6
EVENTS

When you browse the web, your browser registers different types of events. It's the browser's way of saying, "Hey, this just happened." Your script can then respond to these events.

Scripts often respond to these events by updating the content of the web page (via the Document Object Model) which makes the page feel more interactive. In this chapter, you will learn how:

INTERACTIONS CREATE EVENTS

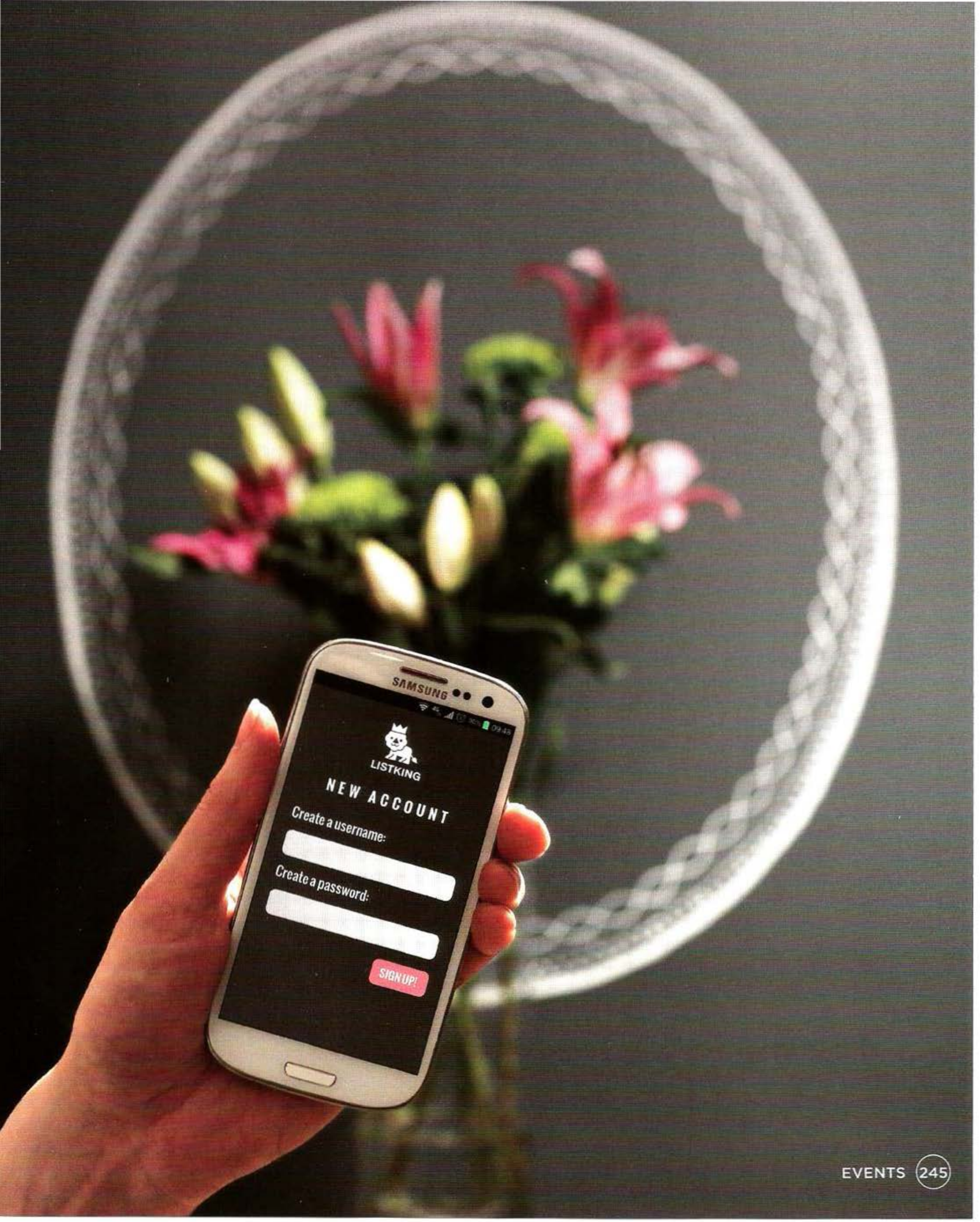
Events occur when users click or tap on a link, hover or swipe over an element, type on the keyboard, resize the window, or when the page they requested has loaded.

EVENTS TRIGGER CODE

When an event occurs, or fires, it can be used to trigger a particular function. Different code can be triggered when users interact with different parts of the page.

CODE RESPONDS TO USERS

In the last chapter, you saw how the DOM can be used to update a page. The events can trigger the kinds of changes the DOM is capable of. This is how a web page reacts to users.


SAMSUNG
LISTKING
NEW ACCOUNT
Create a username:
Create a password:
SIGN UP!

DIFFERENT EVENT TYPES

Here is a selection of the events that occur in the browser while you are browsing the web. Any of these events can be used to trigger a function in your JavaScript code.

UI EVENTS

Occur when a user interacts with the browser's user interface (UI) rather than the web page

EVENT	DESCRIPTION
`load`	Web page has finished loading
`unload`	Web page is unloading (usually because a new page was requested)
`error`	Browser encounters a JavaScript error or an asset doesn't exist
`resize`	Browser window has been resized
`scroll`	User has scrolled up or down the page

KEYBOARD EVENTS

Occur when a user interacts with the keyboard (see also `input` event)

EVENT	DESCRIPTION
`keydown`	User first presses a key (repeats while key is depressed)
`keyup`	User releases a key
`keypress`	Character is being inserted (repeats while key is depressed)

MOUSE EVENTS

Occur when a user interacts with a mouse, trackpad, or touchscreen

EVENT	DESCRIPTION
`click`	User presses and releases a button over the same element
`dblclick`	User presses and releases a button twice over the same element
`mousedown`	User presses a mouse button while over an element
`mouseup`	User releases a mouse button while over an element
`mousemove`	User moves the mouse (not on a touchscreen)
`mouseover`	User moves the mouse over an element (not on a touchscreen)
`mouseout`	User moves the mouse off an element (not on a touchscreen)

TERMINOLOGY

EVENTS FIRE OR ARE RAISED

When an event has occurred, it is often described as having **fired** or been **raised**. In the diagram on the right, if the user is tapping on a link, a `click` event would fire in the browser.

EVENTS TRIGGER SCRIPTS

Events are said to **trigger** a function or script. When the `click` event fires on the element in this diagram, it could trigger a script that enlarges the selected item.

FOCUS EVENTS

Occur when an element (e.g., a link or form field) gains or loses focus

EVENT	DESCRIPTION
`focus / focusin`	Element gains focus
`blur / focusout`	Element loses focus

FORM EVENTS

Occur when a user interacts with a form element

EVENT	DESCRIPTION
`input`	Value in any `<input>` or `<textarea>` element has changed (IE9+) or any element with the `contenteditable` attribute
`change`	Value in select box, checkbox, or radio button changes (IE9+)
`submit`	User submits a form (using a button or a key)
`reset`	User clicks on a form's reset button (rarely used these days)
`cut`	User cuts content from a form field
`copy`	User copies content from a form field
`paste`	User pastes content into a form field
`select`	User selects some text in a form field

MUTATION EVENTS*

Occur when the DOM structure has been changed by a script
* To be replaced by mutation observers (see p284)

EVENT	DESCRIPTION
`DOMSubtreeModified`	Change has been made to document
`DOMNodeInserted`	Node has been inserted as a direct child of another node
`DOMNodeRemoved`	Node has been removed from another node
`DOMNodeInsertedIntoDocument`	Node has been inserted as a descendant of another node
`DOMNodeRemovedFromDocument`	Node has been removed as a descendant of another node

HOW EVENTS TRIGGER JAVASCRIPT CODE

When the user interacts with the HTML on a web page, there are three steps involved in getting it to trigger some JavaScript code. Together these steps are known as **event handling**.

1

Select the **element** node(s) you want the script to respond to.

For example, if you want to trigger a function when a user clicks on a specific link, you need to get the DOM node for that link element. You do this using a DOM query (see Chapter 5).

2

Indicate which **event** on the selected node(s) will trigger the response.

Programmers call this **binding** an event to a DOM node.

The previous two pages showed a selection of the popular events that you can monitor for.

3

State the **code** you want to run when the event occurs.

When the event occurs, on a specified element, it will trigger a function. This may be a named or an anonymous function.

The UI events that relate to the browser window (rather than the HTML page loaded in it) work with the `window` object rather than an element node. Examples include the events that occur when a requested page has finished loading, or when the user scrolls. You will learn about using these on p272.

Some events work with most element nodes, such as the `mouseover` event, which is triggered when the user rolls over any element. Other events only work with specific element nodes, such as the `submit` event, which only works with a form.

Here you can see how event handling can be used to provide feedback to users filling in a registration form. It will show an error message if their username is too short.

1 SELECT ELEMENT

The element that users are interacting with is the text input where they enter the username.

2 SPECIFY EVENT

When users move out of the text input, it loses focus, and the `blur` event fires on this element.

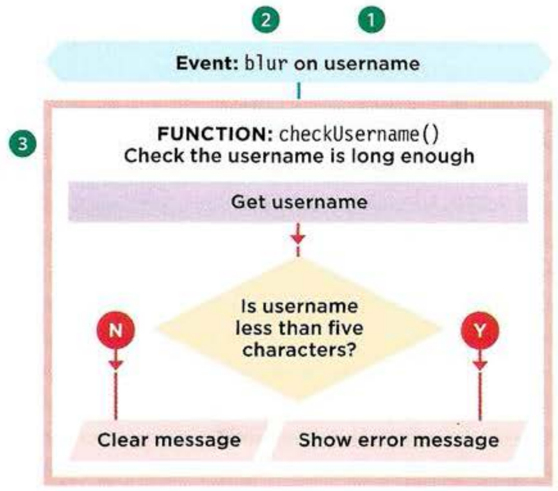
3 CALL CODE

When the `blur` event fires on the username input, it will trigger a function called `checkUsername()`. This function checks if the username is less than 5 characters.

If there are not enough characters, it shows an error message that prompts the user to enter a longer username.

If there *are* enough characters, the element that holds the error message should be cleared.

This is because an error message may have been shown to the user already and they subsequently corrected their mistake. (If the error message was still visible when they had filled in the form correctly, it would be confusing.)


2
1
Event: blur on username
3
FUNCTION: checkUsername()
Check the username is long enough
Get username
Is username less than five characters?
N
Y
Clear message
Show error message

THREE WAYS TO BIND AN EVENT TO AN ELEMENT

Event handlers let you indicate which event you are waiting for on any particular element. There are three types of event handlers.

HTML EVENT HANDLERS

See p251

This is bad practice, but you need to be aware of it because you may see it in older code.

Early versions of HTML included a set of attributes that could respond to events on the element they were added to. The attribute names matched the event names. Their values called the function that was to run when that event occurred.

For example, the following: `<a onclick="hide()">` indicated that when a user clicked on this `<a>` element, the `hide()` function would be called.

This method of event handling is no longer used because it is better to separate the JavaScript from the HTML. You should use one of the other approaches shown on this page instead.

TRADITIONAL DOM EVENT HANDLERS

See p252

DOM **event handlers** were introduced in the original specification for the DOM. They are considered better than HTML event handlers because they let you separate the JavaScript from the HTML.

Support in all major browsers is very strong for this approach. The main drawback is that you can only attach a single function to any event. For example, the submit event of a form cannot trigger one function that checks the contents of a form, and a second to submit the form data if it passes the checks.

As a result of this limitation, if more than one script is used on the same page, and both scripts respond to the same event, then one or both of the scripts may not work as intended.

DOM LEVEL 2 EVENT LISTENERS

See p254

Event listeners were introduced in an update to the DOM specification (DOM level 2, released in the year 2000). They are now the favored way of handling events.

The syntax is quite different and, unlike traditional event handlers, these newer event listeners allow one event to trigger multiple functions. As a result, there are less likely to be conflicts between different scripts that run on the same page.

This approach does not work with IE8 (or earlier versions of IE) but you meet a workaround on p258. Differences in browser support for the DOM and events helped speed adoption of jQuery (but you need to know how events work to understand how jQuery uses them).

HTML EVENT HANDLER ATTRIBUTES (DO NOT USE)

Please note: This approach is now considered bad practice; however, you need to be aware of it because you may see it if you are looking at older code. (See previous page.)

In the HTML, the first `<input>` element has an attribute called `onblur` (triggered when the user leaves the element). The value of the attribute is the name of the function that it should trigger.

The value of the event handler attributes would be JavaScript. Often it would call a function that was written either in the `<head>` element or a separate JavaScript file (as shown below).

HTML c06/event-attributes.html

```
<form method="post" action="http://www.example.org/register">
  <label for="username">Create a username: </label>
  <input type="text" id="username" onblur="checkUsername()" />
  <div id="feedback"></div>

  <label for="password">Create a password: </label>
  <input type="password" id="password" />

  <input type="submit" value="Sign up!" />
</form>
...
<script type="text/javascript" src="js/event-attributes.js"></script>
```

JAVASCRIPT c06/js/event-attributes.js

```
function checkUsername() {                              // Declare function
  var elMsg = document.getElementById('feedback');      // Get feedback element
  var elUsername = document.getElementById('username');// Get username input
  if (elUsername.value.length < 5) {                    // If username too short
    elMsg.textContent = 'Username must be 5 characters or more'; // Set msg
  } else {                                               // Otherwise
    elMsg.textContent = '';                              // Clear message
  }
}
```

The names of the HTML event handler attributes are identical to the event names shown on p246 - p247, preceded by the word "on."

For example:

- `<a>` elements can have `onclick`, `onmouseover`, `onmouseout`
- `<form>` elements can have `onsubmit`
- `<input>` elements for text can have `onkeypress`, `onfocus`, `onblur`

TRADITIONAL DOM EVENT HANDLERS

All modern browsers understand this way of creating an event handler, but you can only attach one function to each event handler.

Here is the syntax to bind an event to an element using an event handler, and to indicate which function should execute when that event fires:

```
element.onevent = functionName;
```

ELEMENT	EVENT	CODE
DOM element node to target	Event bound to node(s) preceded by word "on"	Name of function to call (with no parentheses following it)

Below, the event handler is on the last line (after the function has been defined and the DOM element node(s) selected).

When a function is called, the parentheses that follow its name tell the JavaScript interpreter to "run this code now."

We don't want the code to run until the event fires, so the parentheses are omitted from the event handler on the last line.

```
function checkUsername() {
  // code to check the length of username
}
var el = document.getElementById('username');
el.onblur = checkUsername;
```

The code starts by defining the named function.

A reference to the DOM element node is often stored in a variable.

The function is called by the event handler on the last line, but the parentheses are omitted.

The event name is preceded by the word "on."

An example of an anonymous function and a function with parameters is shown on p256.

USING DOM EVENT HANDLERS

In this example, the event handler appears on the last line of the JavaScript. Before the DOM event handler, two things are put in place:

1. If you use a named function when the event fires on your chosen DOM node, write that function first. (You could also use an anonymous function.)

2. The DOM element node is stored in a variable. Here the text input (whose `id` attribute has a value of `username`) is placed into a variable called `elUsername`.

JAVASCRIPT

c06/js/event-handler.js

```
①  function checkUsername() {                                        // Declare function
      var elMsg = document.getElementById('feedback');               // Get feedback element
      if (this.value.length < 5) {                                   // If username too short
        elMsg.textContent = 'Username must be 5 characters or more';  // Set msg
      } else {                                                       // Otherwise
        elMsg.textContent = '';                                      // Clear message
      }
    }

② var elUsername = document.getElementById('username');  // Get username input
③ elUsername.onblur = checkUsername;   // When it loses focus call checkuserName()
```

When using event handlers, the event name is preceded by the word "on" (`onsubmit`, `onchange`, `onfocus`, `onblur`, `onmouseover`, `onmouseout`, etc).

3. On the last line of the code sample above, the event handler `elUsername.onblur` indicates that the code is waiting for the `blur` event to fire on the element stored in the variable called `elUsername`.

This is followed by an equal sign, then the name of the function that will run when the event fires on that element. Note that there are no parentheses on the function name. This means you cannot pass arguments to this function. (If you want to pass arguments to a function in an event handler, see p256.)

The HTML is the same as that shown on p251 but without the `onblur` event attribute. This means that the event handler is in the JavaScript, not the HTML.

Browser support: On line 3, the `checkUsername()` function uses the `this` keyword in the conditional statement to check the number of characters the user entered. It works in most browsers because they know `this` refers to the element the event happened on.

However, in Internet Explorer 8 or earlier, IE would treat `this` as the `window` object. As a result, it would not know which element the event occurred on and there would be no value that it checked the length of, so it would raise an error. You will learn a solution for this issue on p264.

EVENT LISTENERS

Event listeners are a more recent approach to handling events. They can deal with more than one function at a time but they are not supported in older browsers.

Here is the syntax to bind an event to an element using an event listener, and to indicate which function should execute when that event fires:

Adds an event listener to the DOM element node(s)

METHOD

```
element.addEventListener('event', functionName [, Boolean]);
```

ELEMENT
DOM element node to target

EVENT
Event to bind node(s) to in quote marks

CODE
Name of function to call

EVENT FLOW
Indicates something called capture, and is usually set to `false` (see p260)

```
function checkUsername() {
  // code to check the length of username
}
var el = document.getElementById('username');
el.addEventListener('blur', checkUsername, false);
```

The code starts by defining the named function.

A reference to the DOM element node is often stored in a variable.

The function is called by the event listener on the last line, but the parentheses are omitted.

The event name is enclosed in quotation marks.

An example of an anonymous function and a function with parameters is shown on p256.

USING EVENT LISTENERS

In this example, the event listener appears on the last line of the JavaScript. Before you write an event listener, two things are put in place:

1. If you use a named function when the event fires on your chosen DOM node, write that function first. (You could also use an anonymous function.)

2. The DOM element node(s) is stored in a variable. Here the text input (whose `id` attribute has a value of `username`) is placed into a variable called `elUsername`.

JAVASCRIPT

c06/js/event-listener.js

```
  function checkUsername() {                                  // Declare function
    var elMsg = document.getElementById('feedback');          // Get feedback element
    if (this.value.length < 5) {                              // If username too short
①     elMsg.textContent = 'Username must be 5 characters or more'; // Set msg
    } else {                                                  // Otherwise
      elMsg.textContent = '';                                 // Clear msg
    }
  }

② var elUsername = document.getElementById('username');  // Get username input
  // When it loses focus call checkUsername()
  elUsername.addEventListener('blur', checkUsername, false);
                                 (i)           (ii)        (iii)
```

The `addEventListener()` method takes three properties:

i) The event you want it to listen for. In this case, the `blur` event.

ii) The code that you want it to run when the event fires. In this example, it is the `checkUsername()` function. Note that the parentheses are omitted where the function is called because they would indicate that the function should run as the page loads (rather than when the event fires).

iii) A Boolean indicating how events flow, see p260. (This is usually set to `false`.)

BROWSER SUPPORT

Internet Explorer 8 and earlier versions of IE do not support the `addEventListener()` method, but they do support a method called `attachEvent()` and you will see how to use this on p258.

Also, as with the previous example, IE8 and older versions of IE would not know what `this` referred to in the conditional statement. An alternative approach for dealing with it is shown on p270.

EVENT NAMES

Unlike the HTML and traditional DOM event handlers, when you specify the name of the event that you want to react to, the event name is not preceded by the word "on".

If you need to remove an event listener, there is a function called `removeEventListener()` which removes the event listener from the specified element (it has the same parameters).

USING PARAMETERS WITH EVENT HANDLERS & LISTENERS

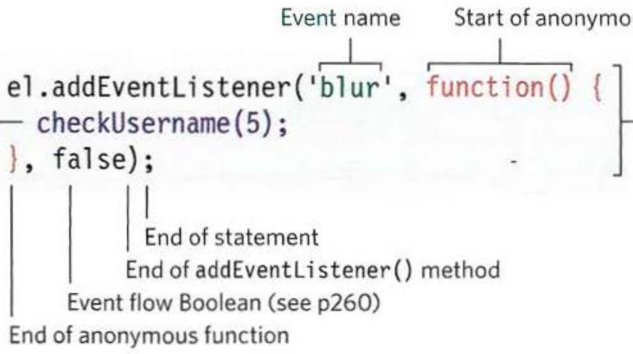
Because you cannot have parentheses after the function names in event handlers or listeners, passing arguments requires a workaround.

Usually, when a function needs some information to do its job, you pass arguments within the parentheses that follow the function name.

When the interpreter sees the parentheses after a function call, it runs the code straight away. In an event handler, you want it to wait until the event triggers it.

Therefore, if you need to pass arguments to a function that is called by an event handler or listener, you wrap the function call in an **anonymous function**.

The named function includes parentheses containing the parameter after the function name.



```
el.addEventListener('blur', function() {
  checkUsername(5);
}, false);
```

Event name
Start of anonymous function
End of statement
End of addEventListener() method
Event flow Boolean (see p260)
End of anonymous function

The anonymous function is used as the second argument. It "wraps around" the named function.

The named function that requires the arguments lives inside the anonymous function.

Although the anonymous function has parentheses, it only runs when the event is triggered.

The named function can use arguments as it only runs if the anonymous function is called.

USING PARAMETERS WITH EVENT LISTENERS

The first line of this example shows the updated checkUsername() function. The minLength parameter specifies the minimum number of characters that the username should be.

The value that is passed into the checkUsername() function is used in the conditional statement to check if the name is long enough, and provide feedback if the username name is too short.

JAVASCRIPT

c06/js/event-listener-with-parameters.js

```
var elUsername = document.getElementById('username');  // Get username input
var elMsg = document.getElementById('feedback');        // Get feedback element

function checkUsername(minLength) {                     // Declare function
  if (elUsername.value.length < minLength) {            // If username too short
   // Set the error message
   elMsg.textContent = 'Username must be ' + minLength + ' characters or more';
  } else {                                              // Otherwise
   elMsg.innerHTML = '';                                // Clear msg
  }
}

elUsername.addEventListener('blur', function() {        // When it loses focus
  checkUsername(5);                                      // Pass arguments here
}, false);
```

The event listener on the last three lines is longer than the previous example because the call to the checkUsername() function needs to include the value for the minLength parameter.

To receive this information, the event listener uses an anonymous function, which acts like a wrapper. Inside that wrapper the checkUsername() function is called, and passed an argument.

Browser support: On the next page you also see how to deal with the lack of support for event listeners in IE8 and earlier.

SUPPORTING OLDER VERSIONS OF IE

IE5-8 had a different event model and did not support `addEventListener()` but you can provide fallback code to make event listeners work with older versions of IE.

IE5-IE8 did not support the `addEventListener()` method. Instead, it used its own method called `attachEvent()` which did the same job, but was only available in Internet Explorer. If you want to use event listeners and need to support Internet Explorer 8 or earlier, you can use a conditional statement as illustrated below.

Using an `if...else` statement, you can check if the browser supports the `addEventListener()` method. The condition in the `if` statement will return `true` if the browser supports the `addEventListener()` method, and you can use it. If the browser does not support that method, it returns `false`, and the code will try to use the `attachEvent()` method.

If the browser supports `addEventListener()`:

Run the code inside these curly braces

If it doesn't, do something else:

Run the code inside these curly braces

```
if (el.addEventListener) {
  el.addEventListener('blur', function() {
    checkUsername(5);
  }, false );
} else {
  el.attachEvent('onblur', function() {
    checkUsername(5);
  });
}
```

When `attachEvent()` is used, the event name should be preceded by the word "on" (e.g., `blur` becomes `onblur`). You will see another approach to supporting the older IE event model in Chapter 13 (using a utility file).

FALLBACK FOR USING EVENT LISTENERS IN IE8

The event handling code builds on the last example, but it is a lot longer this time because it contains the fallback for Internet Explorer 5-8.

After the `checkUsername()` function, an `if` statement checks whether `addEventListener()` is supported or not; it returns `true` if the element node supports this method, and `false` if it does not.

If the browser supports the `addEventListener()` method, the code inside the first set of curly braces is run using `addEventListener()`.

If it is not supported, then the browser will use the `attachEvent()` method that older versions of IE will understand. In the IE version, note that the event name must be preceded by the word "on."

JAVASCRIPT c06/js/event-listener-with-ie-fallback.js

```
var elUsername = document.getElementById('username');  // Get username input
var elMsg = document.getElementById('feedback');        // Get feedback element

function checkUsername(minLength) {                     // Declare function
  if (elUsername.value.length < minLength) {            // If username too short
    // Set message
    elMsg.innerHTML = 'Username must be ' + minLength + ' characters or more';
  } else {                                              // Otherwise
    elMsg.innerHTML = '';                               // Clear message
  }
}

if (elUsername.addEventListener) {                 // If event listener supported
  elUsername.addEventListener('blur', function(){// When username loses focus
    checkUsername(5);                              // Call checkUsername()
  }, false );                                      // Capture during bubble phase
} else {                                           // Otherwise
  elUsername.attachEvent('onblur', function(){    // IE fallback: onblur
    checkUsername(5);                              // Call checkUsername()
  });
}
```

If you need to support IE8 (or older), instead of writing this fallback code for *every* event you are responding to, it is better to write your own function (known as a helper function) that creates the appropriate event handler for you. You will see a demonstration of this in Chapter 13, which covers form enhancement and validation.

It is, however, important to understand this syntax, used by IE8 (and older) so that you know why the helper function is used and what it is doing.

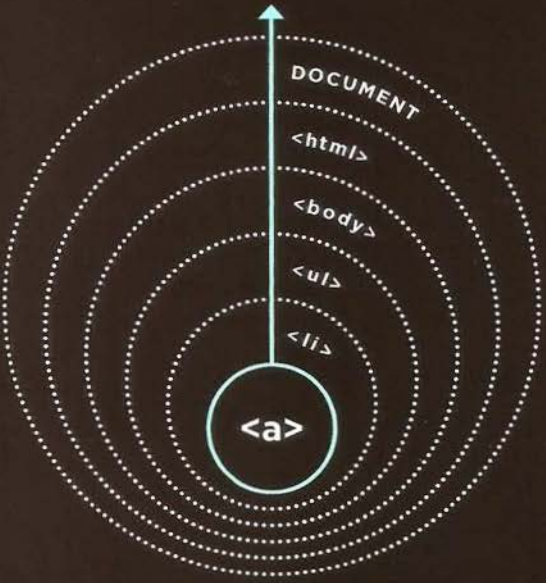
As you will see in the next chapter, this is another type of cross-browser inconsistency that jQuery can take care of for you.

EVENT FLOW

HTML elements nest inside other elements. If you hover or click on a link, you will also be hovering or clicking on its parent elements.

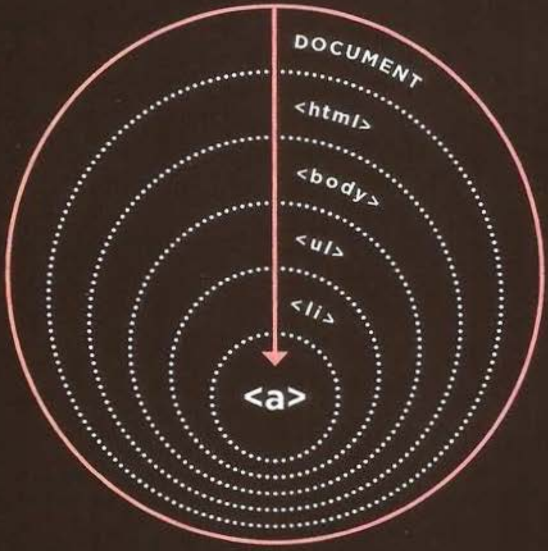
Imagine a list item contains a link. When you hover over the link or click on it, JavaScript can trigger events on the **<a>** element, and also any elements the **<a>** element sits inside.

Event handlers/listeners can be bound to the containing **`<li>`**, **`<ul>`**, **<body>**, and **`<html>`** elements, plus the **`document`** object, and the **`window`** object. The order in which the events fire is known as **event flow**, and events flow in two directions.



EVENT BUBBLING

The event starts at the *most* specific node and **flows outwards** to the *least* specific one. This is the default type of event flow with very wide browser support.



EVENT CAPTURING

The event starts at the *least* specific node and **flows inwards** to the *most* specific one. This is not supported in Internet Explorer 8 and earlier.

WHY FLOW MATTERS

The flow of events only really matters when your code has event handlers on an element *and* one of its ancestor or descendant elements.

The example below has event listeners that respond to the **click** event on each of the following elements:

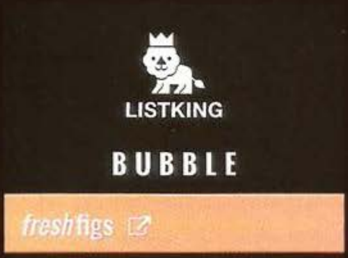
- One on the **<ul>** element
- One on the **<li>** element
- One on the **<a>** element in the list item

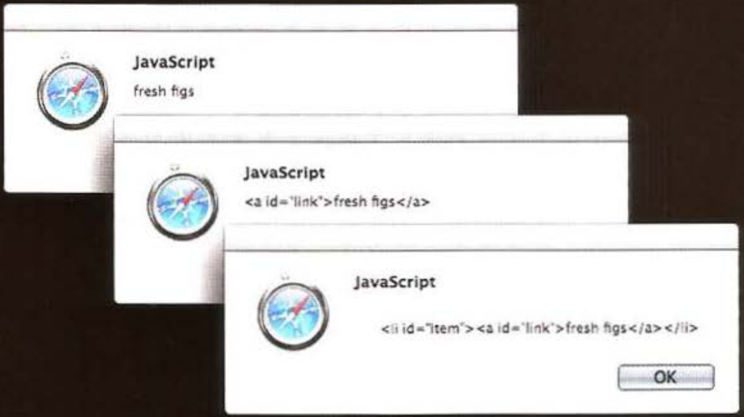
The event will show the HTML content of that element in an alert box, and event flow will tell you which element the click is registered upon first.

For traditional DOM event handlers (and HTML event attributes), all modern browsers default to using event bubbling rather than capturing. With event listeners, the final parameter in the **addEventListener()** method lets you choose the direction to trigger events:

- **true** = capturing phase
- **false** = bubbling phase (**false** is often a default choice because capturing was not supported in IE8 or earlier.)

The **event-flow.js** file (shown on the left, and available in the download code) demonstrates the difference between bubbling and capturing. In this example, the event handlers have a value of **false** for their last parameter indicating events should be followed in **bubbling** phase. So the first alert box shows the content of the innermost **<a>** element, and works its way out. You can also see the capturing version in the download code.




JavaScript
fresh figs

JavaScript
<a id="link">fresh figs</a>

JavaScript
<li id="item"><a id="link">fresh figs</a></li>

THE EVENT OBJECT

When an event occurs, the `event` object tells you information about the event, and the element it happened upon.

Every time an event fires, the `event` object contains helpful data about the event, such as:

- Which element the event happened on
- Which key was pressed for a `keypress` event
- What part of the viewport the user clicked for a `click` event (the viewport is the part of the browser window that shows the web page)

The `event` object is passed to any function that is the event handler or listener.

If you need to pass arguments to a named function, the `event` object will first be passed to the anonymous wrapper function (this happens automatically); then you must specify it as a parameter of the named function (as shown on the next page).

When the `event` object is passed into a function, it is often given the parameter name `e` (for event). It is a widely used shorthand (and you see it adopted throughout this book).

Note, however, that some programmers also use the parameter name `e` to refer to the `error` object; so `e` may mean `event` or `error` in some scripts.

Not only did IE8 have a different syntax for event listeners (as shown on p258), the `event` object in IE5-8 also had different names for the properties and methods shown in the tables below, and the example on p265.

PROPERTY	IE5-8 EQUIVALENT	PURPOSE
`target`	`srcElement`	The target of the event (most specific element interacted with)
`type`	`type`	Type of event that was fired
`cancelable`	not supported	Whether you can cancel the default behavior of an element

METHOD	IE5-8 EQUIVALENT PROPERTY	PURPOSE
`preventDefault()`	`returnValue`	Cancel default behavior of the event (if it can be canceled)
`stopPropagation()`	`cancelBubble`	Stops the event from bubbling or capturing any further

EVENT LISTENER WITH NO PARAMETERS

```
                                 ②
function checkUsername(e) {
③ var target = e.target; // get target of event
}

var el = document.getElementById('username');
el.addEventListener('blur', checkUsername, false);
                                   ③
```

1. Without you doing anything, a reference to the event object is automatically passed from the number 1, where the event listener calls the function...

2. To here, where the function is defined. At this point, the parameter must be named. It is often given the name e for event.

3. This name can then be used inside the function as a reference to the event object. You can now use the properties and methods of the event object.

EVENT LISTENER WITH PARAMETERS

```
                                 ③
function checkUsername(e, minLength) {
④ var target = e.target; // get target of event
}

var el = document.getElementById('username');
el.addEventListener('blur', function(e){ ①
  checkUsername(e, 5);
}, false);       ②
```

1. The reference to the event object is automatically passed to the anonymous function, but it must be named in the parentheses.

2. The reference to the event object can then be passed onto the named function. It is given as the first parameter of the named function.

3. The named function receives the reference to the event object as the first parameter of the method. 4. It can now be used by this name in the named function.

THE EVENT OBJECT IN IE5-8

Below you can see how you get the `event` object in IE5-8. It is *not* passed automatically to event handler/listener functions; but it *is* available as a child of the `window` object.

On the right, an `if` statement checks if the `event` object has been passed into the function. As you saw on p168, the existence of an object is treated as a truthy value, so the condition here is saying "if the event object *does not* exist..."

In IE8 and less, e will not hold an object, so the following code block runs and e is set to be the `event` object that is a child of the `window` object.

```
function checkUsername(e) {
  if (!e) {
    e = window.event;
  }
}
```

GETTING PROPERTIES

Once you have a reference to the `event` object, you can get its properties using the technique on the right. This works on short circuit evaluation (see p169).

```
var target;
target = e.target || e.srcElement;
```

A FUNCTION TO GET THE TARGET OF AN EVENT

If you need to assign event listeners to several elements, here is a function that will return a reference to the element the event happened on.

```
function getEventTarget(e) {
  if (!e) {
    e = window.event;
  }
  return e.target || e.srcElement;
}
```

USING EVENT LISTENERS WITH THE EVENT OBJECT

Here is the example that has been used throughout the chapter so far with some modifications:
1. The function is called `checkLength()` rather than `checkUsername()`. It can be used on any text input.
2. The event object is passed to the event listener. The code includes fallbacks for IE5-8 (Chapter 13 demonstrates using helper functions to do this).
3. In order to determine which element the user was interacting with, the function uses the event object's `target` property (and for IE5-8 it uses the equivalent `srcElement` property).

This function is now far more flexible than the previous code you have seen in this chapter because:
1. It can be used to check the length of any text input so long as that input is directly followed by an empty element that can hold a feedback message for the user. (There should not be space or carriage returns between the two elements; otherwise, some browsers might return a whitespace node.)
2. The code will work with IE5-8 because it tests whether the browser supports the latest features (or whether it needs to fallback to use older techniques).

JAVASCRIPT

c06/js/event-listener-with-event-object.js

```
function checkLength(e, minLength) {         // Declare function
  var el, elMsg;                             // Declare variables
  if (!e) {                                  // If event object doesn't exist
    e = window.event;                        // Use IE fallback
  }
  el = e.target || e.srcElement;             // Get target of event
  elMsg = el.nextSibling;                    // Get its next sibling

  if (el.value.length < minLength) {         // If length is too short set msg
    elMsg.innerHTML = 'Username must be ' + minLength + ' characters or more';
  } else {                                   // Otherwise
    elMsg.innerHTML = '';                    // Clear message
  }
}

var elUsername = document.getElementById('username');// Get username input
if (elUsername.addEventListener) {           // If event listener supported
  elUsername.addEventListener('blur', function(e) {  // On blur event
    checkUsername(e, 5);                             // Call checkUsername()
  }, false);                                         // Capture in bubble phase
} else {                                             // Otherwise
  elUsername.attachEvent('onblur', function(e){      // IE fallback onblur
    checkUsername(e, 5);                             // Call checkUsername()
  });
}
```

EVENT DELEGATION

Creating event listeners for a lot of elements can slow down a page, but event flow allows you to listen for an event on a parent element.

If users can interact with a lot of elements on the page, such as:

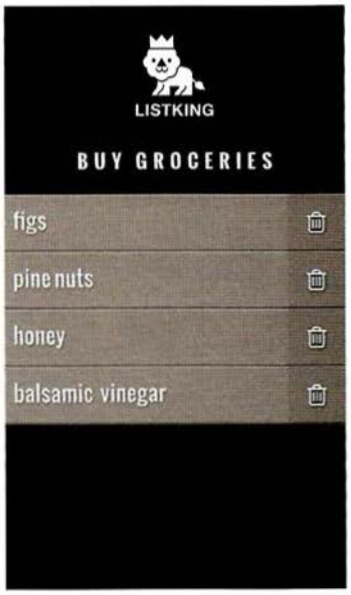
- a lot of buttons in the UI
- a long list
- every cell of a table

adding event listeners to each element can use a lot of memory and slow down performance.

Because events affect containing (or ancestor) elements (due to event flow - p260), you can place event handlers on a containing element and use the event object's `target` property to find which of its children the event happened on.

By attaching an event listener to a containing element, you are only responding to one element (rather than having an event handler for each child element).

You are delegating the job of the event listener to a parent of the elements. In the list shown here, if you place the event listener on the `<ul>` element rather than on links in each `<li>` element, you only need one event listener. This gives better performance, and if you add or remove items from the list it would still work the same. (The code for this example is shown on p269.)



ADDITIONAL BENEFITS OF EVENT DELEGATION

WORKS WITH NEW ELEMENTS

If you add new elements to the DOM tree, you do not have to add event handlers to the new elements because the job has been delegated to an ancestor.

SOLVES LIMITATIONS WITH `this` KEYWORD

Earlier in the chapter, the `this` keyword was used to identify an event's target, but that technique did not work in IE8, or when a function needed parameters.

SIMPLIFIES YOUR CODE

It requires fewer functions to be written, and there are fewer ties between the DOM and your code, which helps maintainability.

CHANGING DEFAULT BEHAVIOR

The `event` object has methods that change: the default behavior of an element and how the element's ancestors respond to the event.

`preventDefault()`

Some events, such as clicking on links and submitting forms, take the user to another page.

To prevent the default behavior of such elements (e.g., to keep the user on the same page rather than following a link or being taken to a new page after submitting a form), you can use the event object's `preventDefault()` method.

IE5-8 have an equivalent property called `returnValue` which can be set to `false`. A conditional statement can check if the `preventDefault()` method is supported, and use IE8's approach if it isn't:

```
if (event.preventDefault) {
  event.preventDefault();
} else {
  event.returnValue = false;
}
```

`stopPropagation()`

Once you have handled an event using one element, you may want to stop that event from bubbling up to its ancestor elements (especially if there are separate event handlers responding to the same events on the containing elements).

To stop the event bubbling up, you can use the `event` object's `stopPropogation()` method.

The equivalent in IE8 and earlier is the `cancelBubble` property which can be set to `true`. Again, a conditional statement can check if the `stopPropogation()` method is supported and use IE8's approach if not:

```
if (event.stopPropogation) {
  event.stopPropogation();
} else {
  event.cancelBubble = true;
}
```

USING BOTH METHODS

You will sometimes see the following used in similar situations that are in a function: `return false;`

It prevents the default behavior of the element, and prevents the event from bubbling up or capturing further. It also works in all browsers, so it is popular.

Note, however, when the interpreter comes across the `return false` statement, it stops processing any subsequent code within that function and moves to the next statement after the function was called.

Since this blocks any further code within the function, it is often better to use the `preventDefault()` method of the event object rather than `return false`.

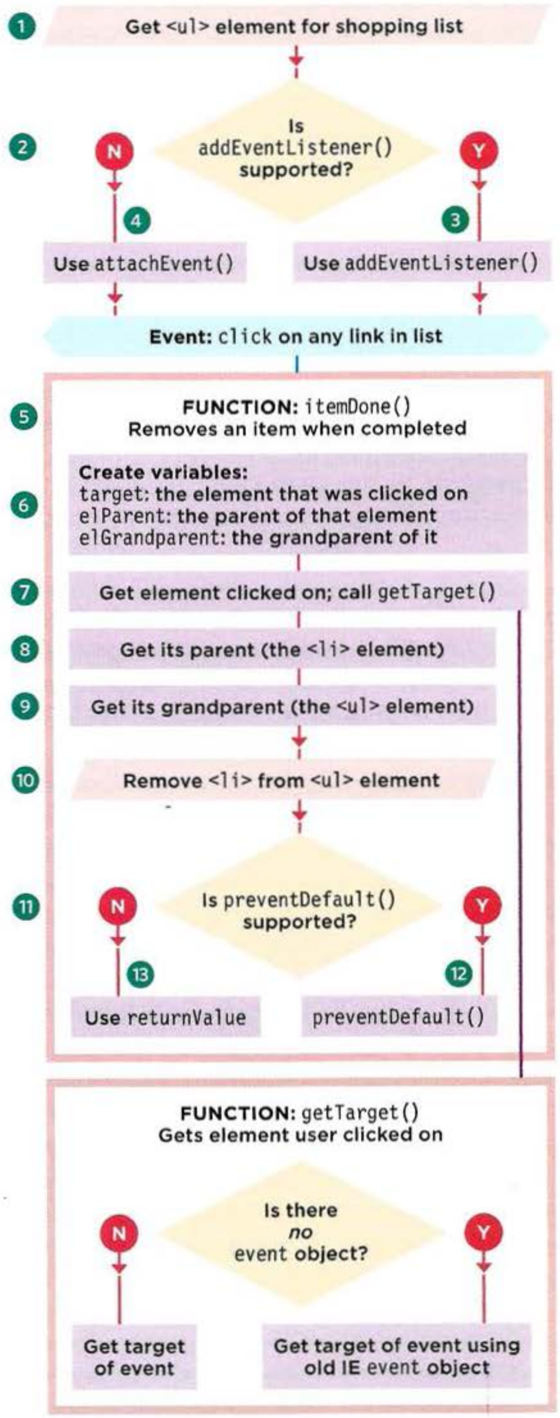
USING EVENT DELEGATION

This example will put together a lot of what you have learned in the chapter so far. Each list item contains a link. When the user clicks on that link (to indicate they have completed that task), the item will be removed from the list.

- There is a screen grab of the example on p266.
- On the right there is a flowchart that helps explain the order in which the code is processed.
- The right-hand page has the code for the example

1. The event listener will be added to the `<ul>` element, so this needs to be selected.
2. Check whether or not the browser supports `addEventListener()`.
3. If so, use it to call the `itemDone()` function when the user clicks anywhere on that list.
4. If not, use the `attachEvent()` method.
5. The `itemDone()` function will remove the item from the list. It requires three pieces of information.
6. Three variables are declared to hold the info.
7. `target` holds the element the user clicked on. To obtain this, the `getTarget()` function is called. This is created at the start of the script, and shown at the bottom of the flowchart.
8. `elParent` holds that element's parent (the `<li>`)
9. `elGrandparent` holds that element's grandparent
10. The `<li>` element is removed from the `<ul>` element.
11. Check if the browser supports `preventDefault()` to prevent the link taking the user to a new page.
12. If so, use it.
13. If not, use the older IE `returnValue` property.

In the HTML, the links would take you to `itemDone.php` if the browser did not support JavaScript. (The PHP file is not supplied with the code download because server-side languages are beyond the scope of this book.)



1
Get <ul> element for shopping list
2
N
Is addEventListener() supported?
Y
4
3
Use attachEvent()
Use addEventListener()
Event: click on any link in list
5
FUNCTION: itemDone()
Removes an item when completed
6
Create variables:
target: the element that was clicked on
elParent: the parent of that element
elGrandparent: the grandparent of it
7
Get element clicked on; call getTarget()
8
Get its parent (the <li> element)
9
Get its grandparent (the <ul> element)
10
Remove <li> from <ul> element
11
N
Is preventDefault() supported?
Y
13
12
Use returnValue
preventDefault()
FUNCTION: getTarget()
Gets element user clicked on
N
Is there no event object?
Y
Get target of event
Get target of event using old IE event object

HTML

c06/event-delegation.html

```
<ul id="shoppingList">
  <li class="complete"><a href="itemDone.php?id=1"><em>fresh</em> figs</a></li>
  <li class="complete"><a href="itemDone.php?id=2">pine nuts</a></li>
  <li class="complete"><a href="itemDone.php?id=3">honey</a></li>
  <li class="complete"><a href="itemDone.php?id=4">balsamic vinegar</a></li>
</ul>
```

JAVASCRIPT

c06/js/event-delegation.js

```
   function getTarget(e) {                          // Declare function
     if (!e) {                                      // If there is no event object
      e = window.event;                             // Use old IE event object
     }
     return e.target || e.srcElement;               // Get the target of event
   }

⑤  function itemDone(e) {                           // Declare function
     // Remove item from the list
⑥    var target, elParent, elGrandparent;           // Declare variables
⑦    target = getTarget(e);                         // Get the item clicked link
⑧    elParent = target.parentNode;                  // Get its list item
⑨    elGrandparent = target.parentNode.parentNode;  // Get its list
⑩    elGrandparent.removeChild(elParent);           // Remove list item from list

     // Prevent the link from taking you elsewhere
⑪    if (e.preventDefault) {                        // If preventDefault() works
⑫      e.preventDefault();                          // Use preventDefault()
     } else {                                       // Otherwise
⑬      e.returnValue = false;                       // Use old IE version
     }
   }

   // Set up event listeners to call itemDone() on click
①  var el = document.getElementById('shoppingList');// Get shopping list
②  if (el.addEventListener) {                       // If event listeners work
③    el.addEventListener('click', function(e) {     // Add listener on click
       itemDone(e);                                 // It calls itemDone()
     }, false);                                     // Use bubbling phase for flow
   } else {                                         // Otherwise
④    el.attachEvent('onclick', function(e){         // Use old IE model: onclick
       itemDone(e);                                 // Call itemDone()
     });
   }
```

WHICH ELEMENT DID AN EVENT OCCUR ON?

When calling a function, the `event` object's `target` property is the best way to determine which element the event occurred on. But you may see the approach below used; it relies on the `this` keyword.

THE `this` KEYWORD

The `this` keyword refers to the owner of a function. On the right, `this` refers to the element that the event is on.

This works when no parameters are being passed to the function (and therefore it is not called from an anonymous function).

```
function checkUsername() {
  var elMsg = document.getElementById('feedback');
  if (this.value.length < 5) {
    elMsg.innerHTML = 'Not long enough';
  } else {
    elMsg.innerHTML = '';
  }
}
```

```
var el = document.getElementById('username');
el.addEventListener('blur', checkUsername, false);
```

It's like the function had been written here rather than higher up

USING PARAMETERS

If you pass parameters to the function, the `this` keyword no longer works because the owner of the function is no longer the element that the event listener was bound to, it is an anonymous function.
You could pass the element the event was called on as another parameter of the function.

In both cases, the `event` object is the preferred approach.

```
function checkUsername(el, minLength) {
 var elMsg = document.getElementById('feedback');
  if (el.value.length < minLength) {
    elMsg.innerHTML = 'Not long enough';
  } else {
    elMsg.innerHTML = '';
  }
}

var el = document.getElementById('username');
el.addEventListener('blur', function() {
  checkUsername(el, 5);
}, false);
```

DIFFERENT TYPES OF EVENTS

In the rest of the chapter, you learn about the different types of events you can respond to.

Events are defined in:

- The W3C DOM specification
- The HTML5 specification
- In Browser Object Models

Most are a result of the user interacting with the HTML, but there are a few that react to the browser or other DOM events.

We do not show every event, but the examples you see should teach you enough so that you can work with all types of events.

W3C DOM EVENTS

The DOM events specification is managed by the W3C (who also look after other specifications including HTML, CSS, and XML). Most of the events you will meet in this chapter are part of this DOM events specification.

Browsers implement all the events using the same event object that you already met. It also provides feedback such as which element the event occurred on, which key a user pressed, or where the cursor is positioned).

There are, however, some events that are not covered in the DOM event model - in particular those that deal with form elements. (They used to be part of the DOM, but got moved to the HTML5 specification.)

HTML5 EVENTS

The HTML5 specification (that is still being developed) details events that browsers are expected to support that are specifically used with HTML. For example, events that are fired when a form is submitted or form elements are changed (which you will meet on p282):

```
submit
input
change
```

There are also new events introduced with the HTML5 specification that are only supported by more recent browsers. Here are a few (which you will meet on p286):

```
readystatechange
DOMContentLoaded
hashchange
```

BOM EVENTS

Browser manufacturers also implement some events as part of their Browser Object Model (or BOM). Typically these are events not (yet) covered by W3C specifications (although some will be added to W3C specifications in the future). Several of these events dealt with touchscreen devices:

```
touchstart
touchend
touchmove
orientationchange
```

Other events are being added to capture gestures and take advantage of accelerometers. Care is needed using such features, as different browsers often create different implementations of similar functionality.

USER INTERFACE EVENTS

User interface (UI) events occur as a result of interaction with the browser window rather than the HTML page contained within it, e.g., a page having loaded or the browser window being resized.

The event handler / listener for UI events should be attached to the browser window.

In old HTML code, you may see these events used as attributes on the opening <body> tag. (For example, older code used the `onload` attribute to trigger code that would run when the page had loaded.)

EVENT	TRIGGER	BROWSER SUPPORT
`load`	Fires when the web page has finished loading. It can also fire on nodes of other elements that load, such as images, scripts, or objects.	The DOM Level 2 (Nov 2000) states that it fires on the `document` object, but prior to this it fired on the `window` object. Browsers support both for backwards compatibility, and developers often still attach `load` event handlers to the `window` (not `document`) object.
`unload`	Fires when the web page is unloading (usually because a new page has been requested). See also the `beforeunload` event (on p286) which fires before the user leaves a page.	The DOM Level 2 states that it fires on the node for the <body> element, but in older browsers it fired on the `window` object (this is often used for backwards compatibility).
`error`	Fires when the browser encounters a JavaScript error or an asset doesn't exist.	Support for this event is inconsistent across browsers and so it is not reliable for error handling, a topic you learn more about in Chapter 10.
`resize`	Fires when the browser window has been resized.	Browsers repeatedly fire the `resize` event as the window is being resized, so avoid using this event to trigger complicated code because this might make the page appear less responsive.
`scroll`	Fires when the user has scrolled up or down the page. It can relate to the entire page or a specific element on the page (such as a `<textarea>` that has scrollbars).	Browsers repeatedly fire the event as the window is scrolled, so avoid running complicated code as the user scrolls.

LOAD

The `load` event is commonly used to trigger scripts that access the contents of the page. In this example, a function called `setup()` gives focus to the text input when the page has loaded.

The event is automatically raised by the `window` object when a page has finished loading the HTML *and* all of its resources: images, CSS, scripts (even third party content e.g., banner ads).

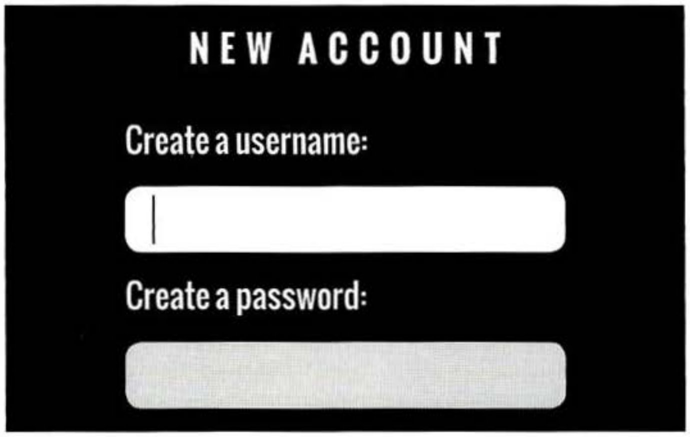
The `setup()` function would not work before the page has loaded because it relies on finding the element whose `id` attribute has a value of `username`, in order to give it focus.

JAVASCRIPT

c06/js/load.js

```
function setup() {                                     // Declare function
  var textInput;                                       // Create variable
  textInput = document.getElementById('username');    // Get username input
  textInput.focus();                                   // Give username focus
}

window.addEventListener('load', setup, false); // When page loaded call setup()
```

RESULT



Note that the event listener is attached to the `window` object (not the `document` object - as this can cause cross-browser compatibility issues).

If the `<script>` element is at the end of the HTML page, then the DOM would have loaded the form elements before the script runs, and there would be no need to wait for the `load` event. (See also: the `DOMContentLoaded` event on p286 and jQuery's `document.ready()` method on p312.)

Because the `load` event only fires when everything else on the page has loaded (images, scripts, even ads), the user already have started to use the page *before* the script has started to run.

Users particularly notice when a script changes the appearance of the page, changes focus, or selects form elements after they have started to use it. (It can make a site look slower to load.)

Imagine this form had more inputs; the user may be filling in the second or third box when the script fires - moving focus back to the first box too late and interrupting the user.

FOCUS & BLUR EVENTS

The HTML elements you can interact with, such as links and form elements, can gain focus. These events fire when they gain or lose focus.

If you can interact with an HTML element, then it can gain (and lose) focus. You can also tab between the elements that can gain focus (a technique often used by those with visual impairments).

In older scripts, the `focus` and `blur` events were often used to change the appearance of an element as it gained focus, but now the CSS `:focus` pseudo-class is a better solution (unless you need to affect an element *other* than the one that gained focus).

The `focus` and `blur` events are most commonly used on forms. They can be particularly helpful when:

- You want to show tips or feedback to users as they interact with an individual element within a form (the tips are usually shown in *other* elements and *not* the one they are interacting with)
- You need to trigger form validation as a user moves from one control to the next (rather than waiting for them to submit the entire form first)

EVENT	TRIGGER	FLOW
`focus`	When an element gains focus, the `focus` event fires for that DOM node.	Capture
`blur`	When an element loses focus, the `blur` event fires for that DOM node.	Capture
`focusin`	Same as `focus` (see above but not supported in Firefox at time of writing)	Bubble & capture
`focusout`	Same as `blur` (see above but not supported in Firefox at time of writing)	Bubble & capture

FOCUS & BLUR

In this example, as the text input gains and loses focus, feedback is shown to the user in the `<div>` element under the text input. The feedback is given using two functions.

`tipUsername()` is triggered when the text input gains focus. It changes the `class` attribute of the element containing the message, and updates the contents of the element.

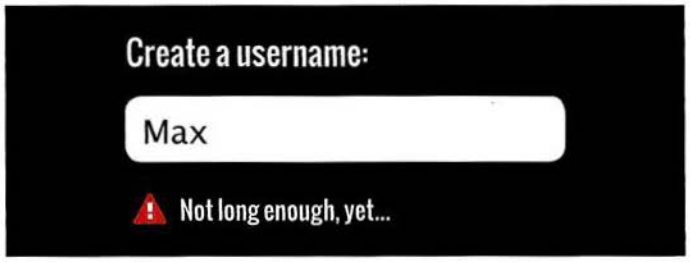
`checkUsername()` is triggered when the text input loses focus. It adds a message and changes the `class` if the username is less than 5 characters; otherwise, it clears the message.

JAVASCRIPT

c06/js/focus-blur.js

```
function checkUsername() {                          // Declare function
  var username = el.value;                          // Store username in variable
  if (username.length < 5) {                        // If username < 5 characters
    elMsg.className = 'warning';                    // Change class on message
    elMsg.textContent = 'Not long enough, yet...';// Update message
  } else {                                          // Otherwise
    elMsg.textContent = '';                         // Clear the message
  }
}
function tipUsername() {                            // Declare function
    elMsg.className = 'tip';                        // Change class for message
    elMsg.innerHTML = 'Username must be at least 5 characters'; // Add message
}

var el = document.getElementById('username');       // Username input
var elMsg = document.getElementById('feedback');    // Element to hold message

// When the username input gains / loses focus call functions above:
el.addEventListener('focus', tipUsername, false); // focus call tipUsername()
el.addEventListener('blur', checkUsername, false);// blur call checkUsername()
```

RESULT



Create a username:

Max

Not long enough, yet...

MOUSE EVENTS

The mouse events are fired when the mouse is moved and also when its buttons are clicked.

All of the elements on a page support the mouse events, and all of these bubble. Note that actions are different on touchscreen devices.

Preventing a default behavior can have unexpected results. E.g., a `click` event only fires when both the `mousedown` and `mouseup` event have fired.

EVENT	TRIGGER	TOUCH
`click`	Fires when the user clicks on the primary mouse button (usually the left button if there is more than one). The `click` event will fire for the element that the mouse is currently over. It will also fire if the user presses the Enter key on the keyboard when an element has focus.	A tap on the touchscreen will be treated like a single left-click.
`dblclick`	Fires when the user clicks the primary mouse button twice in quick succession.	A double-tap will be treated as a double left click.
`mousedown`	Fires when the user clicks down on any mouse button. (Cannot be triggered by keyboard.)	You can use the `touchstart` event.
`mouseup`	Fires when the user releases a mouse button. (Cannot be triggered by keyboard.)	You can use the `touchend` event.
`mouseover`	Fires when the cursor was outside an element and is then moved inside it. (Cannot be triggered by keyboard.)	Fires when the cursor is moved over an element.
`mouseout`	Fires when the cursor is over an element, and then moves onto another element - outside of the current element or a child of it (Cannot be triggered by keyboard.)	Fires when the cursor is moved off an element.
`mousemove`	Fires when the cursor is moved around an element. This event is repeatedly fired. (Cannot be triggered by keyboard.)	Fires when the cursor is moved.

WHEN TO USE CSS

The `mouseover` and `mouseout` events were often used to change the appearance of boxes or to switch images as the user rolls over them. To change the appearance of the element, a preferable technique would be to use the CSS `:hover` pseudo-class.

WHY SEPARATE MOUSEDOWN & UP?

The `mousedown` and `mouseup` events separate out the press and release of a mouse button. They are commonly used for adding drag and drop functionality, or to add controls in game development.

CLICK

The aim of this example is to use the `click` event to remove the big note that has been added to the middle of the page. But first, the script has to create that note.

Because the note is over the top of the page, we only want to show it to users who have JavaScript enabled (otherwise they could not hide it).

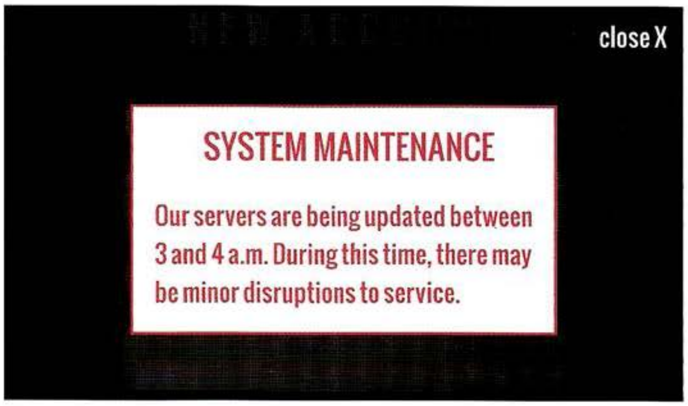
When the `click` event fires on the close link the `dismissNote()` function is called. This function will remove the note that was added by the same script.

JAVASCRIPT

c06/js/click.js

```
// Create the HTML for the message
var msg = '<div class=\"header\"><a id=\"close\" href="#">close X</a></div>';
msg += '<div><h2>System Maintenance</h2>';
msg += 'Our servers are being updated between 3 and 4 a.m. ';
msg += 'During this time, there may be minor disruptions to service.</div>';

var elNote = document.createElement('div');        // Create a new element
elNote.setAttribute('id', 'note');                 // Add an id of note
elNote.innerHTML = msg;                            // Add the message
document.body.appendChild(elNote);                 // Add it to the page

function dismissNote() {                           // Declare function
  document.body.removeChild(elNote);               // Remove the note
}

var elClose = document.getElementById('close');    // Get the close button
elClose.addEventListener('click', dismissNote, false);// Click close-clear note
```

RESULT



SYSTEM MAINTENANCE

Our servers are being updated between 3 and 4 a.m. During this time, there may be minor disruptions to service.

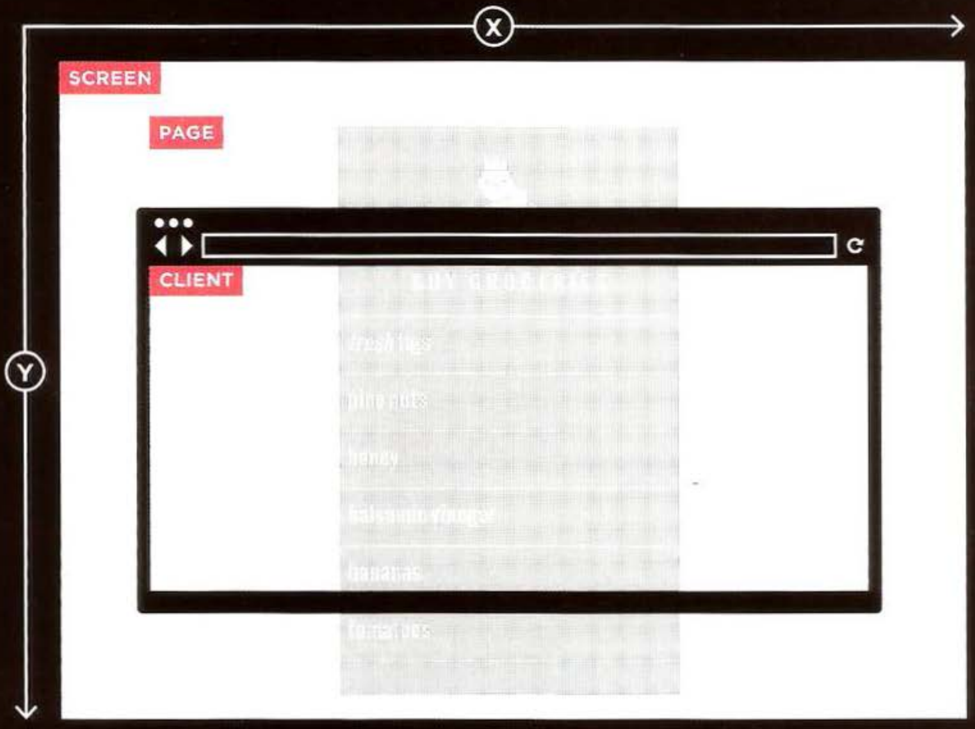
ACCESSIBILITY

The `click` event can be applied to any element, but it is better to only use it on items that are usually clicked or it will not be accessible to people who rely upon keyboard navigation.

You may also be tempted to use the `click` event to run a script when a user clicks on a form element, but it is better to use the `focus` event because that fires when the user accesses that control using the tab key.

WHERE EVENTS OCCUR

The **event** object can tell you where the cursor was positioned when an event was triggered.



SCREEN

The **screenX** and **screenY** properties indicate the position of the cursor within the entire screen on your monitor, measuring from the top left corner of the screen (rather than the browser).

PAGE

The **pageX** and **pageY** properties indicate the position of the cursor within the entire page. The top of the page may be outside of the viewport so even if the cursor is in the same position, page and client coordinates can be different.

CLIENT

The **clientX** and **clientY** properties indicate the position of the cursor within the browser's viewport. If the user has scrolled down and the top of the page is no longer in view, it will not affect the client coordinates.

DETERMINING POSITION

In this example, as you move your mouse around the screen, the text inputs across the top of the page are updated with the current mouse position.

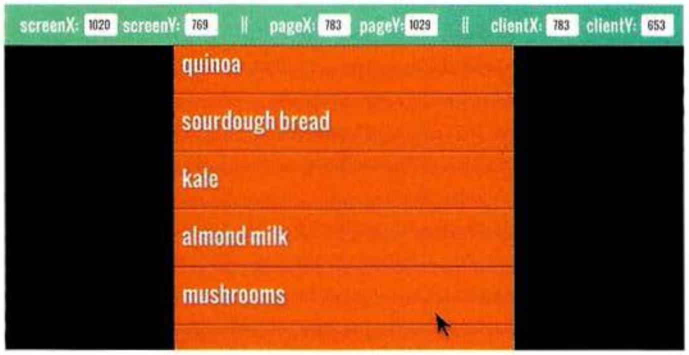
This demonstrates the three different positions you can retrieve when the mouse is moved or when one of the buttons is clicked.

Note how `showPosition()` is passed `event` as a parameter, which refers to the `event` object. The positions are all properties of this `event` object.

JAVASCRIPT c06/js/position.js

```
var sx = document.getElementById('sx');        // Element to hold screenX
var sy = document.getElementById('sy');        // Element to hold screenY
var px = document.getElementById('px');        // Element to hold pageX
var py = document.getElementById('py');        // Element to hold pageY
var cx = document.getElementById('cx');        // Element to hold clientX
var cy = document.getElementById('cy');        // Element to hold clientY

function showPosition(event) {                 // Declare function
  sx.value = event.screenX;                    // Update element with screenX
  sy.value = event.screenY;                    // Update element with screenY
  px.value = event.pageX;                      // Update element with pageX
  py.value = event.pageY;                      // Update element with pageY
  cx.value = event.clientX;                    // Update element with clientX
  cy.value = event.clientY;                    // Update element with clientY
}

var el = document.getElementById('body');                // Get body element
el.addEventListener('mousemove', showPosition, false); // Move updates position
```

RESULT



screenX: 1020 screenY: 769 || pageX: 783 pageY: 1029 || clientX: 783 clientY: 653

quinoa

sourdough bread

kale

almond milk

mushrooms

KEYBOARD EVENTS

The keyboard events are fired when a user interacts with the keyboard (they fire on any kind of device with a keyboard).

EVENT	TRIGGER
`input`	Fires when the value of an `<input>` or `<textarea>` element changes. First supported in IE9 (although it does not fire when deleting text in IE9). For older browsers, you can use **keydown** as a fallback.
`keydown`	Fires when the user presses any key on the keyboard. If the user holds down a key, the event continues to fire repeatedly. This is important because it mimics what would happen in a text input if the user holds down a key (the same character would be added repeatedly while the key is held down).
`keypress`	Fires when the user presses a key that would result in a character being shown on the screen. For example, this event would not fire when the user presses the arrow keys, whereas the **keydown** event would. If the user holds down a key, the event continues to fire repeatedly.
`keyup`	Fires when the user releases a key on the keyboard. The **keydown** and **keypress** events fire before a character shows on screen, whereas **keyup** fires after it appears.

The three events that begin **key...** fire in this order:

1. keydown - user presses key down
2. keypress - user has pressed or is holding a key that adds a character into the page
3. keyup - user releases key

WHICH KEY WAS PRESSED?

When you use the **keydown** or **keypress** events, the **event** object has a property called **keyCode**, which can be used to tell which key was pressed. However, it does not return the *letter* for that key (as you might expect); it returns an **ASCII code** that represents the lowercase character for that key. You can see a table of the characters and their ASCII codes in an online extra on the website accompanying this book.

If you want to get the letter or number as it would be displayed on the keyboard (rather than an ASCII equivalent), the `String` object has a built-in method called `fromCharCode()` which will do the conversion for you: `String.fromCharCode(event.keycode);`

WHICH KEY WAS PRESSED

In this example, the `<textarea>` element should only have 180 characters. When the user enters text, the script will show them how many characters they have left available to use.

The event listener checks for the keypress event on the `<textarea>` element. Each time it fires, the `charCount()` function updates the character count and shows the last character used.

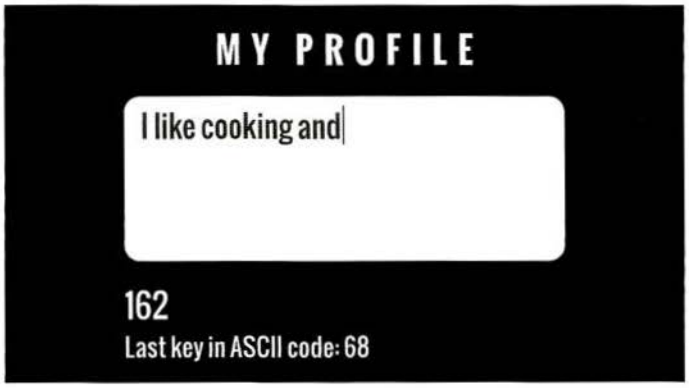
The `input` event would work well to update the count when the user pastes in text or uses keys like backspace, but it does not tell you which key was the last to be pressed.

JAVASCRIPT

c06/js/keypress.js

```
var el;                                                     // Declare variables

function charCount(e) {                                     // Declare function
  var textEntered, charDisplay, counter, lastKey;           // Declare variables
  textEntered = document.getElementById('message').value;   // User's text
  charDisplay = document.getElementById('charactersLeft'); // Counter element
  counter = (180 - (textEntered.length));                   // Num of chars left
  charDisplay.textContent = counter;                        // Show chars left

  lastkey = document.getElementById('lastkey');             // Get last key used
  lastkey.textContent = 'Last key in ASCII code: ' + e.keyCode; // Create msg
}
el = document.getElementById('message');                    // Get msg element
el.addEventListener('keypress', charCount, false);          // keypress event
```

RESULT



MY PROFILE

I like cooking and

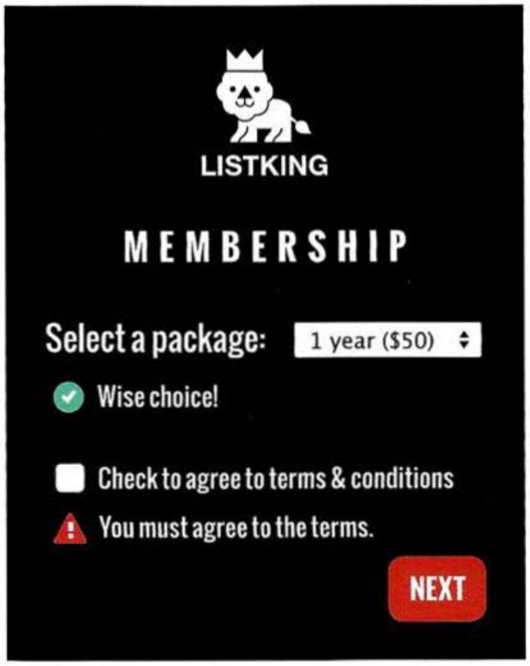
162

Last key in ASCII code: 68

FORM EVENTS

There are two events that are commonly used with forms. In particular you are likely to see `submit` used in form validation.

EVENT	TRIGGER
`submit`	When a form is submitted, the `submit` event fires on the node representing the `<form>` element. It is most commonly used when checking the values a user has entered into a form before sending it to the server.
`change`	Fires when the status of several form elements change. For example, when: • a selection is made from a drop-down select box • a radio button is selected • a checkbox is selected or deselected It is often better to use the change event rather than the `click` event because clicking is not the only way users interact with form elements (for example, they might use the tab, arrow, or Enter keys).
`input`	The `input` event, which you saw on the previous page is commonly used with `<input>` and `<textarea>` elements.


LISTKING
MEMBERSHIP
Select a package:
1 year ($50)
Wise choice!
Check to agree to terms & conditions
You must agree to the terms.
NEXT

FOCUS AND BLUR

The `focus` and `blur` events (which you met on p274) are often used with forms, but they can also be used in conjunction with other elements, such as links (so they are not specifically related to forms).

VALIDATION

Checking form values is known as validation. If users miss required information or enter incorrect information, checking it using JavaScript is faster than sending the data to the server for it to be checked. Validation is covered in Chapter 13.

USING FORM EVENTS

When a user interacts with the drop-down select box, the change event will trigger the packageHint() function. This shows messages below the select box that reflect the choice.

When the form is submitted, the checkTerms() function is called. This tests to see if the user has checked the box that indicates they agree to the terms and conditions.

If not, the script will prevent the default behavior of the form element (and stop it from submitting the form data to the server) and it will show an error message to the user.

JAVASCRIPT

c06/js/form.js

```
var elForm, elSelectPackage, elPackageHint, elTerms;     // Declare variables
elForm          = document.getElementById('formSignup'); // Store elements
elSelectPackage = document.getElementById('package');
elPackageHint   = document.getElementById('packageHint');
elTerms         = document.getElementById('terms');
elTermsHint     = document.getElementById('termsHint');

function packageHint() {                                 // Declare function
  var package = this.options[this.selectedIndex].value;  // Get selected option
  if (package == 'monthly') {                            // If monthly package
    elPackageHint.innerHTML = 'Save $10 if you pay for 1 year!';//Show this msg
  } else {                                               // Otherwise
    elPackageHint.innerHTML = 'Wise choice!';            // Show this message
  }
}

function checkTerms(event) {                             // Declare function
  if (!elTerms.checked) {                                // If checkbox ticked
    elTermsHint.innerHTML = 'You must agree to the terms.'; // Show message
    event.preventDefault();                              // Don't submit form
  }
}

//Create event listeners: submit calls checkTerms(), change calls packageHint()
elForm.addEventListener('submit', checkTerms, false);
elSelectPackage.addEventListener('change', packageHint, false);
```

MUTATION EVENTS & OBSERVERS

Whenever elements are added to or removed from the DOM, its structure changes. This change triggers a mutation event.

When your script adds or removes content from a page it is updating the DOM tree. There are many reasons why you might want to respond to the DOM tree being updated, for example, you might want to tell the user that the page had changed.

Below are some events that are triggered when the DOM changes. These mutation events were introduced in Firefox 3, IE9, Safari 3, and all versions of Chrome. But they are already scheduled to be replaced by an alternative called mutation observers.

EVENT	TRIGGER
`DOMNodeInserted`	Fires when a node is inserted into the DOM tree. e.g. using `appendChild()`, `replaceChild()`, or `insertBefore()`.
`DOMNodeRemoved`	Fires when a node is removed from the DOM tree. e.g. using `removeChild()` or `replaceChild()`.
`DOMSubtreeModified`	Fires when the DOM structure changes. It fires after the two events listed above occur.
`DOMNodeInsertedIntoDocument`	Fires when a node is inserted into the DOM tree as a descendant of another node that is already in the document.
`DOMNodeRemovedFromDocument`	Fires when a node is removed from the DOM tree as a descendant of another node that is already in the document.

PROBLEMS WITH MUTATION EVENTS

If your script makes a lot of changes to a page, you end up with a lot of mutation events firing. This can make a page feel slow or unresponsive. They can also trigger other event listeners as they propagate through the DOM, which modify other parts of the DOM, triggering more mutation events. Therefore they are being replaced by mutation observers.

Browser support: Chrome, Firefox 3, IE 9, Opera 9, Safari 3

NEW MUTATION OBSERVERS

Mutation observers are designed to wait until a script has finished its task before reacting, then report the changes as a batch (rather than one at a time). You can also specify the type of changes to the DOM that you want them to react to. But at the time of writing, the browser support was not widespread enough to use them on public websites.

Browser support: IE 11, Firefox 14, Chrome 27 (or 18 with webkit prefix), Safari 6.1, Opera 15
On mobile: Android 4.4, Safari on iOS 7.

USING MUTATION EVENTS

In this example, two event listeners each trigger their own function. The first is on the last but one line, and it listens for when the user clicks the link to add a new list item. It then uses DOM manipulation events to add a new element (changing the DOM structure and triggering mutation events).

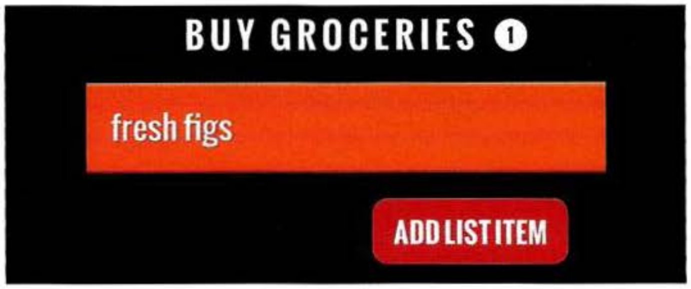
The second event listener waits for the DOM tree within the `<ul>` element to change. When the `DOMNodeInserted` event fires, it calls a function called `updateCount()`. This function counts how many items there are in the list, and then updates the list count at the top of the page accordingly.

JAVASCRIPT

c06/js/mutation.js

```
var elList, addLink, newEl, newText, counter, listItems; // Declare variables

elList  = document.getElementById('list');              // Get list
addLink = document.querySelector('a');                  // Get add item button
counter = document.getElementById('counter');           // Get item counter

function addItem(e) {                                   // Declare function
  e.preventDefault();                                   // Prevent link action
  newEl = document.createElement('li');                 // New <li> element
  newText = document.createTextNode('New list item');   // New text node
  newEl.appendChild(newText);                           // Add text to <li>
  elList.appendChild(newEl);                            // Add <li> to list
}

function updateCount() {                                // Declare function
  listitems = list.getElementsByTagName('li').length;   // Get total of <li>s
  counter.innerHTML = listitems;                        // Update counter
}

addLink.addEventListener('click', addItem, false);      // Click on button
elList.addEventListener('DOMNodeInserted', updateCount, false); // DOM updated
```

RESULT



HTML5 EVENTS

Here are three page-level events that have been included in versions of the HTML5 spec that have become popular very quickly.

EVENT	TRIGGER	BROWSER SUPPORT
`DOMContentLoaded`	Event fires when the DOM tree is formed (images, CSS, and JavaScript might still be loading). Scripts start to run earlier than using the `load` event which waits for other resources such as images and advertisements to load. This makes the page seem faster to load. However, because it does not wait for scripts to load, the DOM tree will not contain any HTML that would have been generated by those scripts. It can be attached to the `window` or `document` objects.	Chrome 0.2, Firefox 1, IE9, Safari 3.1, Opera 9
`hashchange`	Event fires when the URL hash changes (without the entire window refreshing). Hashes are used on links to specific parts (sometimes known as anchors) within a page and also on pages that use AJAX to load content. The `hashchange` event handler works on the `window` object, and after firing, the `event` object will have `oldURL` and `newURL` properties that hold the url before and after the `hashchange`.	IE8, Firefox 20, Safari 5.1, Chrome 26, and Opera 12.1
`beforeunload`	Event fires on the `window` object before the page is unloaded. It should only be used to help the user (not to encourage them to stay on a website if they are trying to leave). For example, it can be helpful to let a user know that changes on a form they completed have not been saved. You can add a message to the dialog box that is shown by the browser, but you do not have control over the text shown before it or on the buttons the user can press (which can vary slightly between browsers and operating systems).	Chrome 1, Firefox 1, IE4, Safari 3, Opera 12

There are also several other events that are being introduced to support more recent devices (such as phones and tablets). They respond to events such as gestures and movements that are based upon an accelerometer (which detects the angle at which a device is being held).

USING HTML5 EVENTS

In this example, as soon as the DOM tree has been formed, focus is given to the text input with an `id` of `username`.

The `DOMContentLoaded` event fires before the `load` event (because the latter waits for all of the page's resources to load).

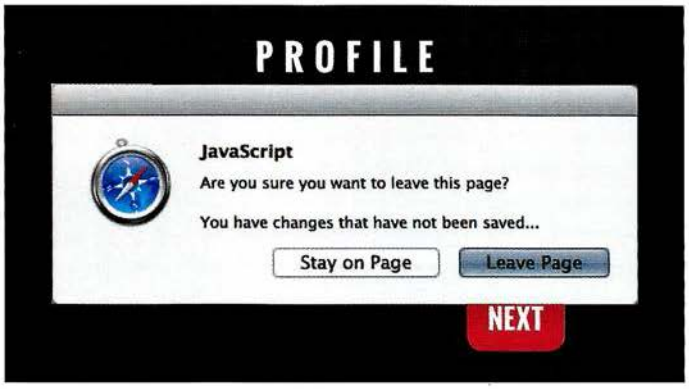
If users try to leave the page before they press the submit button, the `beforeunload` event checks that they want to leave.

JAVASCRIPT

06/js/html5-events.js

```
function setup() {
  var textInput;
  textInput = document.getElementById('message');
  textInput.focus();
}

window.addEventListener('DOMContentLoaded', setup, false);

window.addEventListener('beforeunload', function(event){
  return 'You have changes that have not been saved...';
}, false);
```

RESULT

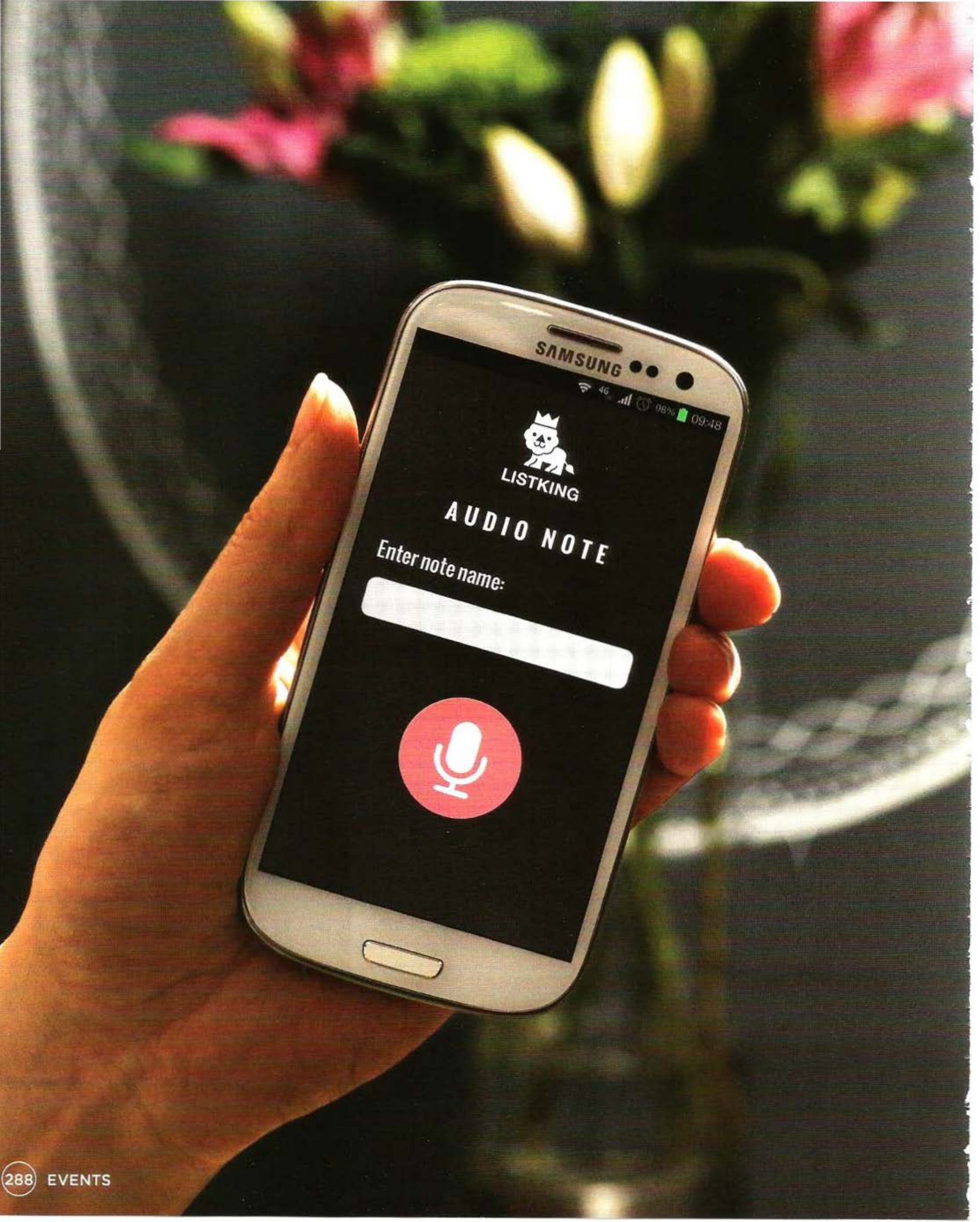


Are you sure you want to leave this page?

You have changes that have not been saved...

On the left, you can see the dialog box that is shown when you try to navigate away from the page.

The text before your message and on the buttons may change from browser to browser (you have no control over this).


SAMSUNG
09:48
LISTKING
AUDIO NOTE
Enter note name:

EXAMPLE
EVENTS

This example shows an interface for a user to record voice notes. The user can enter a name which is displayed in the heading, and they can press record (which changes the image that is shown).

When the user starts typing a name into the text box, the keyup event will trigger a function called `writeLabel()` which copies the text from the form input and writes it into the main heading under the logo for List King, replacing the words 'AUDIO NOTE'.

The record / pause button is a bit more interesting. The button has an attribute called `data-state`. When the page loads, its value is `record`. When the user presses the button, the value of this attribute changes to `pause` (this triggers a new CSS rule to indicate that it is now recording).

If you have not used HTML5's `data-` attributes, they allow you to store custom data on any HTML element. (The name of the attribute can be anything starting with `data-` as long as the name is lowercase.)

This demonstrates a new technique based upon event delegation. The event listener is placed upon the containing element whose `id` is `buttons`. The `event` object is used to determine the value of the `id` attribute on the element that was used. The value from that `id` attribute is then used in a `switch` statement to decide which function to call (depending on whether the button is in `record` state or `pause` state).

This is a good way to handle many buttons because it reduces the number of event listeners in your code.

The event listeners are written at the bottom of the page, and they have fallbacks for users who are running IE8 or less (which has a different event model).

EXAMPLE
EVENTS

The script starts by defining the variables that it will need to use, and then collecting the element nodes that are needed.

The player functions (shown on the right-hand page) would appear next, and at the bottom of this page you can see the event listeners.

The event listeners live inside a conditional statement so that the `attachEvent()` method can be used for visitors who have IE8 or less.

c06/js/example.js

JAVASCRIPT

```
var username, noteName, textEntered, target;     // Declare variables

noteName = document.getElementById('noteName');  // Element that holds note

function writeLabel(e) {                         // Declare function
  if (!e) {                                      // If event object not present
    e = window.event;                            // Use IE5-8 fallback
  }
  target = event.target || event.srcElement;     // Get target of event
  textEntered = e.target.value;                  // Value of that element
  noteName.textContent = textEntered;            // Update note text
}

// This is where the record / pause controls and functions go...
// See right hand page

if (document.addEventListener) {                 // If event listener supported
  document.addEventListener('click', function(e){// For any click document
    recorderControls(e);                         // Call recorderControls()
  }, false);                                     // Capture during bubble phase
  // If input event fires on username input call writeLabel()
  username.addEventListener('input', writeLabel, false);
} else {                                         // Otherwise
  document.attachEvent('onclick', function(e){   // IE fallback: any click
    recorderControls(e);                         // Calls recorderControls()
  });
  // If keyup event fires on username input call writeLabel()
  username.attachEvent('onkeyup', writeLabel, false);
}
```

EXAMPLE
EVENTS

The recorderControls() function is automatically passed the event object. Not only does this offer code to support older versions of IE, but also stops the link from performing its default behavior (of taking the user to a new page).

The switch statement is used to indicate which function to run depending on whether the user is trying to record or stop the audio note. This technique of delegation is a good way to cope with multiple buttons in the UI.

JAVASCRIPT c06/js/example.js

```
function recorderControls(e) {                     // Declare recorderControls()
  if (!e) {                                        // If event object not present
    e = window.event;                              // Use IE5-8 fallback
  }
  target = event.target || event.srcElement;// Get the target element
  if (event.preventDefault) {                      // If preventDefault() supported
    e.preventDefault();                            // Stop default action
  } else {                                         // Otherwise
    event.returnValue = false;                     // IE fallback: stop default action
}

switch(target.getAttribute('data-state')) { // Get the data-state attribute
  case 'record':                                   // If its value is record
    record(target);                                // Call the record() function
    break;                                         // Exit function to where called
  case 'stop':                                     // If its value is stop
    stop(target);                                  // Call the stop() function
    break;                                         // Exit function to where called
    // More buttons could go here...
  }
};

function record(target) {                          // Declare function
    target.setAttribute('data-state', 'stop'); // Set data-state attr to stop
    target.textContent = 'stop';                   // Set text to 'stop'
}

function stop(target) {
    target.setAttribute('data-state', 'record');//Set data-state attr to record
    target.textContent = 'record';                 // Set text to 'record'
}
```

SUMMARY
EVENTS

- Events are the browser's way of indicating when something has happened (such as when a page has finished loading or a button has been clicked).
- Binding is the process of stating which event you are waiting to happen, and which element you are waiting for that event to happen upon.
- When an event occurs on an element, it can trigger a JavaScript function. When this function then changes the web page in some way, it feels interactive because it has responded to the user.
- You can use event delegation to monitor for events that happen on all of the children of an element.
- The most commonly used events are W3C DOM events, although there are others in the HTML5 specification as well as browser-specific events.