns. That is, content belongs in HTML files, and behavior belongs in JavaScript files. This isolates the code such that changes are potentially less risky, and caching is improved as changes to JavaScript do not invalidate the markup files, and vice versa.

Another way to register for the same click event requires attaching a handler function to the element's corresponding event property:

```
buttonEl.onclick = function() {
  // handle button click
};
```

Although this approach is slightly better than the HTML-based event handler, since we are not forced to bind to a global function, it is still a non-optimal solution. You cannot specify an attribute-based event handler *and* an element property handler for the same event on the same element. The last specified handler will effectively remove any other inline event handler on the element for a given event type. In fact, only one total inline event handler may be specified for a given event on a given element. This is likely a big issue for modern web applications, as it is quite common for multiple uncoordinated modules to exist on the same page. Perhaps more than one of these modules needs to attach an event handler of the same type to the same element. With inline event handlers, this simply is not possible.

Since Internet Explorer 9, an `addEventListener()` method has been available on the `EventTarget` interface. All `Element` objects implement this interface, as does `Window` (among other DOM objects). The `EventTarget` interface first appeared in the W3C DOM Level 2 Events specification,[12] and the `addEventListener()` method was part of this initial version of the interface. Registering for custom and

[12]www.w3.org/TR/DOM-Level-2-Events/

DOM events is possible with this method, and the syntax is very similar to jQuery's `on()` method. Continuing with the button example:

```
buttonEl.addEventListener('click', function() {
  // handle button click
});
```

This solution does not suffer from any of the issues that plague inline event handlers. There is no required binding to global functions. You may attach as many different click handlers to this button element as you like. The handler is attached purely with JavaScript, so it may exist exclusively in a JavaScript file. In the previous section, I reminded you how a "resize" event was bound to the `window` with jQuery. The same handler using the modern web API looks like this:

```
window.addEventListener('resize', function() {
  // react to new window size
});
```

This looks a *lot* like our example of binding to the same event with jQuery, with the exception of a longer event binding method (`on()` versus `addEventListener()`). You may be glad to know that an appropriately named web API method exists to un-bind our handler, should we need to do this at some point. The `EventTarget` interface also defines a `removeEventListener()` method. The `removeEventListener()` method differs from jQuery's `off` in one notable way: there is no way to remove *all* event listeners of a given type from a particular element. Perhaps this is a good thing. So, in order to remove our `window` "resize" handler, we must structure our code like so:

```
var resizeHandler = function() {
    // react to new window size
};

window.addEventListener('resize', resizeHandler);

// ...later
// remove only our resize handler
window.removeEventListener('resize', resizeHandler);
```

Remember the one-time click handler we created with jQuery's one API method? Does something similar exist in the web API? Good news: yes! Bad news: It's a fairly new addition (added to the WHATWG's DOM living standard (`https://dom.spec.whatwg.org/#interface-eventtarget`). As of mid-2016, only Firefox offers support, but it's an exciting feature:

```
someElement.addEventListener('click', function(event) {
   // handle click event
}, { once: true });
```

For better cross-browser support, especially since there isn't any elegant way to programmatically determine support for listener options, use this instead:

```
var clickHandler = function() {
  // handle click event
  // ...then unregister handler
  someElement.removeEventListener('click', clickHandler);
};
someElement.addEventListener('click', clickHandler);
```

After the attached handler function is executed, the click event will no longer be observed, just like jQuery's `one()` method. There *are* more elegant ways to solve this problem, but this solution makes use of only what you have learned so far in this book. You may discover a more elegant method after completing this book, especially after reading about JavaScript utilities.

Controlling Event Propagation

I've showed you how to fire and observe events in the last couple sections, but sometimes you need to do more than just create or listen for events. Occasionally, you need to either influence the bubbling/capturing phase or even attach data to an event in order to make it available to subsequent listeners.

As a contrived example (but possibly realistic in a very twisted web application piloted by project managers who have no understanding of the web), suppose you were asked to prevent users from selecting any text or images on an entire page. How can you accomplish that? Perhaps by interfering with some mouse event, somehow. But which event, and how? Perhaps the event to concentrate on is "click". If this was your first guess, you were close, but not quite correct.

According to the W3C DOM Level 3 Events specification, the "mousedown" event[13] starts a drag or text selection operation as its *default* action. So, we must prevent the *default* action of a "mousedown" event. We can prevent text and image selection/dragging across the entire page using jQuery *or* the pure web API by simply registering a "mousedown" event listener on `window` and calling the `preventDefault()` method on the `Event` object that is passed to our handler once our handler is executed:

```
$(window).on('mousedown', function(event) {
  event.preventDefault();
});

// ...or...
$(window).mousedown(function(event) {
  event.preventDefault();
});
```

preventing a default event action - web API - modern browsers

```
window.addEventListener('mousedown', function(event) {
  event.preventDefault();
});
```

The jQuery approach is almost identical to the one that only relies on the native web API. Either way, we've satisfied the requirements: no text or images can be selected or dragged on our page.

This is probably a good time to start discussing the `Event` object. Earlier, an `Event` instance is passed to our event handler function when it is executed by jQuery or the browser. The native `Event` interface is defined in the W3C DOM4 specification.[14] When a custom or native DOM event is created, the browser creates an instance of `Event` and passes it to each registered listener during the capturing and bubbling phases.

The event object passed to each listener contains a number of properties that, for instance, describe the associated event, such as:

- The event type (click, mousedown, focus)
- The element that created the event

[13]www.w3.org/TR/DOM-Level-3-Events/#event-type-mousedown
[14]www.w3.org/TR/dom/#interface-event

- The current event phase (capturing or bubbling)
- The current element in the bubbling or capturing phase

There are other, similar properties, but that list represents a good sampling of the more notable ones. In addition to properties that describe the event, there are also a number of methods that allow the event to be controlled. One such method is `preventDefault()`, as I just demonstrated. But there are others, which I will cover shortly.

jQuery has its own version of the `Event` interface (of course).[15] According to jQuery's docs, their event object "normalizes the event object according to W3C standards." This was potentially useful for ancient browsers. But for modern browsers, not so much. For the most part, with the exception of a few properties and methods, the two interfaces are very similar.

The next two examples demonstrate how to prevent a specific event from reaching other registered event handlers. In order to prevent a click event from reaching any event handler on subsequent DOM nodes, simply call `stopPropagation()` on the passed `Event` object. This method exists in both the jQuery `Event` interface *and* the standardized web API `Event` interface:

```
$someElement.click(function(event) {
    event.stopPropagation();
});

// ...or...

$someElement.on('click', function(event) {
    event.stopPropagation();
});
```

Using jQuery, you can stop an event from bubbling with `stopPropagation()`, but you *cannot* stop the event during the capturing phase, unless of course you defer to the web API, as shown in Listing 10-7.

Listing 10-7. Stop a Click Event from Propagating: Web API, Modern Browsers

```
// stop propagation during capturing phase
someElement.addEventListener('click', function(event) {
    event.stopPropagation();
}, true);

// stop propagation during bubbling phase
someElement.addEventListener('click', function(event) {
    event.stopPropagation();
});
```

The web API is capable of stopping event propagation in either the capturing *or* the bubbling phase. But `stopPropagation()` does not stop the event from reaching any subsequent listeners on the *same* element. For this task, the `stopImmediatePropagation()` event method is available, which stops the event from reaching any further handlers, whether they are registered on the current DOM node or subsequent nodes. Again, jQuery (Listing 10-8) and the web API (Listing 10-9) share the same method name, but jQuery is restricted to the bubbling phase, as always.

[15] `https://api.jquery.com/category/events/event-object/`

Listing 10-8. Stop a Click Event from Reaching Any Other Handlers: jQuery

```
$someElement.on('click', function(event) {
    event.stopImmediatePropagation();
});
```

Listing 10-9. Stop a Click Event from Reaching Any Other Handlers: Web API, Modern Browsers

```
someElement.addEventListener('click', function(event) {
    event.stopImmediatePropagation();
});
```

Note that both jQuery and the web API offer a shortcut to prevent an event's default action *and* to prevent the event from reaching handlers on subsequent DOM nodes. You can effectively call both `event.preventDefault()` and `event.stopPropagation()` by returning `false` in your event handler.

Passing Data to Event Handlers

Sometimes the standard data associated with an event is not enough. Event handlers may need more specific information about the event they are handling. Remember the "uploadError" custom event I detailed earlier? This is triggered from a library embedded on a page, and the "uploadError" event exists to provide information to listeners outside of the library about a failed file upload. Suppose the file upload library we are using is attached to a container element, and our application wraps this container element and registers an "uploadError" event handler. When a particular file upload fails, this event is triggered, and our handler displays an informational message to the user. In order to customize this message, we need the name of the failed file. The upload library can pass the file's name to our handler in the `Event` object.

First, let's review how data is passed to event handlers with jQuery:

```
// send the failed filename w/ an error event
$uploaderElement.trigger('uploadError', {
  filename: 'picture.jpeg'
});

// ...and this is a listener for the event
$uploaderParent.on('uploadError', function(event, data) {
  showAlert('Failed to upload ' + data.filename);
});
```

jQuery makes the object passed to the `trigger()` function available to any event listeners via a second parameter passed to the handler. With this, we can access any properties on the passed object.

To achieve the same result with the web API, we would make use of `CustomElement` and its built-in ability to handle data:

```
// send the failed filename w/ an error event
var event = new CustomEvent('uploadError', {
  bubbles: true,
  detail: {filename: 'picture.jpeg'}
});
uploaderElement.dispatchEvent(event);

// ...and this is a listener for the event
```

```
uploaderParent.addEventListener('uploadError', function(event) {
  showAlert('Failed to upload ' + event.detail.filename);
});
```

This is not as succinct as the jQuery solution, but it works, at least in all modern browsers other than IE. For a more cross-browser solution, you can rely on the old custom event API, as demonstrated earlier. I'll focus only on the old API in the following demonstration, but I encourage you to read about a more future-proof method of creating `CustomEvent` instances covered earlier:

```
// send the failed filename w/ an error event
var event = document.createEvent('CustomEvent');
event.initCustomEvent('uploadError', true, true, {
  filename: 'picture.jpeg'
});
uploaderElement.dispatchEvent(event);

// ...and this is a listener for the event
uploaderParent.addEventListener('uploadError', function(event) {
  showAlert('Failed to upload ' + event.detail.filename);
});
```

In both cases, the data attached to the `CustomEvent` is available to our listeners via a standardized `detail` property.[16] With the `CustomEvent` constructor, this data is provided on a `detail` property of the object passed when creating a new instance. The `detail` property on this object matches the `detail` property on the `CustomEvent` object available to our listeners, which is nice and consistent. Our listeners still have access to this same `detail` property when setting up our "uploadError" event using the old event creation API, but it is buried in a sea of parameters passed to `initCustomEvent()`. In my experience, anything more than two parameters is confusing and non-intuitive. This is not an uncommon preference, which may explain why the more modern `CustomEvent` constructor only mandates two parameters, the second being an object where all custom even configuration and data is provided.

Event Delegation: Powerful and Underused

There have been multiple occasions where I have danced around and entirely avoided a very important topic: event delegation. Simply put, *event delegation* involves attaching a single event handler to a top-level element with the intention of handling events that bubble up from descendant elements. The event handler logic in this top-level element may contain code paths that differ based on the event target element (the element that first received the event). But why do this? Why not simply attach specific event handlers directly to the appropriate elements?

One reason that has been discussed ad nauseam is potential performance benefits of delegated event handlers. CPU cycles are saved by binding a single event handler that is responsible for monitoring events on many descendant elements, as opposed to querying each element and attaching a dedicated handler function to each element directly. This theory makes a lot of sense, and of course it's true. But how much time in terms of CPU cycles is really saved here? I suppose the answer to that question is: it depends. It depends on how many elements you intend to monitor, for one.

[16] https://dom.spec.whatwg.org/#dom-customevent-detail

It's hard to imagine a common scenario where delegated event handling is both desirable and detrimental to performance. The practice has caught on, partially for the expected performance reasons, and also due to the ability to centralize event handling code to one specific root element instead of spreading it out across the DOM. Take React, for example. React is a JavaScript library that focuses specifically on the "view" portion of a typical Model View Controller web application. In terms of event handling, React has implemented an interesting abstraction:[17]

> React doesn't actually attach event handlers to the nodes themselves. When React starts up, it starts listening for all events at the top level using a single event listener.

In other words, all event handlers attached to elements with React are promoted to a single delegated event handler on a common parent element. Perhaps you still don't see an instance where delegated event handlers are appropriate. In the rest of this section, I focus on a simple example that demonstrates the power of event delegation.

Suppose you have a list filled with list items, each with a button that removes an item from the list. You *could* attach a click handler to each individual list item's button. But doesn't it seem like the wrong approach to loop over all of the button elements and attach the very same click handler function to each? You may argue that this is not unreasonable, and even easy to accomplish. But what if new items can be added to this list dynamically after the initial page load? Now attaching a new event handler to each new list item, after it is added, becomes less appealing.

The best solution here is to use event delegation. In other words, attach one click handler to the list element. When any of the delete buttons inside of the list item elements is clicked, the event will bubble up to the list element. At this point, your one event handler will be hit, and you can easily determine, by inspecting the event object, which list item was clicked and respond appropriately by removing the associated list item. In this section, we're using some text to make our delete buttons more accessible, as well as close/remove icons from the Ionicons site[18] to enhance the appearance of our buttons.

The HTML for such a list may look something like this:

```
<link href="http://code.ionicframework.com/ionicons/2.0.1/css/ionicons.min.css"
      rel="stylesheet">
<ul id="cars-list">
    <li>Honda
      <button>
        <span>delete</span>
        <span class="ion-close-circled"></span>
      </button>
    </li>
    <li>Toyota
      <button>
        <span>delete</span>
        <span class="ion-close-circled"></span>
      </button>
    </li>
    <li>Kia
      <button>
        <span>delete</span>
        <span class="ion-close-circled"></span>
```

[17]https://facebook.github.io/react/docs/interactivity-and-dynamic-uis.html#under-the-hood-autobinding-and-event-delegation
[18]http://ionicons.com

```
      </button>
    </li>
    <li>Ford
      <button>
        <span>delete</span>
        <span class="ion-close-circled"></span>
      </button>
    </li>
</ul>
```

With jQuery, we can use the `click` alias to attach a click handler to the `<ul>` and delete the appropriate car list item by inspecting the event object:

```
$('#cars-list').on('click', 'button', function() {
    $(this).closest('li').remove();
});
```

But wait, we didn't have to examine the event object at all! jQuery provides a nice feature here by setting the event handler function's context (`this`) to the click element target. Note that this click event might target the "delete" span element *or* the "x" icon, depending upon which of these elements is selected by our user. In either case, we are *only* interested in clicks on the `<button>` or its children. jQuery ensures that our event handler will only be called if this is true, and at that point we can use jQuery's `closest()` method to find the associated `<li>` and remove it from the DOM, as shown in Listing 10-10.

Listing 10-10. Delegated Event Handling: Web API, All Modern Browsers (with closest Shim)

```
document.querySelector('#cars-list')
  .addEventListener('click', function(event) {
    if (event.target.closest('BUTTON')) {
      var li = event.target.closest('LI');
      li.parentElement.removeChild(li);
    }
  });
```

The above pure web API solution is a bit more verbose than jQuery's API (ok, it's a *lot* more verbose). But it shows how to accomplish a few common goals *without* jQuery:

1. Attach a click event handler to an element.
2. Examine an `Event` object to focus on events triggered only by one of our `<button>` elements.
3. Remove the `<li>` associated with the clicked delete button.

We're making use of `Element.closest` here to easily find the parent `<li>` and determine if the event target's parent is indeed a `<button>`, all without explicitly dealing with the fact that the event target may be multiple levels beneath the `<button>` or `<li>`. Since `Element.closest` is not supported in Internet Explorer, Microsoft Edge (at least, as of version 13), or older versions of iOS Safari and Android, you'll need to make use of the shim demonstrated in Chapter 4 if you require solid cross-browser support.

This may all seem a bit inelegant compared to jQuery, and that may indeed be true. But remember, the mission of this book isn't necessarily about compelling you to eschew jQuery or any other library, but rather to show you how to solve the same problems by yourself *without* the aid of third-party dependencies. The knowledge gained from these exercises and demonstrations will empower you as a web developer by offering insight into the web API and allow you to make better decisions when deciding if your project will benefit from some outside help. And perhaps you will elect to pull in small and focused shims (such as the `Element.closest` polyfill demonstrated earlier) instead of depending on a large library like jQuery.

Handling and Triggering Keyboard Events

Keyboard events are, as you might expect, native DOM events triggered by the browser when a user presses a key on their keyboard. Just like all other events, keyboard events go through both a capturing and a bubbling phase. The target element is the one that is focused when the key is pressed. You may be wondering why I have dedicated a special section exclusively to keyboard events. As you'll see shortly, key events are a bit different than the other DOM and custom events discussed earlier. Plus, keyboard events *can* be a bit trickier to handle than the other DOM events.

This is mostly due to mostly misunderstood multiple keyboard event types *and* the confusing array of event properties used to identify keys with varying browser support. Fear not—after this section, you'll have a pretty good understanding of keyboard events. You'll fully comprehend when to use the three types of keyboard events, how to identify the pressed key, and even how to put this knowledge to use to solve real problems.

Three Types of Keyboard Events

Each key on your keyboard is *not* a separate event type. Instead, a keyboard-triggered action is attached to one of three possible keyboard-specific event types:

1. keydown
2. keyup
3. keypress

The browser triggers a "keydown" event before a pushed key is released. It may fire repeatedly if the key is held down, and is triggered for *any* key on the keyboard (even Shift/Ctrl/Command/other). Though some keys will *not* trigger multiple "keydown" events when held, such as Shift/Command/Option/Function. jQuery provides a `keydown` alias in its API to handle these events:

```
$(document).keydown(function(event) {
  // do something with this event
});
```

And the same event can be handled (for the web API and modern browsers) without jQuery using trusty old `addEventListener()`:

```
document.addEventListener('keydown', function(event) {
  // do something with this event
});
```

After a key (any key) in the down position is released, the browser fires a "keyup" event. The logic for handling this event is identical to "keydown"—just replace "keydown" with "keyup" in the code samples just given. In fact, the same is true for the "keypress" event, which is the the final keyboard event type. The "keypress" event is *very* similar to "keydown"—it is also triggered when a pressed key is in the down position. The only difference is that "keypress" events are only fired for printable characters. For example, pressing the "a," "Enter," or "1" keys will trigger a "keypress" event. Conversely, "Shift," "Command," and the arrow keys will *not* result in a "keypress" event.

Identifying Pressed Keys

Suppose we are building a modal dialog. If a user of our dialog presses the "Esc" key, we would like to close the dialog. In order to accomplish this, we need to do a few things:

1. Listen for "keydown" events on the document.
2. Determine if the "keydown" event corresponds to the "Esc" key.
3. If the "keydown" event *is* the result of pressing "Esc," close the modal dialog.

If we are using jQuery, our code will look something like this:

```
$(document).keydown(function(event) {
  if (event.which === 27) {
    // close the dialog...
  }
});
```

The number 27 corresponds to the ESC key's key code.[19] We can make use of the same key code without jQuery (for the web API and modern browsers) by also looking at the `which` property on the `KeyboardEvent` that our handler receives:

```
document.addEventListener('keydown', function(event) {
  if (event.which === 27) {
    // close the dialog...
  }
});
```

But the web API is beginning to advance beyond jQuery in this respect. The UI Events Specification,[20] maintained by the W3C, defines a *new* property on `KeyboardEvent`: `key`.[21] When a key is pressed, the `KeyboardEvent` (in supported browsers) will contain a `key` property with a value that corresponds to the exact key pressed (for printable characters) *or* a standardized string that describes the pressed key (for non-printable characters). For example, if the "a" key is pressed, the `key` property on the corresponding `KeyboardEvent` will contain a value of "a". In our case, the Esc key is represented as the string "Escape". This value, along with the key values for other non-printable characters, are defined in the DOM Level 3 Events specification,[22] also maintained by W3C. If we are able to use this key property, our code will look like Listing 10-11.

Listing 10-11. Closing a Modal Dialog on Esc: Web API, All Modern Browsers Except Safari

```
document.addEventListener('keydown', function(event) {
  if (event.key === 'Escape') {
    // close the dialog...
  }
});
```

[19] https://lists.w3.org/Archives/Public/www-dom/2010JulSep/att-0182/keyCode-spec.html#fixed-virtual-key-codes
[20] www.w3.org/TR/uievents/
[21] www.w3.org/TR/uievents/#widl-KeyboardEvent-key
[22] www.w3.org/TR/DOM-Level-3-Events-key/

At the moment, Safari is the *only* modern browser *without* support for this `KeyboardEvent` property. This will likely change as the WebKit engine evolves. In the meantime, you may want to consider sticking with the `which` property until Safari is brought up-to-date.

Making an Image Carousel Keyboard Accessible with the Web API

One important benefit of mastering keyboard events: accessibility. An accessible web application is one that can be easily used by those with varying requirements. Perhaps the most common accessibility consideration involves ensuring that those without the ability to use a pointing device can completely and effectively navigate the web application. This requires, in some cases, listening for keyboard events and responding appropriately.

Suppose you are building an image carousel library. An array of image URLs is passed to the carousel, the first image rendered in a full-screen modal dialog, and the user can move to the next or previous image by clicking buttons on either side of the current image. Making it possible to cycle through the images with the keyboard allows those without the ability to use a pointing device to use the carousel. It also adds convenience to the carousel for those who simply don't *want* to use a mouse or trackpad.

Just to keep this simple, let's say our image gallery HTML template looks like this:

```
<div class="image-gallery">
  <button class="previous" type="button">Previous</button>
  <img>
  <button class="next" type="button">Next</button>
</div>
```

And JavaScript to cycle through the images when a button is clicked looks like this (for modern browsers):

```
// assuming we have an array of images in an `images` var
var currentImageIndex = 0;
var container = document.querySelector('.image-gallery');
var updateImg = function() {
  container.querySelector('IMG').src =
    images[currentImageIndex];
};
var moveToPreviousImage = function() {
  if (currentImageIndex === 0) {
    currentImageIndex = images.length - 1;
  }
  else {
    currentImageIndex--;
  }
  updateImg();
};
var moveToNextImage = function() {
  if (currentImageIndex === images.length - 1) {
    currentImageIndex = 0;
  }
  else {
    currentImageIndex++;
  }
  updateImg();
};

```

```
updateImg();

container.querySelector('.previous')
  .addEventListener('click', function() {
    moveToPreviousImage();
  });

container.querySelector('.next')
  .addEventListener('click', function() {
    moveToNextImage();
  });
```

If we want to allow our users to move though the set of images with the left and right arrow keys, we can attach keyboard event handlers to the left and right arrows and delegate to the existing `moveToPreviousImage()` and `moveToNextImage()` functions as appropriate:

```
// add this after the block of code above:
document.addEventListener('keydown', function(event) {
  // left arrow
  if (event.which === 37) {
    event.preventDefault();
    moveToPreviousImage();
  }
  // right arrow
  else if (event.which === 39) {
    event.preventDefault();
    moveToNextImage();
  }
});
```

The addition of `event.preventDefault()` ensures that our arrow keys *only* change the image in this context instead of providing any undesired default actions, such as scrolling the page. In our example, it's not clear when our carousel is no longer in use, but we would likely provide some mechanism to close the carousel. Once the carousel is closed, don't forget to use `removeEventListener()` to unregister the keydown event handler. You'll need to refactor the event listener code such that the logic is moved into a standalone function. This will make it easy to unregister the keydown handler by passing `removeEventListener()` the 'keydown' event type as the first parameter, and the event listener function variable as the second. For more information on using `removeEventListener()`, check out the earlier section on observing events.

Determining When Something Has Loaded

The following questions may occur to you as a web developer at one point or another:

- When have all elements on the page fully loaded and rendered with applied styles?
- When has all static markup been placed on the page?
- When has a particular element on the page fully loaded? When has an element failed to load?

The answer to all of these questions lies in the browser's native event system. The "load" event,[23] defined in the W3C UI Events specification, allows us to determine when an element or page has loaded. There are some other related events, such as "DOMContentLoaded" and "beforeunload". I discuss both of those as well in this section.

When Have All Elements on the Page Fully Loaded and Rendered with Applied Styles?

To answer this particular question, we can rely on the "load" event fired by the `window` object. This event will be fired after:

1. All markup has been placed on the page.
2. All style sheets have been loaded.
3. All `<img>` elements have loaded.
4. All `<iframe>` elements have fully loaded.

jQuery provides an alias for the "load" event, similar to many other DOM events:

```
$(window).load(function() {
  // page is fully rendered
});
```

But you can (and should) use the generic on() method instead and pass in the name of the event - "load" - especially since the load() alias is deprecated and was subsequently removed in jQuery 3.0. Here's the same listener without the deprecated alias:

```
$(window).on('load', function() {
  // page is fully rendered
});
```

The web API solution looks almost exactly like the preceding jQuery code. We're making using of `addEventListener()`, which is available to all modern browsers, and passing the name of the event followed by a callback to be invoked when the page has been loaded:

```
window.addEventListener('load', function() {
  // page is fully rendered
});
```

But why might we care about this event? Why is it important to know when the page has entirely loaded? The most common understanding of the load event is that it should be used to determine when it is safe to perform DOM manipulation. This is technically true, but waiting for the "load" event is probably unnecessary. Do you really need to ensure all images, style sheets, and iframes have loaded before operating on the document? Probably not.

[23] www.w3.org/TR/uievents/#event-type-load

When Has All Static Markup Been Placed on the Page?

Another question we can ask here: when is the earliest point at which I can safely operate on the DOM? The answer to this and the question in this heading here is the same: wait for the browser to fire the "DOMContentLoaded" event. This event fires after all markup has been placed on the page, which means it often occurs much sooner than "load".

jQuery provides a "ready" function that mirrors the behavior of the native "DOMContentLoaded". But under the covers, it delegates to "DOMContentLoaded" itself in modern browsers. Here's how you have been determining when a page is ready for interaction with jQuery:

```
$(document).ready(function() {
  // markup is on the page
});
```

You may even be familiar with the shorthand version:

```
$(function() {
  // markup is on the page
});
```

Because jQuery just makes use of the browser's native "DOMContentLoaded" event in modern browsers to feed its `ready` API method, we can build our own `ready` using "DOMContentLoaded" with `addEventListener()`:

```
document.addEventListener('DOMContentLoaded', function() {
  // markup is on the page
});
```

Note that you will likely want to ensure your script that registers the "DOMContentLoaded" event handler is placed before any style sheet `<link>` tags, since loading these style sheets will block any script execution and prolong the "DOMContentLoaded" event until the defined style sheets have completely loaded.

When Has a Particular Element on the Page Fully Loaded? When Has It Failed to Load?

In addition to `window`, load events are associated with a number of elements, such as `<img>`, `<link>`, and `<script>`. The most common use of this event outside of `window` is to determine when a specific image has loaded. The appropriately named "error" event is used to signal a failure to load an image (or a `<link>` or `<script>`, for example).

jQuery, as you might expect, has aliases in its API for the "load" and "error" events, but these are both deprecated and were subsequently removed from the library in jQuery 3.0. So to determine if an image has loaded, or has failed to load with jQuery, we should simply rely on the on() method. And our code would look something like this:

```
$('IMG').on('load', function() {
  // image has successfully loaded
});

$('IMG').on('error', function() {
  // image has failed to load
});
```

There is a one-to-one mapping between the event names used in jQuery and those used in the browser's native event system. As you might expect, jQuery relies on the browser's "load" and "error" events to signal success and failure, respectively. So, the same end can be reached without jQuery by registering for these events with `addEventListener()`:

```
document.querySelector('IMG').addEventListener('load', function() {
  // image has successfully loaded
});

document.querySelector('IMG').addEventListener('error', function() {
  // image has failed to load
});
```

As we've seen many times before, the syntax between jQuery and the web API here for modern browsers is strikingly similar.

Preventing a User from Accidentally Leaving the Current Page

Imagine yourself as a user (we can't always be the developer). You're filling out a (long) series of form fields. It takes ten minutes to complete the form, but you're finally done. And . . . then . . . you . . . accidentally . . . close . . . the browser tab. All your work is gone! Replace "filling out a form" with "writing a document" or "drawing a picture." Regardless of the situation, it's a tragic turn of events. As a developer, how can you save your users from this mistake? Can you? You can!

The "beforeunload" event is fired on the `window` just before the current page is unloaded. By observing this event, you can force the user to confirm that they really to want to leave the current page, or close the browser, or reload the page. They will be presented with a confirm dialog, and if they select Cancel, they will remain safely on the current page.

In jQuery-land, you can observe this event using the `on` API method and return a message for the user to be displayed in the confirm dialog:

```
$(window).on('beforeunload', function() {
  return 'Are you sure you want to unload the page?';
});
```

Note that not every browser will display this specific message. Some will always display a hard-coded message, and there is nothing that jQuery can do about this.

The web API approach is similar, but we must deal with a small difference in implementation of this event between various browsers:

```
window.addEventListener('beforeunload', function(event) {
  var message = 'Are you sure you want to unload the page?';
  event.returnValue = message;
  return message;
});
```

Some browsers accept the return value of the handler function as the text to display to the user, whereas others take a more non-standard approach and require this message be set on the event's `returnValue` property. Still, this isn't much of a hoop to jump through. It's pretty simple either way.

A History Lesson: Ancient Browser Support

This final section in the events chapter is where I describe a time when jQuery *was* a required library for web applications. This section applies to ancient browsers only. Back when it was common to support Internet Explorer 8, the web API was a bit of a mess in *some* instances. This was especially true when dealing with the browser's event system. In this section, I discuss how you can manage, observe, and fire events in ancient browsers. Because ancient browsers are becoming less important to worry about, all of this serves as more of a history lesson than a tutorial. Please keep that in mind when reading this section, as it is not intended to be a comprehensive guide to event handling in super-old browsers.

The API for Listening to Events Is Non-standard

Take a look at the following code snippet that registers for a click event:

```
someElement.attachEvent('onclick', function() {
  // do something with the click event...
});
```

You'll notice two distinct differences between this and the modern browser approach:

1. We are relying on `attachEvent` instead of `addEventListener`.
2. The click event name includes a prefix of "on".

The `attachEvent()` method[24] is proprietary to Microsoft's Internet Explorer. In fact, it is still technically supported up until (and including) Internet Explorer 10. `attachEvent()` was never part of any official standard. Unless you must support IE8 and older, avoid using `attachEvent()` entirely. The W3C standardized `addEventListener()` provides a more elegant and comprehensive solution for observing events.

Perhaps you're wondering how you can programmatically use the correct event-handling method based on the current browser's capabilities. If you're developing apps exclusively for modern browsers, this isn't a concern. But if, for some reason, you must target ancient browsers such as IE8 (or older), you can use the following code to register for an event in *any* browser:

```
function registerHandler(target, type, callback) {
  var listenerMethod = target.addEventListener
        || target.attachEvent,

      eventName = target.addEventListener
        ? type
        : 'on' + type;

  listenerMethod(eventName, callback);
}

// example use
registerHandler(someElement, 'click', function() {
  // do something with the click event...
});
```

[24] https://msdn.microsoft.com/en-us/library/ms536343(VS.85).aspx

And if you want to *remove* an event handler in an ancient browser, you must use `detachEvent()` instead of `removeEventListener()`. `detachEvent()` is another non-standard proprietary web API method. If you're looking for a cross-browser way to remove an event listener, try this out:

```
function unregisterHandler(target, type, callback) {
  var removeMethod = target.removeEventListener
        || target.detachEvent,

      eventName = target.removeEventListener
        ? type
        : 'on' + type;

  removeMethod(eventName, callback);
}

// example use
unregisterHandler(someElement, 'click', someEventHandlerFunction);
```

Form Field Change Events Are a Minefield

Very old versions of Internet Explorer have some serious change-event deficiencies. Here are the two big ones that you may come across (if you haven't already):

1. Change events in old versions of IE do *not* bubble.
2. Checkboxes and radio buttons *may* not trigger a change event at all in old versions of IE.

Keep in mind that the second issue was *also* reproducible when using jQuery with IE7 and 8 for quite a long time. As far as I can tell, current versions of jQuery do properly address this issue. But this is yet another reminder that jQuery is not without its own bugs.

To solve the change event issue, you must attach a change handler *directly* to *any* form field that you'd like to monitor, since event delegation is not possible. In order to tackle the checkbox and radio buttons conundrum, your best bet may be to attach a click handler directly to radio/checkbox fields (or attach the handler to a parent element and make use of event delegation) instead of relying on the change event to occur at all.

■ **Note** Generally, the enter key will fire a click event (for example on a button). In other words, click events are not only fired by a pointing device. For accessibility reasons, activating an element via the keyboard will also trigger a click event. In the case of a checkbox or radio button input element, the "enter" key will not activate the form field. Instead, "spacebar" key is required to activate a checkbox or radio button and trigger a click event.

The Event Object Is Also Non-standard

Some properties of the `Event` object instance are a bit different in older browsers. For example, although the target of an event in modern browsers can be found by checking the `target` property of the `Event` instance, IE8 and older contain a different property for this element: `srcElement`. The relevant portion of a cross-browser event handler function may look like this:

```
function myEventHandler(event) {
  var target = event.target || event.srcElement
  // ...
}
```

In modern browsers, the `event.target` will be truthy, which short-circuits the conditional evaluation just given. But in IE8 and older, the `Event` object instance will not contain a `target` property, so the `target` variable will be the value of the `srcElement` property on the `Event`.

In terms of controlling your events, the `stopPropagation()` method is *not* available on an `Event` object instance in IE8 and older. If you want to stop an event from bubbling, you must instead set the non-standard `cancelBubble` property on the `Event` instance. A cross-browser solution looks like this:

```
function myEventHandler(event) {
  if (event.stopPropgation) {
      event.stopPropagation();
  }
  else {
      event.cancelBubble = true;
  }
}
```

IE8 and older also do not have a `stopImmediatePropagation()` method. There isn't much that can be done to work around this limitation. However, I personally don't see lack of this method as a big problem. Using `stopImmediatePropagation()` seems like a code smell to me since the behavior of this call is completely dependent on the order that multiple event handlers are attached to the element in question.

The important takeaway for this chapter: events are pretty simple to work with in modern browsers without jQuery, but if you are unlucky enough to support Internet Explorer 8 or older, consider using the cross-browser functions demonstrated here, or pull in a reliable events library for more complex event handling requirements.