ANIMATING CSS PROPERTIES

The `.animate()` method allows you to create some of your own effects and animations by changing CSS properties.

You can animate any CSS property whose value can be represented as a number, e.g., `height`, `width`, and `font-size`. But not those whose value would be a string, such as `font-family` or `text-transform`.

The CSS properties are written using camelCase notation, so the first word is all lowercase and each subsequent word starts with an uppercase character, e.g.: `border-top-left-radius` would become `borderTopLeftRadius`.

The CSS properties are specified using object literal notation (as you can see on the right-hand page). The method can also take three optional parameters, shown below.

```
.animate({
  // Styles you want to change
  }[, speed][, easing][, complete]);
```

1. `speed` indicates the duration of the animation in milliseconds. (It can also take the keywords `slow` and `fast`.)

2. `easing` can have two values: `linear` (the speed of animation is uniform); or `swing` (speeds up in the middle of the transition, and is slower at start and end).

3. `complete` is used to call a function that should run when the animation has finished. This is known as a **callback function**.

EXAMPLES OF JQUERY EQUIVALENTS OF CSS PROPERTY NAMES

`bottom` `left` `right` `top` `backgroundPositionX` `backgroundPositionY` `height` `width` `maxHeight` `minHeight` `maxWidth` `minWidth` `margin` `marginBottom` `marginLeft` `marginRight` `marginTop` `outlineWidth` `padding` `paddingBottom` `paddingLeft` `paddingRight` `paddingTop` `fontSize` `letterSpacing` `wordSpacing` `lineHeight` `textIndent` `borderRadius` `borderWidth` `borderBottomWidth` `borderLeftWidth` `borderRightWidth` `borderTopWidth` `borderSpacing`

USING ANIMATION

In this example, the `.animate()` method is used to gradually change the values of two CSS properties. Both of them have numerical values: `opacity` and `padding-left`.

When the user clicks on a list item, it fades out and the text content slides to the right. (This takes 500ms.) Once that is complete, a callback function removes the element.

You can increase or decrease numeric values by a specific amount. Here, `+=80` is used to increase the `padding` property by 80 pixels. (To decrease it by 80 pixels, you would use `-=80`.)

JAVASCRIPT c07/js/animate.js

```
$(function() {
  $('li').on('click', function() {
    $(this).animate({
      opacity: 0.0,
      paddingLeft: '+=80'
    }, 500, function() {
      $(this).remove();
    });
  });
});
```

1. All list items are selected and, when a user clicks on one of them, an anonymous function runs. Inside it, `$(this)` creates a new jQuery object holding the element the user clicked on. The `.animate()` method is then called on that jQuery object.

2. Inside the `.animate()` method, the `opacity` and `paddingLeft` are changed. The value of the `paddingLeft` property is increased by 80 pixels, which makes it look like the text is sliding to the right as it fades out.

3. The `.animate()` method has two more parameters. The first is the speed of the animation in milliseconds (in this case, 500ms). The second is another anonymous function indicating what should happen when the animation finishes.

4. When the animation has finished, the callback function removes that list item from the page using the `.remove()` method.

RESULT

If you want to animate between two colors, rather than using the `.animate()` method, there is a helpful jQuery color plugin here:

`https://github.com/jquery/jquery-color`

TRAVERSING THE DOM

When you have made a jQuery selection, you can use these methods to access other element nodes relative to the initial selection.

Each method finds elements that have a different relationship to those that are in the current selection (e.g., parents or children of the current selection).

The `.find()` and `.closest()` methods both *require* a CSS-style selector as an argument.

For the other methods, the CSS-style selector is optional. But if a selector is provided, both the method and selector must match in order for the element to be added to the new selection.

For example, if you start with a selection that contains one list item, you could create a new selection containing the other items from the list using the `.siblings()` method.

If you added a selector into the method such as this: `.siblings('.important')` then it would find only siblings with a `class` attribute whose value included `important`.

SELECTOR REQUIRED

METHOD	DESCRIPTION
`.find()`	All elements within current selection that match selector
`.closest()`	Nearest ancestor (not just parent) that matches selector

SELECTOR OPTIONAL

METHOD	DESCRIPTION
`.parent()`	Direct parent of current selection
`.parents()`	All parents of current selection
`.children()`	All children of current selection
`.siblings()`	All siblings of current selection
`.next()`	Next sibling of current element
`.nextAll()`	All subsequent siblings of current element
`.prev()`	Previous sibling of current element
`.prevAll()`	All previous siblings of current element

If the original selection contains multiple elements, these methods will work on all of the elements in the selection (which can result in quite an odd selection of elements). You may need to narrow down your initial selection before traversing the DOM.

Behind the scenes, jQuery will handle the cross-browser inconsistencies involved in traversing the DOM (such as whitespace nodes being added by some browsers).

When the page loads, the list is hidden, and a link is added to the heading that indicates the users can display the list if they wish.

The link is added inside the heading and, if the user clicks anywhere on the <h2> element, the <ul> element is faded in.

Any child <li> elements that have a class attribute whose value is hot are also given an extra value of complete.

JAVASCRIPT

c07/js/traversing.js

```
$(function() {
  var $h2 = $('h2');
  $('ul').hide();
  $h2.append('<a>show</a>');

① $h2.on('click', function() {
② $h2.next()
③ .fadeIn(500)
④ .children('.hot')
⑤ .addClass('complete');
⑥ $h2.find('a').fadeOut();
  });

});
```

1. A click event anywhere in the <h2> element will trigger an anonymous function.
2. The .next() method is used to select the next sibling after the <h2> element, which is the <ul> element.
3. The <ul> is faded into view.
4. The .children() method then selects any child elements of the <ul> element, and the selector indicates that it should pick only those whose class attribute has a value of hot.
5. The .addClass() method is then used on those <li> elements to add a class name of complete. This shows how you can chain methods and traverse from one node to another.
6. In the last step, the .find() method can be used to select the <a> element that is a child of the <h2> element and fade it out because the list is now being shown to the users.

RESULT

ADD & FILTER ELEMENTS IN A SELECTION

Once you have a jQuery selection, you can add more elements to it, or you can filter the selection to work with a subset of the elements.

The `.add()` method allows you to add a new selection to an existing one.

The second table on the right shows you how to find a subset of your original selection.

The methods take another selector as a parameter and return a filtered matched set.

The items in this table that begin with a colon can be used wherever you would use a CSS-style selector.

The `:not()` and `:has()` selectors take another CSS-style selector as a parameter. There is also a selector called `:contains()` that lets you find elements that contain specific text.

The `.is()` method lets you use another selector to check whether the current selection matches a condition. If it does, it will return `true`. This is helpful in conditional statements.

ADDING ELEMENTS TO A SELECTION

METHOD	DESCRIPTION
`.add()`	Selects all elements that contain the text specified (parameter is case sensitive)

FILTERING WITH A SECOND SELECTOR

METHOD / SELECTOR	DESCRIPTION
`.filter()`	Finds elements in matched that in turn match a second selector
`.find()`	Finds descendants of elements in matched set that match the selector
`.not()` / `:not()`	Finds elements that do not match the selector
`.has()` / `:has()`	Finds elements from the matched set that have a descendant that matches the selector
`:contains()`	Selects all elements that contain the text specified (parameter is case sensitive)

The following two selectors are equivalent:

```
$('li').not('.hot').addClass('cool');
$('li:not(.hot)').addClass('cool');
```

In browsers that support `querySelector()` / `querySelectorAll()`, `:not()` is faster than `.not()` and `:has()` is faster than `.has()`

TESTING CONTENT

METHOD	DESCRIPTION
`.is()`	Checks whether current selection matches a condition (returns Boolean)

FILTERS IN USE

This example selects all list items and then uses different filters to select a subset of the items from the list to work with.

The example uses both the filtering methods as well as the CSS-style pseudo-selector `:not()`.

Once the filters have selected a subset of the list items, other jQuery methods are used to update them.

JAVASCRIPT c07/js/filters.js

```
var $listItems = $('li');
① $listItems.filter('.hot:last').removeClass('hot');
② $('li:not(.hot)').addClass('cool');
③ $listItems.has('em').addClass('complete');

$listItems.each(function() {
  var $this = $(this);
④ if ($this.is('.hot')) {
    $this.prepend('Priority item: ');
  }
});

⑤ $('li:contains("honey")').append(' (local)');
```

1. The `.filter()` method finds the last list item with a `class` attribute whose value is `hot`. It then removes that value from the `class` attribute.

2. The `:not()` selector is used within the jQuery selector to find `<li>` elements without a value of `hot` in their `class` attribute and adds a value of `cool`.

3. The `.has()` method finds the `<li>` element that has an `<em>` element within it and adds the value `complete` to the `class` attribute.

4. The `.each()` method loops through the list items. The current element is cached in a jQuery object. The `.is()` method looks to see if the `<li>` element has a `class` attribute whose value is `hot`. If it does, `'Priority item: '` is added to the start of the item.

5. The `:contains` selector checks for `<li>` elements that contain the text `"honey"` and appends the text `" (local)"` to the end of those items.

RESULT

FINDING ITEMS BY ORDER

Each item returned by a jQuery selector is given an index number, which can be used to filter the selection.

The `jQuery` object is sometimes referred to as being an **array-like** object because it assigns a number to each of the elements that is returned by a selector. That number is an index number, which means it starts at 0.

You can filter the selected elements based on this number using methods or these additional CSS-style selectors that jQuery has added.

Methods are applied to the jQuery selection, whereas selectors are used as part of the CSS-style selector.

On the right, you can see a selector which picks all of the `<li>` elements from the list example used throughout this chapter. The table shows each list item and its corresponding index number. The example on the next page will use these numbers to select list items and update their `class` attributes.

FINDING ELEMENTS BY INDEX NUMBER

METHOD / SELECTOR	DESCRIPTION
`.eq()`	The element that matches the index number
`:lt()`	Elements with an index less than the number specified
`:gt()`	Elements with an index greater than the number specified

`$('li')`

INDEX	HTML
0	`<li id="one" class="hot"><em>fresh</em> figs</li>`
1	`<li id="two" class="hot">pine nuts</li>`
2	`<li id="three" class="hot">honey</li>`
3	`<li id="four">balsamic vinegar</li>`

USING INDEX NUMBERS

This example demonstrates how jQuery gives an index number to each of the elements in the jQuery selection.

The `:lt()` and `:gt()` selectors and the `.eq()` method are used to find elements based on their index numbers.

For each of the matching elements, the value of the `class` attributes are changed.

JAVASCRIPT c07/js/index-numbers.js

```
$(function() {
  $('li:lt(2)').removeClass('hot');
  $('li').eq(0).addClass('complete');
  $('li:gt(2)').addClass('cool');
});
```

1. The `:lt()` selector is used in the selector to pick list items with an index number less than 2. It removes the value `hot` from their `class` attribute.

2. The `.eq()` method selects the first item (using the number 0 because the index numbers start at zero). It adds the value of `complete` to the `class` attribute.

3. The `:gt()` selector is used in the jQuery selector to pick the list items with an index number higher than 2. It adds a value of `cool` to their `class` attribute.

RESULT

SELECTING FORM ELEMENTS

jQuery has selectors that are designed specifically to work with forms, however, they are not always the quickest way to select elements.

If you use one of these selectors on its own, jQuery will examine each element in the document to find a match (using code in the jQuery file, which is not as quick as CSS selectors).

Therefore, you should narrow down the part of the document the script needs to look through by placing an element name or other jQuery selector before using the selectors shown on this page.

You can also access elements in a form using the same selectors used to pick any element in jQuery. This will often be the faster option.

It is also worth noting that, because jQuery handles inconsistencies in the way browsers treat whitespace, it is easier to traverse between form elements using jQuery than it is when you are using plain JavaScript.

SELECTORS FOR FORM ELEMENTS

SELECTOR	DESCRIPTION
`:button`	`<button>` and `<input>` elements whose `type` attribute has a value of `button`
`:checkbox`	`<input>` elements whose `type` attribute has a value of `checkbox`. Note that you get better performance with `$('[type="checkbox"]')`
`:checked`	Checked elements from checkboxes and radio buttons (see `:selected` for select boxes)
`:disabled`	All elements that have been disabled
`:enabled`	All elements that are enabled
`:focus`	Element that currently has focus. Note that you get better performance with `$(document.activeElement)`
`:file`	All elements that are file inputs
`:image`	All image inputs. Note that you get better performance using `[type="image"]`
`:input`	All `<button>`, `<input>`, `<select>`, and `<textarea>` elements. Note that you get better performance from selecting elements, then using `.filter(":input")`
`:password`	All password inputs. Note that you get better performance using `$('input:password')`
`:radio`	All radio inputs. To select a group of radio buttons, you can use `$('input[name="gender"]:radio')`
`:reset`	All inputs that are reset buttons
`:selected`	All elements that are selected. Note that you get better performance using a CSS selector inside the `.filter()` method, e.g., `.filter(":selected")`
`:submit`	`<button>` and `<input>` elements whose `type` attribute has a value of `submit`. Note that you will get better performance using `[type="submit"]`
`:text`	Selects `<input>` elements with a `type` attribute whose value is text, or whose `type` attribute is not present. You will likely get better performance from `('input:text')`

FORM METHODS & EVENTS

RETRIEVE THE VALUE OF ELEMENTS

METHOD	DESCRIPTION
`.val()`	Primarily used with `<input>`, `<select>`, and `<textarea>` elements. It can be used to get the value of the first element in a matched set, or update the value of all of them.

OTHER METHODS

METHOD	DESCRIPTION
`.filter()`	Used to filter a jQuery selection using a second selector (especially form-specific filters)
`.is()`	Often used with filters to check whether a form input is selected/checked

`$.isNumeric()`	Checks whether the value represents a numeric value and returns a Boolean. It returns `true` for the following:
	`$.isNumeric(1)` `$.isNumeric(-3)`
	`$.isNumeric("2")` `$.isNumeric(4.4)`
	`$.isNumeric(+2)` `$.isNumeric(0xFF)`

EVENTS

METHOD	DESCRIPTION
`.on()`	Used to handle all events

EVENT	DESCRIPTION
`blur`	When an element loses focus
`change`	When the value of an input changes
`focus`	When an element gains focus
`select`	When the option for a `<select>` element is changed
`submit`	When a form is submitted

When submitting a form, there is also a helpful method called `.serialize()` which you will learn about on p394–p395.

The `.val()` method gets the value of the first `<input>`, `<select>`, or `<textarea>` element in a jQuery selection. It can also be used to set the value for all matching elements.

The `.filter()` and `.is()` methods are commonly used with form elements. You met them on p338.

`$.isNumeric()` is a global method. It is not used on a jQuery selection; rather, the value you want to test is passed as an argument.

All of the event methods on the left correspond to JavaScript events that you might use to trigger functions. As with other jQuery code, they handle the inconsistencies between browsers behind the scenes.

jQuery also makes it easier to work with a group of elements (such as radio buttons, checkboxes, and the options in a select box), because, once you have selected the elements, you can simply apply individual methods to each of them without having to write a loop.

There is an example using forms on the next page, and there are more examples in Chapter 13.

WORKING WITH FORMS

In this example, a button and form have been added under the list. When the user clicks on the button to add a new item, the form will come into view.

The form lets users add a new item to the list with a single text input and a submit button. (The new item button is hidden when the form is in view.)

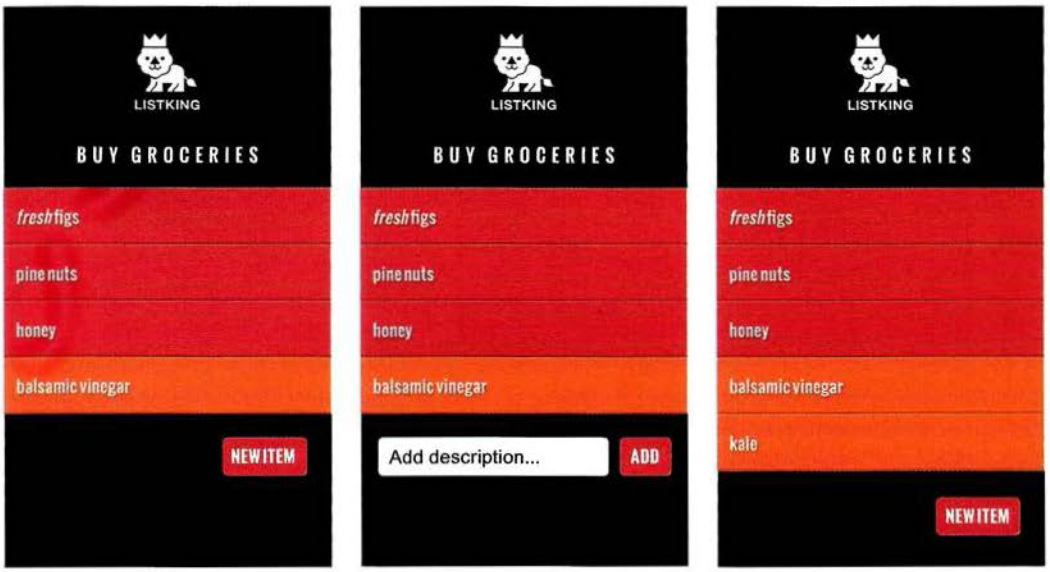
When the user presses the submit button, the new item is added to the bottom of the list. (The form is also hidden and the new item button is shown again.)

c07/form.html

HTML

```
<!-- list goes here -->...</ul>
<div id="newItemButton"><button href="#" id="showForm">new item</button></div>
<form id="newItemForm">
  <input type="text" id="itemDescription" placeholder="Add description..." />
  <input type="submit" id="addButton" value="add" />
</form>
```

RESULT



1. New jQuery objects are created to hold the new item button, the form to add new items, and the add button. These are cached in variables.

2. When the page loads, the CSS hides the new item button (and shows the form), so jQuery methods show the new item button and hide the form.

3. If a user clicks on the new item button (the `<button>` element whose `id` attribute has a value of `showForm`), the new item button is hidden and the form is shown.

JAVASCRIPT

c07/js/form.js

```
$(function() {

  var $newItemButton = $('#newItemButton');
  var $newItemForm = $('#newItemForm');
  var $textInput = $('input:text');

  $newItemButton.show();
  $newItemForm.hide();

  $('#showForm').on('click', function(){
    $newItemButton.hide();
    $newItemForm.show();
  });

  $newItemForm.on('submit', function(e){
    e.preventDefault();
    var newText = $('input:text').val();
    $('li:last').after('<li>' + newText + '</li>');
    $newItemForm.hide();
    $newItemButton.show();
    $textInput.val('');
  });

});
```

4. When the form is submitted, an anonymous function is called. It is passed the `event` object.

5. The `.preventDefault()` method can stop the form being submitted.

6. The `:text` selector picks the `<input>` element whose `type` attribute has a value of `text`, and the `.val()` method gets the value the user entered into it. This value is stored in a variable called `newText`.

7. A new item is added to the end of the list using the `.after()` method.

8. The form is hidden, the new item button is shown again, and the content of the text input is emptied (so the user can add a new entry if they want to).

CUTTING & COPYING ELEMENTS

Once you have a jQuery selection, you can use these methods to remove those elements or make a copy of them.

The `.remove()` method deletes the matched elements and all of their descendants from the DOM tree.

The `.detach()` method also removes the matched elements and all of their descendants from the DOM tree; however, it retains any event handlers (and any other associated jQuery data) so they can be inserted back into the page.

The `.empty()` and `.unwrap()` methods remove elements in relation to the current selection.

The `.clone()` method creates a copy of the matched set of elements (and any descendants). If you use this method on HTML that contains `id` attributes, the value of the `id` attributes would need updating otherwise they would no longer be unique. If you want to pass any event handlers, you should add `true` between the parentheses.

CUT

METHOD	DESCRIPTION
`.remove()`	Removes matched elements from DOM tree (including any descendants and text nodes)
`.detach()`	Same as `.remove()` but keeps a copy of them in memory
`.empty()`	Removes child nodes and descendants from any elements in matched set
`.unwrap()`	Removes parents of matched set, leaving matched elements

COPY

METHOD	DESCRIPTION
`.clone()`	Creates a copy of the matched set (including any descendants and text nodes)

PASTE

You saw how to add elements into the DOM tree on p318.

CUT, COPY, PASTE

In this example, you can see parts of the DOM tree being removed, duplicated, and placed elsewhere on the page.

The HTML has an extra <p> element after the list, which contains a quote. It is moved to a new position under the heading.

In addition, the first list item is detached from the list and moved to the end of it.

JAVASCRIPT

c07/js/cut-copy-paste.js

```
$(function() {
①   var $p = $('p');
②   var $clonedQuote = $p.clone();
③   $p.remove();
④   $clonedQuote.insertAfter('h2');

⑤   var $moveItem = $('#one').detach();
⑥   $moveItem.appendTo('ul');
});
```

RESULT

1. A jQuery selection is made containing the <p> element at the end of the page, and this is cached in a variable called `$p`.

2. That element is copied using the `.clone()` method (along with its content and child elements). It is stored in a variable called `$clonedQuote`.

3. The paragraph is removed.

4. The cloned version of the quote is inserted after the <h2> element at the top of the page.

5. The first list item is detached from the DOM tree and stored in a variable called `$moveItem` (effectively removing it from the DOM tree).

6. That list item is then appended to the end of the list.

BOX DIMENSIONS

These methods allow you to discover or update the width and height of all boxes on the page.

CSS treats each element on a web page as if it were in its own box. A box can have padding, a border, and a margin. If you set the width or height of the box in CSS, it does not include any padding, border, or margin – they are added to the dimensions.

The methods shown here allow you to retrieve the width and height of the first element in the matched set. The first two also allow you to update the dimensions of all boxes in the matched set.

The remaining methods give different measurements depending on whether you want to include padding, border, and a margin. Note how the `.outerHeight()` and `.outerWidth()` methods take a parameter of `true` if you want the margin included.

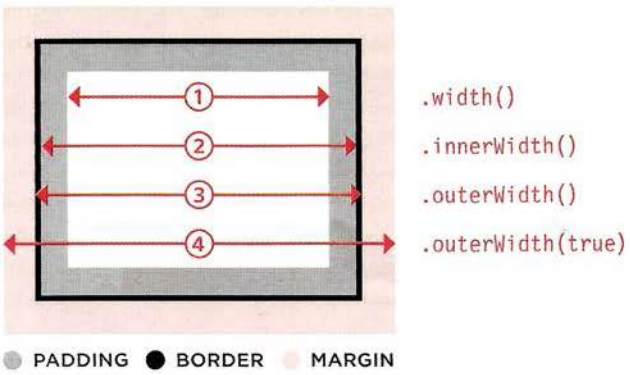
When retrieving dimensions, these methods return a number in pixels.

RETRIEVE OR SET BOX DIMENSIONS

METHOD	DESCRIPTION
`.height()`	Height of box (no margin, border, padding)
`.width()`	Width of box (no margin, border, padding) (1)

RETRIEVE BOX DIMENSIONS ONLY

METHOD	DESCRIPTION
`.innerHeight()`	Height of box plus padding
`.innerWidth()`	Width of box plus padding (2)
`.outerHeight()`	Height of box plus padding and border
`.outerWidth()`	Width of box plus padding and border (3)
`.outerHeight(true)`	Height of box plus padding, border, and margin
`.outerWidth(true)`	Width of box plus padding, border, and margin (4)



CHANGING DIMENSIONS

This example demonstrates how the `.height()` and `.width()` methods can be used to retrieve and update box dimensions.

The page displays the height of the container. It then changes the width of the list items using percentages and pixels.

JAVASCRIPT

c07/js/dimensions.js

```
$(function() {
  var listHeight = $('#page').height();
  $('ul').append('<p>Height: ' + listHeight + 'px</p>');
  $('li').width('50%');
  $('li#one').width(125);
  $('li#two').width('75%');
});
```

1. A variable called `listHeight` is created to store the height of the page container. It is obtained using the `.height()` method.

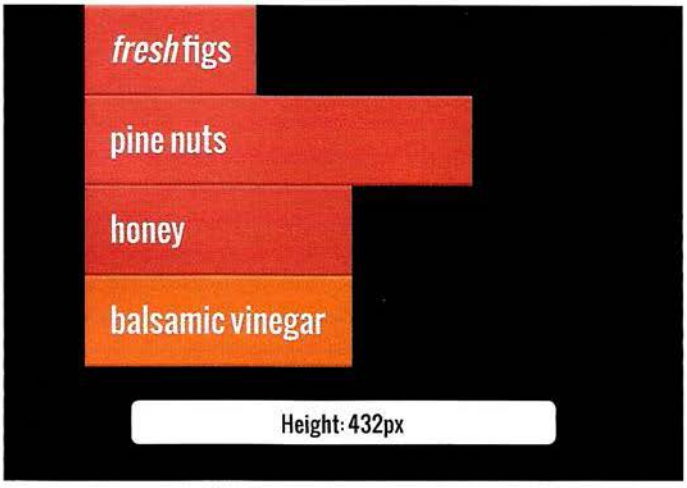
2. The height of the page is written at the end of the list using the `.append()` method and may vary between browsers.

3. The selector picks all the `<li>` elements and sets their width to 50% of their current width using the `.width()` method.

4. These two statements set the width of the first list item to 125 pixels and the width of the second list item to be 75% of the width it was when the page loaded.

Measurements in percentages or ems should be given as a string, with the suffix `%` or `em`. Pixels do not require a suffix and are not enclosed in quotes.

RESULT



WINDOW & PAGE DIMENSIONS

The `.height()` and `.width()` methods can be used to determine the dimensions of both the browser window and the HTML document. There are also methods to get and set the position of the scroll bars.

On p348, you saw that you can get and set the height or width of a box using the `.height()` and `.width()` methods.

These can also be used on a jQuery selection containing the `window` or `document` objects.

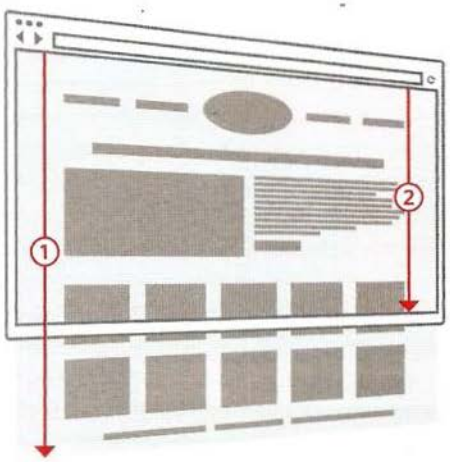
The browser can display scroll bars if the height or width of:

- A box's content is larger than its allocated space.
- The current page represented by the `document` object is larger than the dimensions of the browser window's viewable area (viewport).

The `.scrollLeft()` and `.scrollTop()` methods allow you to get and set the position of the scroll bars.

When retrieving dimensions, these methods return a number in pixels.

METHOD	DESCRIPTION
`.height()`	Height of the jQuery selection
`.width()`	Width of the jQuery selection
`.scrollLeft()`	Gets the horizontal position of the scroll bar for the first element in the jQuery selection, or sets the horizontal scroll bar position for matched nodes
`.scrollTop()`	Gets the vertical position of the scroll bar for the first element in the jQuery selection, or sets the vertical scroll bar position for matched nodes




```
$(document).height();
```

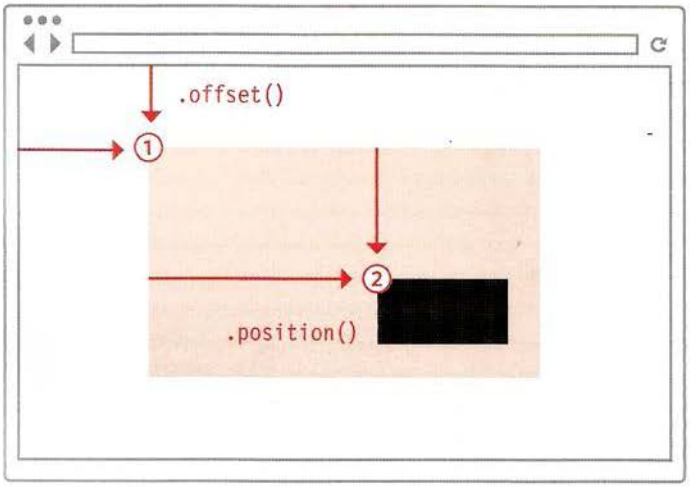
```
$(window).height();
```

This method will often return the incorrect value unless a `DOCTYPE` declaration is specified for the HTML page.

POSITION OF ELEMENTS ON THE PAGE

The `.offset()` and `.position()` methods can be used to determine the position of elements on the page.

METHOD	DESCRIPTION
`.offset()`	Gets or sets coordinates of the element relative to the top left-hand corner of the `document` object **(1)**
`.position()`	Gets or sets coordinates of the element relative to any ancestor that has been taken out of normal flow (using CSS box offsets). If no ancestor is out of normal flow, it will return the same as `.offset()` **(2)**



To get the offset or position, store the object that is returned by these methods in a variable. Then use the `left` or `right` properties of the object to retrieve their position.

```
var offset = $('div').offset();
var text = 'Left: ' + offset.left + ' Right: ' + offset.right;
```

The two methods on the left help you to determine the position of an element:

- Within the page.
- In relation to an ancestor that is offset from normal flow.

Each of them returns an object that has two properties:

`top` - the position from the top of the document or containing element.

`left` - the position from the left of the document or containing element.

As with other jQuery methods, when used to retrieve information, they return the co-ordinates of the first element in the matched set.

If they are used to set the position of elements, they will update the position of all elements in the matched set (putting them in the same spot).

DETERMINING POSITION OF ITEMS ON THE PAGE

In this example, as the user scrolls down the page, a box slides into view if they get within 500 pixels of the footer.

We will call this part of the page the end zone, and you need to work out the height at which the endZone starts.

Every time the user scrolls, you then check the position of the scroll bar from the top of the page.

If the scroll bar is further down the page than the start of the end zone, the box is animated into the page. If not, then the box is hidden.

The HTML for this example contains an extra `<div>` element at the end of the page containing the advert. A lot of items have been added to the list to create a long page that scrolls.

c07/position.html

HTML

```
    ...<li>quinoa</li>
    </ul>
    <p id="footer">&copy; ListKing</p>
    <div id="slideAd">
      Buy ListKing Pro for only $1.99
    </div>
  </div>
  <script src="js/jquery-1.9.1.min.js"></script>
  <script src="js/position.js"></script>
```

RESULT

1. Cache the window and advert.

2. The height of the end zone is calculated, and stored in a variable called endZone.

3. The `scroll` event triggers an anonymous function every time the user scrolls up or down.

4. A conditional statement checks if the user's position is further from the top of the page than the start of the end zone.

5. If the condition returns `true`, the box slides in from the right-hand edge of the page. This takes 250 milliseconds.

6. If the condition is false or the box is in the middle of animating, it is stopped using the `.stop()` method. The advert then slides off the right-hand edge of the page. Again, this animation will take 250 milliseconds.

JAVASCRIPT

c07/js/position.js

```
$(function() {
  var $window = $(window);
  var $slideAd = $('#slideAd');
  var endZone = $('#footer').offset().top - $window.height() - 500;

  $window.on('scroll', function() {

    if  ( (endZone) < $window.scrollTop() ) {
      $slideAd.animate({ 'right': '0px' }, 250);
    } else {
      $slideAd.stop(true).animate({ 'right': '-360px' }, 250);
    }

  });

});
```

CALCULATING THE END ZONE

Calculate the height at which the box should come into view by:

a) Getting the height from the top of the page to the top of the footer (the gray bar) in pixels.

b) Subtracting the height of the viewport from this result.

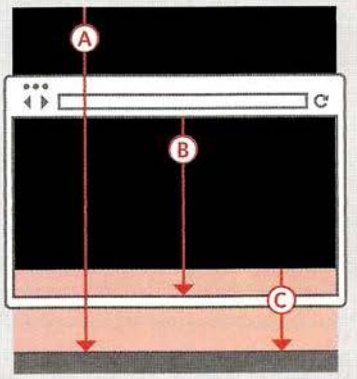
c) Subtracting a further 500px for the area where the box will come into view (shown in pink).

You can tell how far the user has scrolled down the page using:

```
$(window).scrollTop();
```

If the distance extends down further than the height at which the end zone should show, the box should be made visible.

If not, then the box should move off the page.

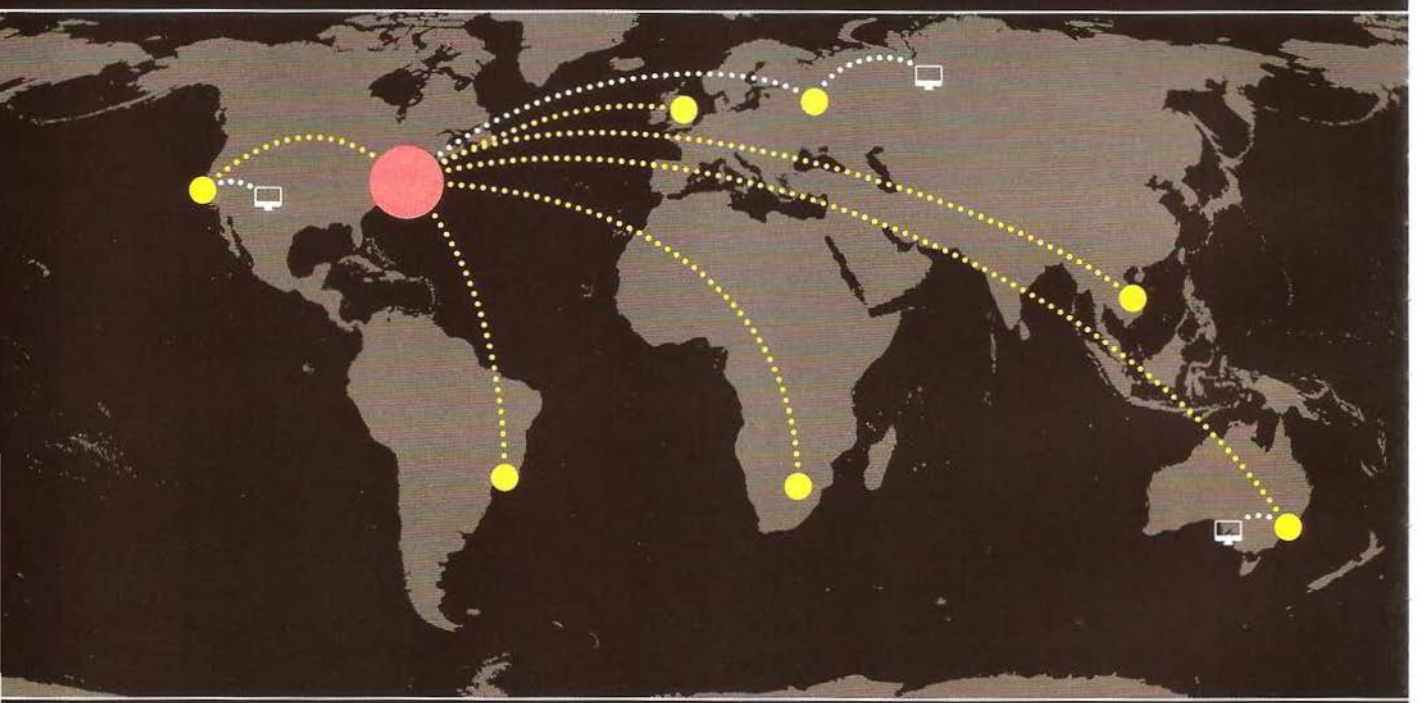


WAYS TO INCLUDE JQUERY IN YOUR PAGE

In addition to hosting the jQuery file with the rest of your website, you can also use a version that is hosted by other companies. However, you should still include a fallback version.

At the time of writing, the main CDNs to offer jQuery are jQuery CDN (powered by Max CDN), Google, and Microsoft.

● ORIGIN ● CDN ● USER



A Content Delivery Network (or CDN) is a series of servers spread out around the world. They are designed to serve static files (such as HTML, CSS, JavaScript, images, audio, and video files) very quickly.

The CDN tries to find a server near you, then sends files from that server so the data does not travel as far. With jQuery, users might have already downloaded and cached the file from a CDN when visiting another site.

When including jQuery in your pages, you can try to load it from one of these CDNs. Then you check if it loaded, and if not, you can include a version that is stored on your own servers (this is known as a fallback).

LOADING JQUERY FROM A CDN

When a page loads jQuery from a CDN, you will often see a syntax like the one shown below. It starts with a `<script>` tag that tries to load the jQuery file from the CDN. But note that the URL for the script starts with two forward slashes (not `http:`).

This is known as a **protocol relative URL**. If the user is looking at the current page through `https`, then they will not see an error that tells them there are unsecure items on the page. **Note:** This does not work locally with the `file://` protocol.

This is often followed by a second `<script>` tag that contains a logical operator, which checks to see if jQuery has loaded. If it has not loaded, the browser tries to load the jQuery script from the same server as the rest of the website.

HTML

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js">
</script>

<script>
window.jQuery || document.write('<script src="js/jquery-1.10.2.js"><\/script>')
</script>
```

The logical operator looks for the `jQuery` object that the jQuery script makes available. If it exists, then a truthy value is returned and the logical operator short circuits (see p157).

If jQuery has not loaded, then the `document.write()` method is used to add a new `<script>` tag into the page. This will load a version of jQuery from the same server as the rest of the website.

The fallback option is important because the CDN may be unavailable, the file may have moved, and some countries ban some domain names (such as Google).

WHERE TO PLACE YOUR SCRIPTS

The position of <script> elements can affect how quickly a web page seems to load.

SPEED

In the early days of the web, developers were told to place the <script> tags in the <head> of the page as you do with style sheets. However, this can make pages seem slower to load.

Your web page may use files from several different locations (e.g., images or CSS files might be loaded from one CDN, jQuery could be loaded from the jQuery or Google CDNs, and fonts might be loaded from another third party).

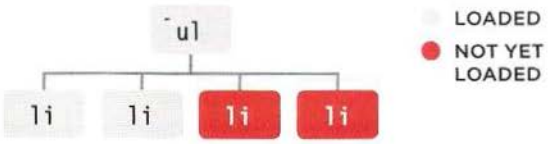
Usually a browser will collect up to two files at a time from each different server. However, when a browser starts to download a JavaScript file, it stops all other downloads and pauses laying out the page until the script has finished loading and been processed.

Therefore, if you place the script at the end of the page before the closing </body> tag, it will not affect the rendering of the rest of the page.

HTML LOADED INTO THE DOM TREE

Whenever a script is accessing the HTML within a web page, it also needs to have loaded that HTML into the DOM tree before the script can work. (This is often referred to as the DOM having loaded.)

You can use the `load` event to trigger a function so that you know the HTML has loaded. However, it fires only when the page and all of its resources load. You can also use the HTML5 `DOMContentLoaded` event, but it does not work in older browsers.



If the script tries to access an element before it has loaded, it causes an error. In the diagram above, the script could access the first two <li> elements, but not the third or fourth.

Where possible, do consider using alternatives to scripts. For example, use CSS for animations or HTML5's `autofocus` attribute rather than using the `load` event to bring focus to an element.

If your page is slow to load and you only want to include a small amount of code before the rest of the page has loaded, you can place a <script> tag within the body of the page.

At the time of writing, this technique was commonly used by Google for speed advantages, but it is acknowledged that it makes code much harder to maintain.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sample Page</title>
    <link rel="stylesheet" href="sample.css" />
    <script src="js/sample.js"></script>
  <head>
  <body>
    <h1>Sample Page</h1>
    <div id="page">Main content here...</div>
  </body>
</html>
```

IN THE HEAD

This location is best avoided as:
1. Pages seem slower to load.
2. DOM content is not loaded, when the script is executed so you have to wait for an event like **`load`** or **`DOMContentLoaded`** to trigger your functions.

If you must use a **`<script>`** element within the head of the page, it should be just before the closing **`</head>`** tag.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sample Page</title>
    <link rel="stylesheet" href="sample.css" />
  <head>
  <body>
    <h1>Sample Page</h1>
    <script src="js/sample.js"></script>
    <div id="page">Main content here...</div>
  </body>
</html>
```

IN THE PAGE

As with scripts in the **`<head>`**, those in the middle of the page will slow the rest of the page down when it is loading.

If you use **`document.write()`**, the **`<script>`** element has to be placed where you want that content to appear. This is one of several good reasons to avoid using **`document.write()`**.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sample Page</title>
    <link rel="stylesheet" href="sample.css" />
  <head>
  <body>
    <h1>Sample Page</h1>
    <div id="page">Main content here...</div>
    <script src="js/sample.js"></script>
  </body>
</html>
```

BEFORE THE CLOSING </body> TAG

This is an ideal location as:
1. The script is not blocking other things from downloading.
2. The DOM has already loaded by the time the script is executed.

JQUERY DOCUMENTATION

For an exhaustive list of the functionality provided in jQuery, visit `http://api.jquery.com`

It is not possible to teach you everything about jQuery in one (albeit long) chapter. But you have seen many of the most popular features, and you should now know enough about jQuery to understand how it works and how to make use of it in your scripts.

Throughout the remaining chapters of this book, you will see many more examples that use jQuery.

What you have learned should also give you enough experience to work with the comprehensive jQuery documentation available online at: `http://api.jquery.com`

This site lists each method and property available to you, along with new functionality added in the latest versions, and notes that indicate which features are scheduled to be dropped.

HOW THE DOCUMENTATION WORKS

On the left-hand side of the page, you will see the different types of functionality that you can explore.

When you click on any of the methods in the main column, you will see a list of the parameters that it can take. When parameters are optional, they are shown in square brackets.

You will also find deprecated methods. This means that you are no longer advised to use this markup because it is likely to be removed from future versions of jQuery (if it has not already been removed).

EXTENDING JQUERY WITH PLUGINS

Plugins are scripts that extend the functionality of the jQuery library. Hundreds have been written and are available for you to use.

Plugins offer functionality that is not included in the jQuery library. They usually deal with a particular task such as creating slideshows or video players, performing animations, transforming data, enhancing forms, and displaying new data from a remote server.

To get an idea of the number and range of plugins available, see `http://plugins.jquery.com`. All of these are free for you to download and use on your own sites. You may also find other sites listing jQuery plugins for sale (such as `codecanyon.net`).

Plugins are written so that new methods extend the `jQuery` object and can, therefore, be used on a jQuery selection. As long as you know how to do the following with jQuery:

- Make a selection of elements
- Call a method and use parameters

You can use a lot of the functionality of these plugins without having to write the code yourself. In Chapter 11, you will see an example of how to create a basic jQuery plugin.

HOW TO CHOOSE A PLUGIN

When you are choosing a plugin to work with, it can be worth checking that it is still being maintained or whether other people have experienced problems using it. Finding out the following can help:

- When was the plugin last updated?
- How many people are watching the plugin?
- What do the bug reports say?

If you ask a question or find a bug in a script, bear in mind that their authors may have a day job and only maintain these plugins in their spare time to help others and to give back to the community.

JAVASCRIPT LIBRARIES

jQuery is an example of what programmers call a JavaScript library. It is a JavaScript file that you include in your page, which then lets you use the functions, objects, methods, and properties it contains.

The concept of a library is that it allows you to borrow code from one file and use its functions, objects, methods, and properties in another script.

Once you have included the script in your page, its functionality is available to use. The documentation for the library will tell you how to use it.

jQuery is the most widely used library on the web, but when you have learned it, you might like to explore some of the other libraries listed below.

Popular libraries have the advantage that they will be well-tested, and some have a whole team of developers who work on them in their spare time.

One of the main drawbacks with a library is that they will usually contain functionality that you will not need to use. This means users have to download code that will not be needed (which can slow your site down). You may find that you can strip out the subset of the library you need or indeed write your own script to do that job.

DOM & EVENTS

Zepto.js
YUI
Dojo.js
MooTools.js

USER INTERFACE

jQuery UI
jQuery Mobile
Twitter Bootstrap
YUI

GRAPHICS & CHARTS

Chart.js
D3.js
Processing.js
Raphael.js

TEMPLATING

Mustache.js
Handlebars.js
jQuery Mobile

WEB APPLICATIONS

Angular.js
Backbone.js
Ember.js

COMPATIBILITY

Modernizr.js
YepNope.js
Require.js

PREVENTING CONFLICTS WITH OTHER LIBRARIES

Earlier in the chapter, you saw that `$()` was shorthand for `jQuery()`. The $ symbol is used by other libraries such as prototype.js, MooTools, and YUI. To avoid conflicts with those scripts, use these techniques.

INCLUDING JQUERY AFTER OTHER LIBRARIES

Here, jQuery's meaning of $ takes precedence:

```
<script src="other.js"></script>
<script src="jquery.js"></script>
```

You can use the `.noConflict()` method at the start of your script, to tell jQuery to release the $ shortcut so that other scripts can use it. Then you can use the full name rather than the shortcut:

```
jQuery.noConflict();
jQuery(function() {
    jQuery('div').hide();
});
```

You can wrap your script in an IIFE and still use $:

```
jQuery.noConflict();
(function($) {
    $('div').hide();
})(jQuery);
```

Or you can specify your own alias instead, e.g., `$j`:

```
var $j = jQuery.noConflict();
$j(document).ready(function() {
    $j('div').hide();
});
```

INCLUDING JQUERY BEFORE OTHER LIBRARIES

Here, the other scripts' use of $ takes precedence:

```
<script src="jquery.js"></script>
<script src="other.js"></script>
```

$ will have the meaning defined in the other library. There is no need to use the `.noConflict()` method because it will have no effect. But you can continue to use the full name `jQuery`:

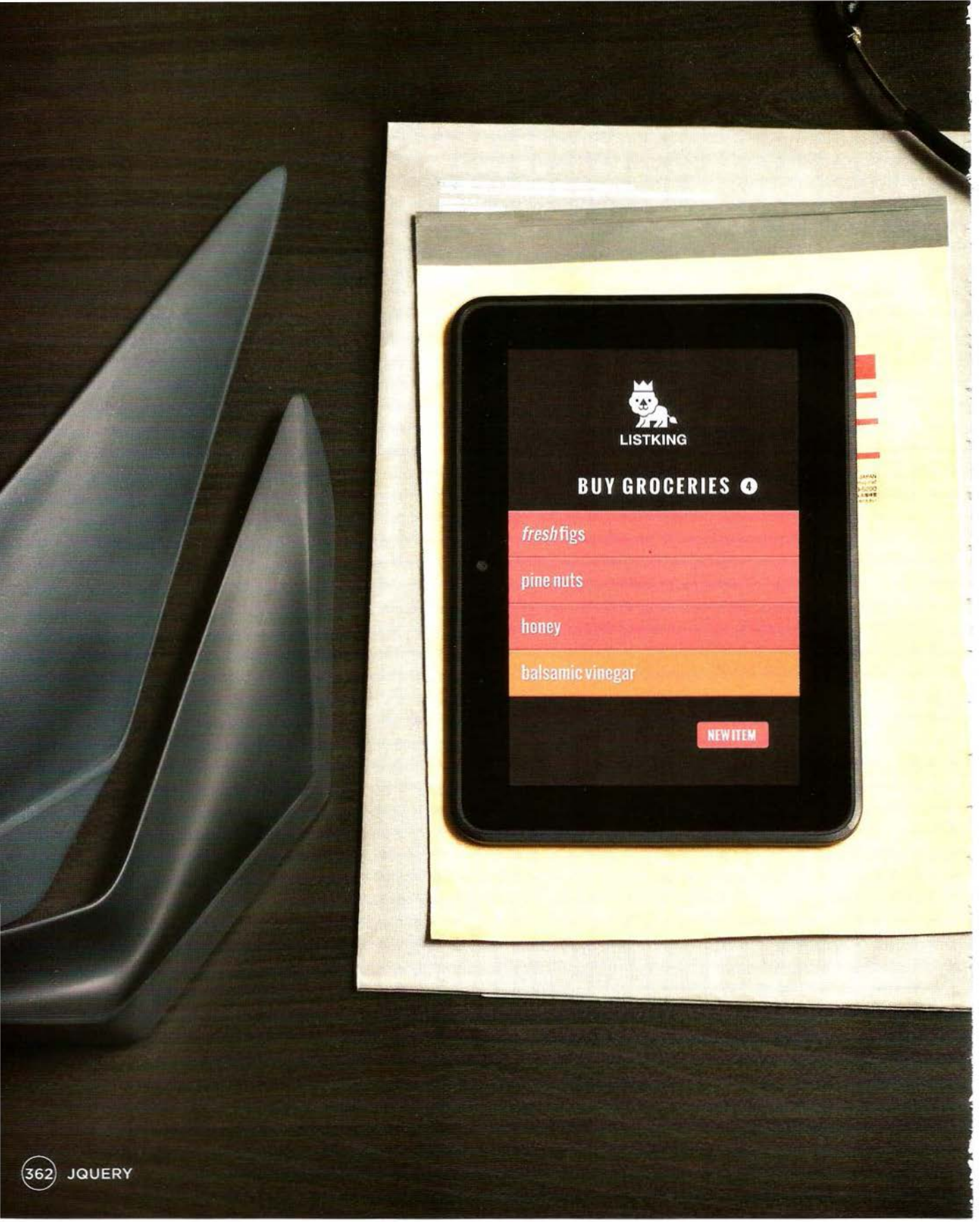
```
jQuery(document).ready(function() {
    jQuery('div').hide();
});
```

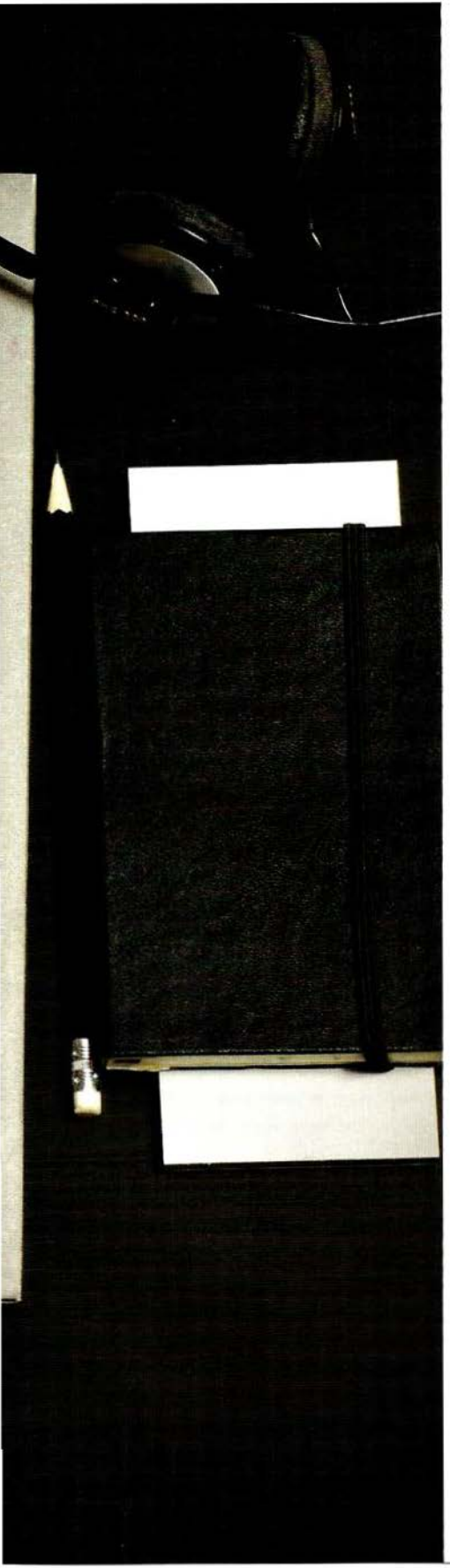
You can pass $ as an argument to the anonymous function called by the `.ready()` method like so:

```
jQuery(document).ready(function($) {
    $('div').hide();
});
```

This is equivalent to the code shown above:

```
jQuery(function($){
    $('div').hide();
});
```


LISTKING
BUY GROCERIES 4
fresh figs
pine nuts
honey
balsamic vinegar
NEW ITEM



EXAMPLE
JQUERY

This example brings together a number of the techniques you have seen in this chapter to create a list that users can add items to and remove items from.

- Users can add new list items.
- They can also click to indicate that an item is complete (at which point it is moved to the bottom of the list and marked as `complete`).
- Once an item is marked as `complete`, a second click on the item will remove it from the list.

An updated count of the number of items there are in the list will be shown in the heading.

As you will see, the code using jQuery is more compact than it would be if you were writing this example in plain JavaScript, and it will work across browsers even though there is no explicit fallback code.

Because new items can be added to the list, the events are handled using event delegation. When the user clicks anywhere on the `<ul>` element, the `.on()` event method handles the event. Inside the event handler, there is a conditional statement to check whether the list item is:

- Not complete - in which case, the click is used to change the item to complete, move it to the bottom of the list, and update the counter.
- Complete - in which case, the second click on the item fades it out and removes it from the list altogether.

The use of conditional statements and custom functions (used for the counter) illustrate how jQuery techniques are used in combination with traditional JavaScript that you have been learning throughout the book.

The appearance and removal of the elements is also animated, and these animations demonstrate how methods can be chained together to create complex interactions based on the same selection of elements.

EXAMPLE
JQUERY

c07/js/example.js

JAVASCRIPT

```
$(function() {

  // SETUP
  var $list, $newItemForm, $newItemButton;
  var item = '';                                  // item is an empty string
  $list = $('ul');                                // Cache the unordered list
  $newItemForm = $('#newItemForm');               // Cache form to add new items
  $newItemButton = $('#newItemButton');           // Cache button to show form

  $('li').hide().each(function(index) {           // Hide list items
    $(this).delay(450 * index).fadeIn(1600);      // Then fade them in
  });

  // ITEM COUNTER
  function updateCount() {                        // Declare function
    var items = $('li[class!=complete]').length; // Number of items in list
    $('#counter').text(items);                    // Added into counter circle
  }
  updateCount();                                  // Call the function

  // SETUP FORM FOR NEW ITEMS
  $newItemButton.show();                          // Show the button
  $newItemForm.hide();                            // Hide the form
  $('#showForm').on('click', function() {         // When new item clicked
    $newItemButton.hide();                        // Hide the button
    $newItemForm.show();                          // Show the form
  });
```

The entire script will wait until the DOM is ready before running, because it is inside the shorthand for the `document.ready()` method. Variables are created that will be used in the script, including jQuery selections that need to be cached.

The `updateCounter()` function checks how many items are in the list and writes it into the heading. It is called straight away to calculate how many list items are on the page when it loads, and then write that number next to the heading.

The form to add new items is hidden when the page loads, and is shown when the user clicks on the add button. When the user clicks on the add button a new item is added to the form and the `updateCounter()` is called.

EXAMPLE
JQUERY

JAVASCRIPT

c07/js/example.js

```
// ADDING A NEW LIST ITEM
$newItemForm.on('submit', function(e) {          // When a new item is submitted
  e.preventDefault();                            // Prevent form being submitted
  var text = $('input:text').val();              // Get value of text input
  $list.append('<li>' + text + '</li>');         // Add item to end of the list
  $('input:text').val('');                       // Empty the text input
  updateCount();                                 // Update the count
});

// CLICK HANDLING - USES DELEGATION ON <ul> ELEMENT
$list.on('click', 'li', function() {
  var $this = $(this);                 // Cache the element in a jQuery object
  var complete = $this.hasClass('complete'); // Is item complete

  if (complete === true) {             // Check if item is complete
    $this.animate({                    // If so, animate opacity + padding
      opacity: 0.0,
      paddingLeft: '+=180'
    }, 500, 'swing', function() {      // Use callback when animation completes
      $this.remove();                  // Then completely remove this item
    });
  } else {                             // Otherwise indicate it is complete
    item = $this.text();               // Get the text from the list item
    $this.remove();                    // Remove the list item
    $list                              // Add back to end of list as complete
      .append('<li class=\"complete\">' + item + '</li>')
      .hide().fadeIn(300);             // Hide it so it can be faded in
    updateCount();                     // Update the counter
  }                                    // End of else option
});                                    // End of event handler

});
```

The `.on()` event method listens for the user clicking anywhere on the list because this script uses event delegation. When they do, the element that was clicked on is stored in a jQuery object and cached in a variable called `$this`.

Next, the code checks if that element has a class name of `complete`. If it does, then the list item is animated out of view and removed. If it was not already complete, then it is moved to the end of the list.

When it is added to the end of the list, its `class` attribute is given a value of `complete`.

Finally, `updateCount()` is called to update the number of items left to do on the list.

SUMMARY

JQUERY

- jQuery is a JavaScript file you include in your pages.
- Once included, it makes it faster and easier to write cross-browser JavaScript, based on two steps:

  1. Using CSS-style selectors to collect one or more nodes from the DOM tree.

  2. Using jQuery's built-in methods to work with the elements in that selection.
- jQuery's CSS-style selector syntax makes it easier to select elements to work with. It also has methods that make it easier to traverse the DOM.
- jQuery makes it easier to handle events because the event methods work across all browsers.
- jQuery offers methods that make it quick and simple to achieve a range of tasks that JavaScript programmers commonly need to perform.

8

AJAX & JSON

Ajax is a technique for loading data into part of a page without having to refresh the entire page. The data is often sent in a format called JavaScript Object Notation (or JSON).

The ability to load new content into part of a page improves the user experience because the user does not have to wait for an entire page to load if only part of it is being updated. This has led to a rise in so-called single page web applications (web-based tools that feel more like software applications, even though they run in the browser). This chapter covers:

WHAT AJAX IS

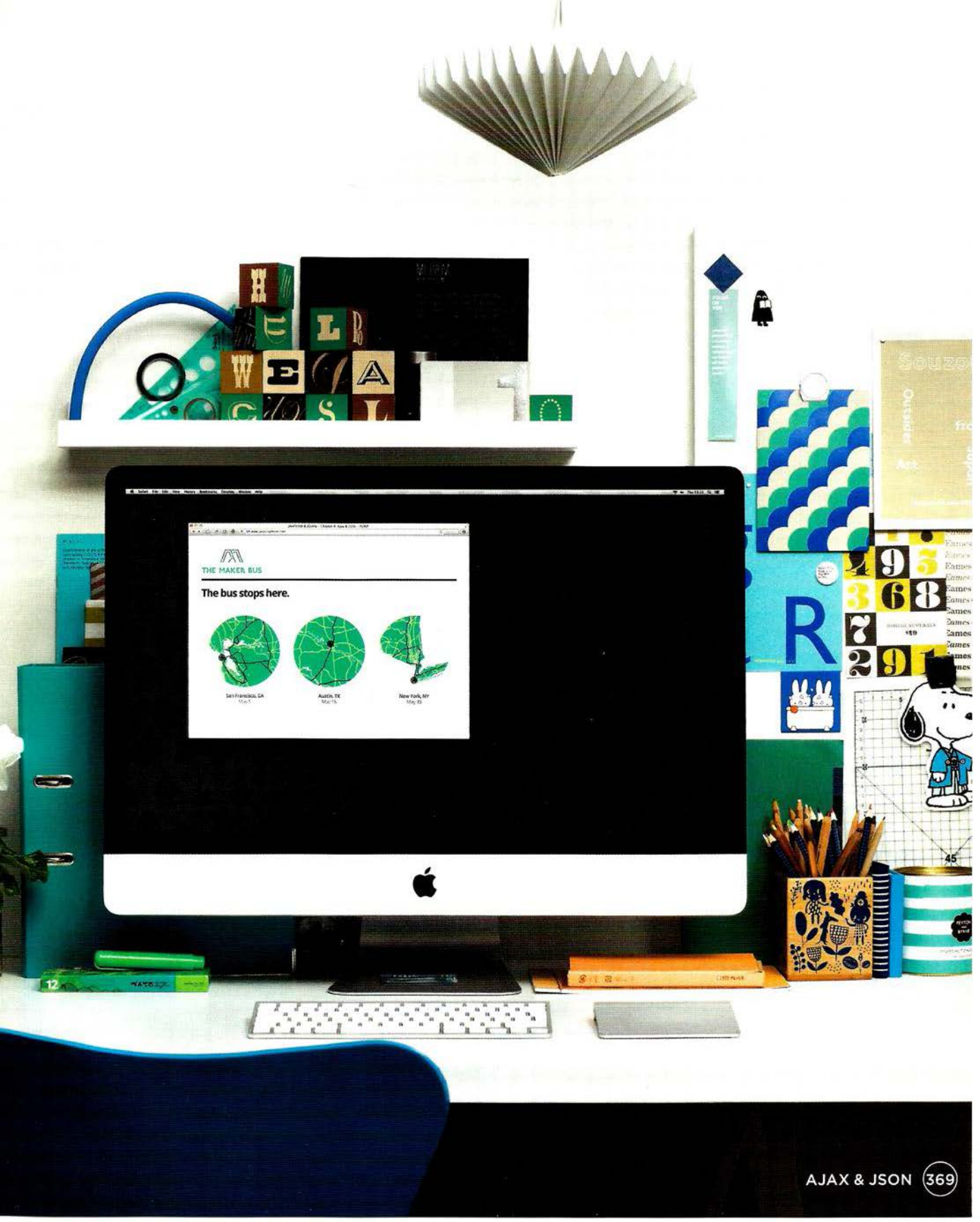
Ajax allows you to request data from a server and load it without having to refresh the entire page.

DATA FORMATS

Servers typically send back HTML, XML, or JSON, so you will learn about these formats.

JQUERY & AJAX

jQuery makes it easier to create Ajax requests and process the data the server returns.


THE MAKER BUS
The bus stops here.
San Francisco, CA
Austin, TX
New York, NY

WHAT IS AJAX?

You may have seen Ajax used on many websites, even if you were not aware that it was being used.

Live search (or autocomplete) commonly uses Ajax. You may have seen it used on the Google website. When you type into the search bar on the home page, sometimes you will see results coming up before you have finished typing.

Websites with user-generated content (such as Twitter and Flickr) may allow you to display your information (such as your latest tweets or photographs) on your own website. This involves collecting data from their servers.

Sometimes when you are shopping online and add items to your shopping cart, it is updated without you leaving the page. At the same time, the site may display a message confirming the item was added.

If you are registering for a website, a script may check whether your username is available before you have completed the rest of the form.

Sites may also use Ajax to load data behind the scenes so that they can use or show that data later on.

WHY USE AJAX?

Ajax uses an asynchronous processing model. This means the user can do other things while the web browser is waiting for the data to load, speeding up the user experience.

USING AJAX WHILE PAGES ARE LOADING

When a browser comes across a `<script>` tag, it will typically stop processing the rest of the page until it has loaded and processed that script. This is known as a **synchronous processing model**.

When a page is loading, if a script needs to collect data from a server (e.g., if it collects financial exchange rates or status updates), then the browser would not just wait for the script to be loaded and processed; it would also have to wait for a server to send the data that the script is going to display.

With Ajax, the browser can request some data from a server and – once that data has been requested – continue to load the rest of the page and process the user's interactions with the page. It is known as an **asynchronous** (or **non-blocking**) **processing model**.

The browser does not wait for the third party data in order to show the page. When the server responds with the data, an event is fired (like the `load` event that fires when a page has loaded). This event can then call a function that processes the data.

USING AJAX WHEN PAGES HAVE LOADED

Once a page has loaded, if you want to update what the user sees in the browser window, typically you would refresh the entire page. This means that the user has to wait for a whole new page to download and be rendered by the browser.

With Ajax, if you only want to update a *part* of the page, you can just update the content of one element. This is done by intercepting an event (such as the user clicking on a link or submitting a form) and requesting the new content from the server using an asynchronous request.

While that data is loading, the user can continue to interact with the rest of the page. Then, once the server has responded, a special Ajax event will trigger another part of the script that reads the new data from the server and updates just that one part of the page.

Because you do not have to refresh the whole page, the data will load faster and the user can still use the rest of the page while they are waiting.

Historically, AJAX was an acronym for the technologies used in asynchronous requests like this. It stood for Asynchronous JavaScript And XML. Since then, technologies have moved on and the term Ajax is now used to refer to a *group* of technologies that offer asynchronous functionality in the browser.

HOW AJAX WORKS

When using Ajax, the browser requests information from a web server. It then processes the server's response and shows it within the page.

1

THE REQUEST

The browser requests information from the server.

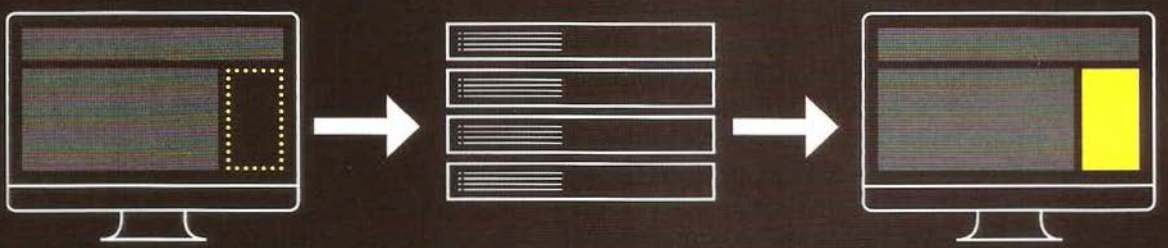
ON THE SERVER

The server responds with data (usually HTML, XML, or JSON).

2

THE RESPONSE

The browser processes the content and adds it to the page.



The browser requests data from the server. The request may include information that the server needs - just like a form might send data to a server.

Browsers implement an object called `XMLHttpRequest` to handle Ajax requests. Once a request has been made, the browser does not wait for a response from the server.

What happens on the server is not part of what is called Ajax.

Server-side technologies such as ASP.net, PHP, NodeJS, or Ruby can generate web pages for each user. When there is an Ajax request, the server might send back HTML, or it might send data in a different format such as JSON or XML (which the browser turns into HTML).

When the server has finished responding to the request, the browser will fire an event (just like it can fire an event when a page has finished loading).

This event can be used to trigger a JavaScript function that will process the data and incorporate it into one part of the page (without affecting the rest of the page).

HANDLING AJAX REQUESTS & RESPONSES

To create an Ajax request, browsers use the `XMLHttpRequest` object. When the server responds to the browser's request, the same `XMLHttpRequest` object will process the result.

THE REQUEST

```
① var xhr = new XMLHttpRequest();
② xhr.open('GET', 'data/test.json', true);
③ xhr.send('search=arduino');
```

1. An instance of the `XMLHttpRequest` object is created using object constructor notation (which you met on p106). It uses the `new` keyword and stores the object in a variable. The variable name `xhr` is short for **`XMLHttpRequest`** (the name of the object).

2. The `XMLHttpRequest` object's `open()` method prepares the request. It has three parameters (which you meet on p379):
i) The HTTP method
ii) The url of the page that will handle your request
iii) A Boolean indicating if it should be asynchronous

3. The `send()` method is the one that sends the prepared request to the server. Extra information can be passed to the server in the parentheses. If no extra information is sent, you may see the keyword `null` used (although it is not strictly needed): `xhr.send(null)`.

THE RESPONSE

```
① xhr.onload = function() {
②   if (xhr.status === 200) {
      // Code to process the results from the server
    }
  }
```

1. When the browser has received and loaded a response from the server, the `onload` event will fire. This will trigger a function (here, it is an anonymous function).

2. The function checks the `status` property of the object. This is used to make sure the server's response was okay. (If this property is blank, check the setup of the server.)

Note that IE9 was the first version of IE to support this way of dealing with Ajax responses. To support older browsers, you can use jQuery (see p388).

DATA FORMATS

The response to an Ajax request usually comes in one of three formats: HTML, XML, or JSON. Below is a comparison of these formats. XML and JSON are introduced over the next three pages.

HTML

You are probably most familiar with HTML, and, when you want to update a section of a web page, it is the simplest way to get data into a page.

BENEFITS

- It is easy to write, request, and display.
- The data sent from the server goes straight into the page. There's no need for the browser to process it (as with the other two methods).

DRAWBACKS

- The server must produce the HTML in a format that is ready for use on your page.
- It is not well-suited for use in applications other than web browsers. It does not have good **data portability**.
- The request must come from the same domain* (see below).

XML

XML looks similar to HTML, but the tag names are different because they describe the data that they contain. The syntax is also more strict than HTML.

BENEFITS

- It is a flexible data format and can represent complex structures.
- It works well with different platforms and applications.
- It is processed using the same DOM methods as HTML.

DRAWBACKS

- It is considered a verbose language because the tags add a lot of extra characters to the data being sent.
- The request must come from the same domain as the rest of the page* (see below).
- It can require a lot of code to process the result.

JSON

JavaScript Object Notation (JSON) uses a similar syntax to object literal notation (which you met on p102) in order to represent data.

BENEFITS

- It can be called from any domain (see JSON-P/CORS).
- It is more concise (less verbose) than HTML/XML.
- It is commonly used with JavaScript (and is gaining wider use across web applications).

DRAWBACKS

- The syntax is not forgiving. A missed quote, comma, or colon can break the file.
- Because it is JavaScript, it can contain malicious content (see XSS on p228). Therefore, you should only use JSON that has been produced by trusted sources.

* Browsers only let Ajax load HTML and XML from the same domain name as the rest of the page (e.g., if the page is on `www.example.com`, the Ajax request must return data from `www.example.com`).

XML: EXTENSIBLE MARKUP LANGUAGE

XML looks a lot like HTML, but the tags contain different words. The purpose of the tags is to describe the kind of data that they hold.

```
<?xml version="1.0" encoding="utf-8" ?>
<events>
  <event>
    <location>San Francisco, CA</location>
    <date>May 1</date>
    <map>img/map-ca.png</map>
  </event>
  <event>
    <location>Austin, TX</location>
    <date>May 15</date>
    <map>img/map-tx.png</map>
  </event>
  <event>
    <location>New York, NY</location>
    <date>May 30</date>
    <map>img/map-ny.png</map>
  </event>
</events>
```

You can process an XML file using the same DOM methods as HTML. Because different browsers deal with whitespace in HTML/XML documents in different ways, it is easier to process XML using jQuery rather than plain JavaScript (just as it can be with HTML).

In the same way that HTML is a markup language that can be used to describe the structure and semantics of a web page, XML can be used to create markup languages for other types of data - anything from stock reports to medical records.

The tags in an XML file should describe the data they contain. As a result, even if you have never seen the code to the left, you can see that the data describes information about several events. The `<events>` element contains several individual events. Each individual event is represented in its own `<event>` element.

XML works on any platform and gained wide popularity in the early 2000s because it made it easy to transfer data between different types of applications. It is also a very flexible data format because it is capable of representing complex data structures.

JSON: JAVASCRIPT OBJECT NOTATION

Data can be formatted using JSON (pronounced "Jason").
It looks very similar to object literal syntax, but it is not an object.

JSON data looks like the object literal notation which you met on p102; however, it is just plain text data (not an object).

The distinction may sound small but remember that HTML is just plain text, and the browser converts it into DOM objects.

You cannot transfer the actual objects over a network. Rather, you send text which is converted into objects by the browser.

```
{
  "location": "San Francisco, CA",
  "capacity": 270,
  "booking":  true
}
```

KEY (in double quotes)

VALUE

KEYS

In JSON, the key should be placed in **double quotes** (not single quotes).

The key (or name) is separated from its value by a colon.

Each key/value pair is separated by a comma. However, note that there is *no* comma after the last key/value pair.

VALUES

The value can be any of the following data types (some of these are demonstrated above; others are shown on the right-hand page):

DATA TYPE	DESCRIPTION
`string`	Text (must be written in quotes)
`number`	Number
`Boolean`	Either `true` or `false`
`array`	Array of values - this can also be an array of objects
`object`	JavaScript object - this can contain child objects or arrays
`null`	This is when the value is empty or missing

WORKING WITH JSON DATA

JavaScript's JSON object can turn JSON data into a JavaScript object. It can also convert a JavaScript object into a string.

```
{
  "events": [
    {
      "location": "San Francisco, CA",
      "date": "May 1",
      "map": "img/map-ca.png"
    },
    {
      "location": "Austin, TX",
      "date": "May 15",
      "map": "img/map-tx.png"
    },
    {
      "location": "New York, NY",
      "date": "May 30",
      "map": "img/map-ny.png"
    }
  ]
}
```

● OBJECT ● ARRAY

The object on the left represents a series of three events, stored in an array called `events`. The array uses square bracket notation, and it holds three objects (one for each event).

`JSON.stringify()` converts JavaScript objects into a string, formatted using JSON. This allows you to send JavaScript objects from the browser to another application.

`JSON.parse()` processes a string containing JSON data. It converts the JSON data into a JavaScript objects ready for the browser to use.

Browser support: Chrome 3, Firefox 3.1, IE8, and Safari 4

An object can also be written on one line, as you can see here:

```
{
  "events": [
    { "location": "San Francisco, CA", "date": "May 1", "map": "img/map-ca.png" },
    { "location": "Austin, TX", "date": "May 15", "map": "img/map-tx.png" },
    { "location": "New York, NY", "date": "May 30", "map": "img/map-ny.png" }
  ]
}
```

LOADING HTML WITH AJAX

HTML is the easiest type of data to add into a page using Ajax.
The browser renders it just like any other HTML.
The CSS rules for the rest of the page are applied to the new content.

Below, the example loads data about three events using Ajax. (The result will look the same for the next four examples.)

The page users open does not hold the event data (highlighted in pink). Ajax is used to load it into the page from another file.

HIGHLIGHTED AREA LOADED USING AJAX

When a server responds to any request, it should send back a status message, to indicate if it completed the request. The values can be:

200 The server has responded and all is ok
304 Not modified
404 Page not found
500 Internal error on the server

If you run the code locally, you will not get a server status property, so this check must be commented out, and return `true` for the condition. If a server fails to return a `status` property, check the server setup.

Browsers will only let you use this technique to load HTML that comes from the same domain name as the rest of the page.

Whether HTML, XML, or JSON is being returned from the server, the process of setting up the Ajax request and checking whether the file is ready to be worked with is the same. What changes is how you deal with the data that is returned.

In the example on the right-hand page, the code to display the new HTML is placed inside a conditional statement.

Please note: These examples do not work locally in Chrome. They should work locally in Firefox and Safari. IE support is mixed until IE9.

Later in the chapter, you will see that jQuery offers better cross-browser support for Ajax.

1. An `XMLHttpRequest` object is stored in a variable called `xhr`.

2. The `XMLHttpRequest` object's `open()` method prepares the request. It has three parameters:
i) Either HTTP `GET` or `POST` to specify how to send the request
ii) The path to the page that will handle the request
iii) Whether or not the request is asynchronous (this is a Boolean)

3. Up to this point, the browser has not yet contacted the server to request the new HTML.

This does not happen until the script gets to the last line that calls the `XMLHttpRequest` object's `send()` method. The `send()` method requires an argument to be passed. If there is no data to send, you can just use `null`.

4. The object's `onload` event will fire when the server responds. It triggers an anonymous function.

5. Inside the function, a conditional statement checks if the `status` property of the object is 200, indicating the server responded successfully. If the example is run locally, there will be no response so you cannot perform this check.

JAVASCRIPT — c08/js/data-html.js

```
① var xhr = new XMLHttpRequest();                    // Create XMLHttpRequest object

④ xhr.onload = function() {                          // When response has loaded
    // The following conditional check will not work locally - only on a server
⑤   if(xhr.status === 200) {                         // If server status was ok
⑥     document.getElementById('content').innerHTML = xhr.responseText; // Update
    }
  };

② xhr.open('GET', 'data/data.html', true);           // Prepare the request
③ xhr.send(null);                                    // Send the request
```

6. Finally, the page is updated: `document.getElementById('content').innerHTML = xhr.responseText;` (A) (B) (C)

A) The element that will contain the new HTML is selected. (Here it is an element whose `id` attribute has a value of `content`.)

B) The `innerHTML` property replaces the content of that element with the new HTML that has been sent from the server.

C) The new HTML is retrieved from the `XMLHttpRequest` object's `responseText` property.

Remember that `innerHTML` should only be used when you know that the server will not return malicious content. All content that has been created by users or third parties should be escaped on the server (see p228).

LOADING XML WITH AJAX

Requesting XML data is very similar to requesting HTML. However, processing the data that is returned is more complicated because the XML must be converted into HTML to be shown on the page.

On the right-hand page, you can see that the code to request an XML file is almost identical to the code to request an HTML file shown on the previous page. What changes is the part *inside* the conditional statement that processes the response (points 1–4 on the right-hand page). The XML must be turned into HTML. The structure of the HTML for each event is shown below.

1. When a server responds with XML, it can be obtained using the `responseXML` property of the `XMLHttpRequest` object. Here, the XML returned is stored in a variable called `response`.

2. This is followed by the declaration of a new variable called `events`, which holds all of the `<event>` elements from the XML document. (You saw the XML file on p375.)

3. The XML file is then processed using the DOM methods you learned about in Chapter 5. First, the `for` loop goes through each of the `<event>` elements, collecting the data stored in their child elements, and placing it into new HTML elements.

Each of those HTML elements is then added into the page.

4. Inside the `for` loop, you will see the `getNodeValue()` function is called several times. Its purpose is to get the contents from each of the XML elements. It takes two parameters:

i) `obj` is an XML fragment.
ii) `tag` is the name of the tag you want to collect the information from.

The function looks for the matching tag within the XML fragment (using the DOM's `getElementsByTagName()` method). It then gets the text from the first matching element within that fragment.

The XML for each event is being transformed into the following HTML structure:

HTML

```
<div class="event">
  <img src="file.png" alt="Location" />
  <p><b>Location</b><br />Event date</p>
</div>
```

JAVASCRIPT

c08/js/data-xml.js

```
var xhr = new XMLHttpRequest();          // Create XMLHttpRequest object

xhr.onload = function() {                // When response has loaded
 // The following conditional check will not work locally - only on a server
 if (xhr.status === 200) {               // If server status was ok

// THIS PART IS DIFFERENT BECAUSE IT IS PROCESSING XML NOT HTML
① var response = xhr.responseXML;                              // Get XML from the server
② var events = response.getElementsByTagName('event'); // Find <event> elements

 for (var i = 0; i < events.length; i++) {                     // Loop through them
  var container, image, location, city, newline;               // Declare variables
  container = document.createElement('div');                   // Create <div> container
  container.className = 'event';                               // Add class attribute

  image = document.createElement('img');                       // Add map image
  image.setAttribute('src', getNodeValue(events[i], 'map'));
  image.appendChild(document.createTextNode(getNodeValue(events[i], 'map')));
  container.appendChild(image);

③ location = document.createElement('p');                      // Add location data
  city = document.createElement('b');
  newline = document.createElement('br');
  city.appendChild(document.createTextNode(getNodeValue(events[i], 'location')));
  location.appendChild(newline);
  location.insertBefore(city, newline);
  location.appendChild(document.createTextNode(getNodeValue(events[i], 'date')));
  container.appendChild(location);

  document.getElementById('content').appendChild(container);
 }
 function getNodeValue(obj, tag) {                             // Gets content from XML
④ return obj.getElementsByTagName(tag)[0].firstChild.nodeValue;
 }

  // THE FINAL PART IS THE SAME AS THE HTML EXAMPLE BUT IT REQUESTS AN XML FILE
  }
 };
 xhr.open('GET', 'data/data.xml', true);                       // Prepare the request
 xhr.send(null);                                               // Send the request
```

LOADING JSON WITH AJAX

The request for JSON data uses the same syntax you saw in the requests for HTML and XML data. When the server responds, the JSON will be converted into HTML.

When JSON data is sent from a server to a web browser, it is transmitted as a string.

When it reaches the browser, your script must then convert the string into a JavaScript object. This is known as **deserializing** an object.

This is done using the `parse()` method of a built-in object called `JSON`. This is a global object, so you can use it without creating an instance of it first.

Once the string has been parsed, your script can access the data in the object and create HTML that can be shown in the page.

The HTML is added to the page using the `innerHTML` property. Therefore, it should only be used when you are confident that it will not contain malicious code (see XSS on p228).

This example will look the same as the last two examples when you view it in a web browser.

The `JSON` object also has a method called `stringify()`, which converts objects into a string using JSON notation so it can be sent from the browser back to a server. This is also known as **serializing** an object.

This method can be used when the user has interacted with the page in a way that has updated the data held in the JavaScript object (e.g., filling in a form), so that it can then update the information stored on the server.

Here you can see the JSON data that is being processed again (it was introduced on p377). Note how it is saved with the `.json` file extension.

c08/data/data.json

JAVASCRIPT

```
{
  "events": [
    { "location": "San Francisco, CA", "date": "May 1", "map": "img/map-ca.png" },
    { "location": "Austin, TX", "date": "May 15", "map": "img/map-tx.png" },
    { "location": "New York, NY", "date": "May 30", "map": "img/map-ny.png"}
  ]
}
```

1. The JSON data from the server is stored in a variable called `responseObject`. It is made available by the `XMLHttpRequest` object's `responseText` property

When it comes from the server, the JSON data is a string, so it is converted into a JavaScript object using the `JSON` object's `parse()` method.

2. The `newContent` variable is created to hold the new HTML data. It is set to an empty string outside the loop so that the code in the loop can add to the string.

3. Loop through the objects that represent each event using a `for` loop. The data in the objects are accessed using dot notation, just like you access other objects.

Inside the loop, the contents of the object are added to the `newContent` variable, along with their corresponding HTML markup.

4. When the loop has finished running through the `event` objects in `responseObject`, the new HTML is added to the page using the `innerHTML` property.

JAVASCRIPT

c08/js/data-json.js

```
var xhr = new XMLHttpRequest();                          // Create XMLHttpRequest object

xhr.onload = function() {                                // When readystate changes
 if(xhr.status === 200) {                                // If server status was ok
① responseObject = JSON.parse(xhr.responseText);

  // BUILD UP STRING WITH NEW CONTENT (could also use DOM manipulation)
② var newContent = '';
  for (var i = 0; i < responseObject.events.length; i++) {//Loop through object
    newContent += '<div class="event">';
    newContent += '<img src="' + responseObject.events[i].map + '" ';
③   newContent += 'alt="' + responseObject.events[i].location + '" />';
    newContent += '<p><b>' + responseObject.events[i].location + '</b><br>';
    newContent += responseObject.events[i].date + '</p>';
    newContent += '</div>';
  }

  // Update the page with the new content
④ document.getElementById('content').innerHTML = newContent;

 }
};

xhr.open('GET', 'data/data.json', true);                 // Prepare the request
xhr.send(null);                                          // Send the request
```

WORKING WITH DATA FROM OTHER SERVERS

Ajax works smoothly with data from your own server but – for security reasons – browsers do not load Ajax responses from other domains (known as cross-domain requests). There are three common workarounds.

A PROXY FILE ON THE WEB SERVER

The first way to load data from a remote server is to create a file on *your* server that collects the data from the remote server (using a server-side language such as ASP.net, PHP, NodeJS, or Ruby). The other pages on your site then request the data from the file on your server (which in turn gets it from the remote server). This is called a **proxy**, because it acts on behalf of the other page.

Because this relies upon creating pages in server-side languages, it is beyond the scope of this book.

JSONP (JSON WITH PADDING)

JSONP (sometimes written JSON-P) involves adding a `<script>` element into the page, which loads the JSON data from another server. This works because there are no restrictions on the source of script in a `<script>` element.

The script contains a call to a function, and the JSON-formatted data is provided as an argument to that function. The function that is called is defined in the page that requests the data, and is used to process and display the data. See next page.

ALTERNATIVES

Many people use jQuery when making requests for remote data, as it simplifies the process and handles backward compatibility for older browsers. As you can see in the next column, support for new approaches is an issue.

CROSS-ORIGIN RESOURCE SHARING

Every time a browser and server communicate, they send information to each other using HTTP headers. **Cross-Origin Resource Sharing** or **CORS** involves adding extra information to the HTTP headers to let the browser and server know that they should be communicating with each other.

CORS is a W3C specification, but is only supported by the most recent browsers and – because it requires setting up of HTTP headers on the server – is beyond the scope of this book.

CORS SUPPORT

Standard support is as follows: Chrome 4, FF 3.5, IE10, Safari 4 Android 2.1, iOS 3.2

IE8+9 used a non-standard `XDomainRequest` object to handle cross-origin requests.

HOW JSONP WORKS

First, the page must include a function to process the JSON data. It then requests the data from the server using a `<script>` element.

BROWSER

The HTML page will use two pieces of JavaScript:

1. A function that will process the JSON data that the server sends. In the example on the next page, the function is called `showEvents()`.

2. A `<script>` element whose `src` attribute will request the JSON data from the remote server.

```
<script>
function showEvents(data) {
  // Code to process data and
  // display it in the page here
}
</script>

<script src="http://example.org/jsonp">
</script>
```

The server returns a file that calls the function that processes the data. The JSON data is provided as an argument to that function.

SERVER

When the server responds, the script contains a call to the named function that will process the data (that function was defined in step 1). This function call is the "padding" in JSONP. The JSON-formatted data is sent as an argument to this function.

So, in this case, the JSON data sits inside the call to the `showEvents()` function.

```
showEvents({
  "events": [
    {
      "location": "San Francisco, CA",
      "date": "May 1",
      "map": "img/map-ca.png"
    }...
  ]
});
```

It is important to note that there is no need to use the JSON object's `parse()` or `stringify()` methods when working with JSONP. Because the data is being sent as a script file (not as a string), it will be treated as an object.

The file on the server is often written so that you can specify the name of the function that will process the data that is returned. The name of the function is usually given in the query string of a URL:
`http://example.org/upcomingEvents.php?callback=showEvents`

USING JSONP

This example looks the same as the JSON example, but the event details come from a remote server. Therefore, the HTML uses two `<script>` elements.

The first `<script>` element loads a JavaScript file that contains the the `showEvents()` function. This will be used to display the deals information.

The second `<script>` element loads the information from a remote server. The name of the function that processes the data is given in the query string.

c08/data-jsonp.html HTML

```
    <script src="js/data-jsonp.js"></script>
    <script src="http://deciphered.com/js/jsonp.js?callback=showEvents"></script>
  </body>
</html>
```

c08/js/data-jsonp.js JAVASCRIPT

```
function showEvents(data) {                          // Callback when JSON loads
  var newContent = '';                               // Variable to hold HTML

  // BUILD UP STRING WITH NEW CONTENT (could also use DOM manipulation)
  for (var i = 0; i < data.events.length; i++) {     // Loop through data
    newContent += '<div class="event">';
    newContent += '<img src="' + data.events[i].map + '"';
    newContent += ' alt="' + data.events[i].location + '" />';
    newContent += '<p><b>' + data.events[i].location + '</b><br>';
    newContent += data.events[i].date + '</p>';
    newContent += '</div>';
  }

  // Update the page with the new content
  document.getElementById('content').innerHTML = newContent; }
```

① **1.** The code in the `for` loop (which is used to process the JSON data and create the HTML) and the line that writes it into the page are the same as the code that processed the JSON data from the same server.

There are three key differences:
i) It is wrapped in a function called `showEvents()`.
ii) The JSON data comes in as an argument of the function call.
iii) The data does not need to be parsed with `JSON.parse()`. In the for loop, it is just referred to by the parameter name `data`.

Instead of using a second `<script>` element in the HTML pages, you can use JavaScript to write that `<script>` element into the page (just like you would add any other element into the page). That would place all the functionality for the external data in the one JavaScript file.

JSONP loads JavaScript, and any JavaScript data may contain malicious code. For this reason, you should load data only from trusted sources.

Since JSONP is loading data from a different server, you might add timer to check if the server has replied within a fixed time (and, if not, show an error message).

You will see more about handling errors in Chapter 10, and there is an example of a timer in Chapter 11 (where you create a content slider).

JAVASCRIPT `http://htmlandcssbook.com/js/jsonp.js`

```
showEvents({
  "events": [
    {
      "location": "San Francisco, CA",
      "date": "May 1",
      "map": "img/map-ca.png"
    },
    {
      "location": "Austin, TX",
      "date": "May 15",
      "map": "img/map-tx.png"
    },
    {
      "location": "New York, NY",
      "date": "May 30",
      "map": "img/map-ny.png"
    }
  ]
});
```

RESULT

The bus stops here.

The file that is returned from the server wraps the JSON-formatted data inside the call to the `showEvents()` function. So the `showEvents()` function is only called when the browser has loaded this remote data.

JQUERY & AJAX: REQUESTS

jQuery provides several methods that handle Ajax requests. Just like other examples in this chapter, the process involves two steps: making a request and handling the response.

Here you can see the six ways jQuery lets you make Ajax requests. The first five are all shortcuts for the `$.ajax()` method, which you meet last.

The `.load()` method operates on a jQuery selection (like most jQuery methods). It loads new HTML content into the selected element(s).

You can see that the other five methods are written differently. They are methods of the global `jQuery` object, which is why they start with $. They only request data from a server; they do not automatically use that data to update the elements of a matched set, which is why the $ symbol is not followed by a selector.

When the server returns data, the script needs to indicate what to do with it.

METHOD / SYNTAX	DESCRIPTION
`.load()`	Loads HTML fragments into an element It is the simplest method for retrieving data
`$.get()`	Loads data using the HTTP `GET` method Used to **request** data from the server
`$.post()`	Loads data using the HTTP `POST` method Used to **send** data that updates data on server
`$.getJSON()`	Loads JSON data using a `GET` request Used for JSON data
`$.getScript()`	Loads and executes JavaScript data using `GET` Used for JavaScript (e.g., JSONP) data
`$.ajax()`	This method is used to perform all requests The above methods all use this under the hood

JQUERY & AJAX: RESPONSES

When using the `.load()` method, the HTML returned from the server is inserted into a jQuery selection. For the other methods, you specify what should be done when the data that is returned using the `jqXHR` object.

JQXHR PROPERTIES	DESCRIPTION
`responseText`	Text-based data returned
`responseXML`	XML data returned
`status`	Status code
`statusText`	Status description (typically used to display information about an error if one occurs)

JQXHR METHODS	DESCRIPTION
`.done()`	Code to run if request was successful
`.fail()`	Code to run if request was unsuccessful
`.always()`	Code to run if request succeeded or failed
`.abort()`	Halt the communication

jQuery has an object called `jqXHR`, which makes it easier to handle the data that is returned from the server. You will see its properties and methods (shown in the tables on the left) used over the next few pages.

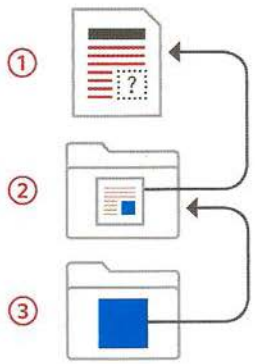
Because jQuery lets you chain methods, you can use the `.done()`, `.fail()`, and `.always()` methods to run different code depending on the outcome of loading the data.

RELATIVE URLS

If the content you load via Ajax contains relative URLs (e.g., images and links) those URLs get treated as if they are relative to the *original* page that was loaded.

If the new HTML is in a different folder from the original page, the relative paths could be broken.

1. This HTML file uses Ajax to load content from a page in the folder shown in step 2.
2. The page in the this folder has an image whose path is a relative link to the second folder:
`<img src="img/box.gif" />`
3. The HTML file cannot find the image as the path is no longer correct – it is not in a child folder.



LOADING HTML INTO A PAGE WITH JQUERY

The `.load()` method is the simplest of the jQuery Ajax methods. It can only be used to load HTML from the server, but when the server responds, the HTML is then loaded into the jQuery Selection for you.

JQUERY SELECTOR

You start by selecting the element that you want the HTML code to appear inside.

URL OF THE PAGE

Then you use the `.load()` method to specify the URL of the HTML page to load.

SELECTOR

You can specify that you want to load only part of the page (rather than the whole page).

```
$('#content').load('jq-ajax3.html #content');
```

1. This creates a jQuery object with the element whose `id` attribute has a value of `content`.

2. This is the URL of the page you want to load the HTML from. There must be a space between the URL and the selector in step 3.

3. This is the fragment of the HTML page to show. Again, it is the section whose `id` attribute has a value of `content`.

Here, links in the top right corner take the user to other pages. If the user has JavaScript enabled, when they click on a link, code inside the `.on()` event method stops it from loading a whole new page. Instead, the `.load()` method will replace the area highlighted in pink (whose `id` attribute has a value of `content`) with the equivalent area from the page that the user just requested. Only the pink area is refreshed – not the whole page.

LOADING CONTENT

When users click on any of the links in the `<nav>` element, one of two things will occur:

If they have JavaScript enabled, a `click` event will trigger an anonymous function that loads new content into the page.

If they do not have JavaScript enabled, they will move from page to page as normal.

Inside the anonymous function, five things happen:

1. `e.preventDefault()` stops the link taking users to a new page.

2. A variable called `url` holds the URL of the page to load. This is collected from the `href` attribute of the link the user clicked on. It indicates which page to load.

3. The `class` attributes on the links are updated to indicate which page is the current page.

4. The element holding the content is removed.

5. The container element is selected and `.load()` fetches new the new content. It is hidden straight away using `.hide()` so that `fadeIn()` can fade it in.

JAVASCRIPT c08/js/jq-load.js

```
$('nav a').on('click', function(e) {                    // User clicks nav link
  e.preventDefault();                                   // Stop loading new link
  var url = this.href;                                  // Get value of href

  $('nav a.current').removeClass('current');            // Clear current indicator
  $(this).addClass('current');                          // New current indicator

  $('#container').remove();                             // Remove old content
  $('#content').load(url + ' #content').hide().fadeIn('slow'); // New content
});
```

HTML c08/jq-load.html

```
<nav>
  <a href="jq-load.html" class="current">Home</a>
  <a href="jq-load2.html">Route</a>
  <a href="jq-load3.html">Toys</a>
</nav>
<section id="content">
  <div id="container">
      <!-- Page content lives here -->
  </div>
</section>
```

The links still work if JavaScript is not enabled. If JavaScript is enabled, jQuery will load content into the `<div>` whose `id` has a value of `content` from the target URL. The rest of the page does not need to be reloaded.

JQUERY'S AJAX SHORTHAND METHODS

jQuery provides four shorthand methods to handle specific types of Ajax requests.

The methods below are all shorthand methods. If you looked at the source code for jQuery, you would see that they all use the `$.ajax()` method.

You will meet each one over the next few pages because they introduce key aspects of the `$.ajax()` method.

These methods do not work on a selection like other jQuery methods, which is why you prefix them with only the $ symbol rather than a jQuery selection. They are usually triggered by an event, such as the page having loaded or the user interacting with the page (e.g., clicking on a link, or submitting a form).

With an Ajax request, you will often want to send data to the server, which will in turn affect what the server sends back to the browser.

As with HTML forms (and the Ajax requests you met earlier in the chapter), you can send the data using HTTP GET or POST.

METHOD / SYNTAX	DESCRIPTION
`$.get(`*`url`*`[, `*`data`*`][, `*`callback`*`][, `*`type`*`])`	HTTP GET request for data
`$.post(`*`url`*`[, `*`data`*`][, `*`callback`*`][, `*`type`*`])`	HTTP POST to update data on the server
`$.getJSON(`*`url`*`[, `*`data`*`][, `*`callback`*`])`	Loads JSON data using a GET request
`$.getScript(`*`url`*`[, `*`callback`*`])`	Loads and executes JavaScript (e.g., JSONP) using a GET request

The parameters in square brackets are optional.

$ shows that this is a method of the `jQuery` object.
`url` specifies where the data is fetched from.
`data` provides any extra information to send to the server.
`callback` indicates that the function should be called when data is returned (can be named or anonymous).
`type` shows the type of data to expect from the server.

Note: The examples in this section only work on a web server (and not on local file systems). Server-side languages and server setup are beyond the scope of this book, but you can try out the examples on our website. PHP files have been included with the download code, but they are for demonstration purposes only.

REQUESTING DATA

Here, users vote for their favorite t-shirt without leaving the page.

1. If users click on a t-shirt an anonymous function is triggered.

2. `e.PreventDefault()` stops the link opening a new page.

3. The user's choice is the value of the `id` attribute on the image. It is stored in a variable called `queryString` in the format of a query string, e.g., `vote=gray`

4. The `$.get()` method is called using three parameters:

i) The page that will handle the request (on the same server).

ii) The data being sent to the server (here it is a query string, but it could be JSON).

iii) The function that handles the result the server sends back; in this case it is an anonymous function.

When the server responds, the anonymous callback function handles the data. In this case, the code in that function selects the element that the held the t-shirts and replaces it with the HTML sent back from the server. This is done using jQuery's `.html()` method.

JAVASCRIPT c08/js/jq-get.js

```
① $('#selector a').on('click', function(e) {
② e.preventDefault();
③   var queryString = 'vote=' + event.target.id;
④   $.get('votes.php', queryString, function(data) {
⑤     $('#selector').html(data);
  });
});
```

HTML (This HTML is created by code inside the JS file.)

```
<div class="third"><a href="vote.php?vote=gray">
  <img src="img/t-gray.png" id="gray" alt="gray" /></a></div>
<div class="third"><a href="vote.php?vote=yellow">
  <img src="img/t-yellow.png" id="yellow" alt="yellow" /></a></div>
<div class="third"><a href="vote.php?vote=green">
  <img src="img/t-green.png" id="green" alt="green" /></a></div>
```

RESULT

The t-shirt links are created in the JavaScript file to ensure they only show if the browser supports JavaScript (the resulting HTML structure is shown above). When the server responds, it does not have to send back HTML; it can return any kind of data that the browser can process and use.

SENDING FORMS USING AJAX

To send data to the server, you are likely to use the `.post()` method. jQuery also provides the `.serialize()` method to collect form data.

SENDING FORM DATA

The HTTP `POST` method is often used when sending form data to a server and it has a corresponding function, the `.post()` method. It takes the same three parameters as the `.get()` method:

i) The name of the file on the (same) server that will process the data from the form
ii) The form data that you are sending
iii) The callback function that will handle the response from the server

On the right-hand page you can see the `$.post()` method used with a method called `.serialize()`, which is very helpful when working with forms. Together they send the form data to the server.

COLLECTING FORM DATA

jQuery's `.serialize()` method:

- Selects all of the information from the form
- Puts it into a string ready to send to the server
- Encodes characters that cannot be used in a query string

Typically it will be used on a selection containing a `<form>` element (although it can be used on individual elements or a subsection of a form).

It will only send *successful* form controls, which means it will not send:

- Controls that have been disabled
- Controls where no option has been selected
- The submit button

SERVER-SIDE

When a server-side page handles a form, you might want the same page to work whether:

- It was a normal request for a web page (in which case you would send the whole page); or
- It was an Ajax request (where you might respond with just a fragment of the page)

On the server, you can check whether a page is being requested by an Ajax call using the `X-Requested-With` header.

If it is set and has a value of `XMLHttpRequest`, you know that the request was an Ajax request.

SUBMITTING FORMS

1. When users submit the form, an anonymous function runs.

2. `e.PreventDefault()` stops the form from submitting.

3. The form data is collected by the `.serialize()` method and stored in the `details` variable.

4. The `$.post()` method is called using all three parameters:

i) The url of the page that the data is being sent to
ii) The data that was just collected from the form
iii) A callback function that will display the results to the user

5. When the server responds, the content of the element whose `id` attribute has a value of `register` is overwritten with new HTML sent from the server.

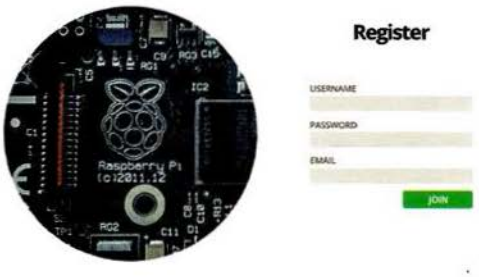
JAVASCRIPT c08/js/jq-post.js

```
① $('#register').on('submit', function(e) {          // When form is submitted
②   e.preventDefault();                              // Prevent it being sent
③   var details = $('#register').serialize();        // Serialize form data
④   $.post('register.php', details, function(data) { // Use $.post() to send it
⑤     $('#register').html(data);                     // Where to display result
    });
});
```

HTML c08/jq-post.html

```
<form id="register" action="register.php" method="post">
 <h2>Register</h2>
 <label for="name">Username</label><input type="text" id="name" name="name" />
 <label for="pwd">Password</label><input type="password" id="pwd" name="pwd" />
 <label for="email">Email</label><input type="email" id="email" name="email" />
 <input type="submit" value="Join" />
</form>
```

RESULT



This example needs to be run on a web server. The server-side page will return a confirmation message (but it does not validate the data submitted nor send a confirmation email).

LOADING JSON & HANDLING AJAX ERRORS

You can load JSON data using the `$.getJSON()` method. There are also methods that help you deal with the response if it fails.

LOADING JSON

If you want to load JSON data, there is a method called `$.getJSON()` which will retrieve JSON from the same server that the page is from. To use JSONP you should use the method called `$.getScript()`.

AJAX AND ERRORS

Occasionally a request for a web page will fail and Ajax requests are no exception. Therefore, jQuery provides two methods that can trigger code depending on whether the request was successful or unsuccessful, along with a third method that will be triggered in both cases (successful or not).

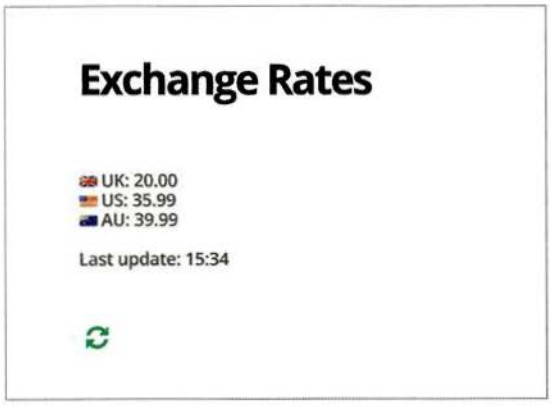
Below is an example that will demonstrate these concepts. It loads fictional exchange rates.

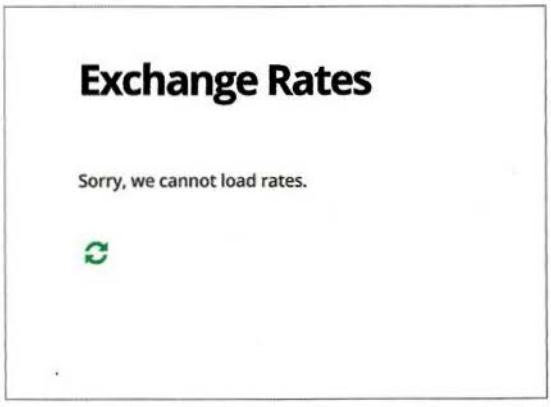
SUCCESS / FAILURE

There are three methods you can chain after `$.get()`, `$.post()`, `$.getJSON()`, and `$.ajax()` to handle success / failure. These methods are:

`.done()` - an event method that fires when the request has successfully completed
`.fail()` - an event method that fires when the request did not complete successfully
`.always()` - an event method that fires when the request has completed (whether it was successful or not)

Older scripts may use the `.success()`, `.error()`, and `.complete()` methods instead of these methods. They do the same thing, but these newer methods have been the preferred option since jQuery 1.8.





JSON & ERRORS

1. In this example, JSON data representing currency exchange rates is loaded into the page by a function called `loadRates()`.

2. On the first line of the script an element is added to the page to hold the exchange rate data.

3. The function is called on the last line of the script.

4. Inside `loadRates()`, the `$.getJSON` method tries to load some JSON data. There are three methods chained after this method. They do not all run.

5. `.done()` only runs if the data is retrieved successfully. It contains an anonymous function that shows exchange rates and the time they were displayed.

6. `.fail()` only runs if the server cannot return the data. Its job is to display an error message to the user.

7. `.always()` will run whether or not the answer was returned. It adds a refresh button to the page, along with an event handler that triggers the `loadRates()` function again.

JAVASCRIPT

c08/js/jq-getJSON.js

```
② $('#exchangerates').append('<div id="rates"></div><div id="reload"></div>');

① function loadRates() {
④   $.getJSON('data/rates.json')
⑤   .done( function(data){                                    // SERVER RETURNS DATA
      var d = new Date();                                     // Create date object
      var hrs = d.getHours();                                 // Get hours
      var mins = d.getMinutes();                              // Get mins
      var msg = '<h2>Exchange Rates</h2>';                    // Start message
      $.each(data, function(key, val) {                       // Add each rate
        msg += '<div class="' + key + '">' + key + ': ' + val + '</div>';
      });
      msg += '<br>Last update: ' + hrs + ':' + mins + '<br>'; // Show update time
      $('#rates').html(msg);                                  // Add rates to page
⑥   }).fail( function() {                                     // THERE IS AN ERROR
      $('aside').append('Sorry, we cannot load rates.');      // Show error message
⑦   }).always( function() {                                  // ALWAYS RUNS
       var reload = '<a id="refresh" href="#">';              // Add refresh link
       reload += '<img src="img/refresh.png" alt="refresh" /></a>';
       $('#reload').html(reload);                             // Add refresh link
       $('#refresh').on('click', function(e) {                // Add click handler
         e.preventDefault();                                  // Stop link
         loadRates();                                         // Call loadRates()
       });
    });
  }

③ loadRates();                                                // Call loadRates()
```

AJAX REQUESTS WITH FINE-GRAINED CONTROL

The `$.ajax()` method gives you greater control over Ajax requests. Behind the scenes, this method is used by all of jQuery's Ajax shorthand methods.

Inside the jQuery file, the `$.ajax()` method is used by the other Ajax helper methods that you have seen so far (which are offered as a simpler way of making Ajax requests).

This method offers greater control over the entire process, with over 30 different settings that you can use to control the Ajax request. You can see a selection of these settings in the table below. These settings are provided using object literal notation (the object is referred to as the settings object).

The example on the right-hand page looks and works like the one that demonstrated the `.load()` method on p390. But it uses the `$.ajax()` method instead.

- The settings can appear in any order, as long as they use valid JavaScript literal notation.
- The settings that take a function can use a named function or an anonymous function written inline.
- `$.ajax()` does not let you load just one part of the page so the jQuery `.find()` method is used to select the required part of the page.

SETTING	DESCRIPTION
`type`	Can take values `GET` or `POST` depending on whether the request is made using HTTP `GET` or `POST`
`url`	The page the request is being sent to
`data`	The data that is being sent to the server with the request
`success`	A function that runs if the Ajax request completes successfully (similar to the `.done()` method)
`error`	A function that runs if there is an error with the Ajax request (similar to the `.fail()` method)
`beforeSend`	A function (anonymous or named) that is run before the Ajax request starts In the example on the right, this is used to trigger a loading icon
`complete`	Runs after success/error events In the example on the right, this removes a loading icon
`timeout`	The number of milliseconds to wait before the event should fail

CONTROLLING AJAX

When the user clicks on a link in the <nav> element, new content is loaded into the page. This is very similar to the example on p390 for the `.load()` method, but that shorthand method only required one line.

1. Here the click event handler triggers the `$.ajax()` method.

This example sets seven settings for the `$.ajax()` method. The first three are properties, the final four are anonymous functions triggered at different points in the Ajax request.

2. This example sets the timeout property to wait two seconds for the Ajax response.

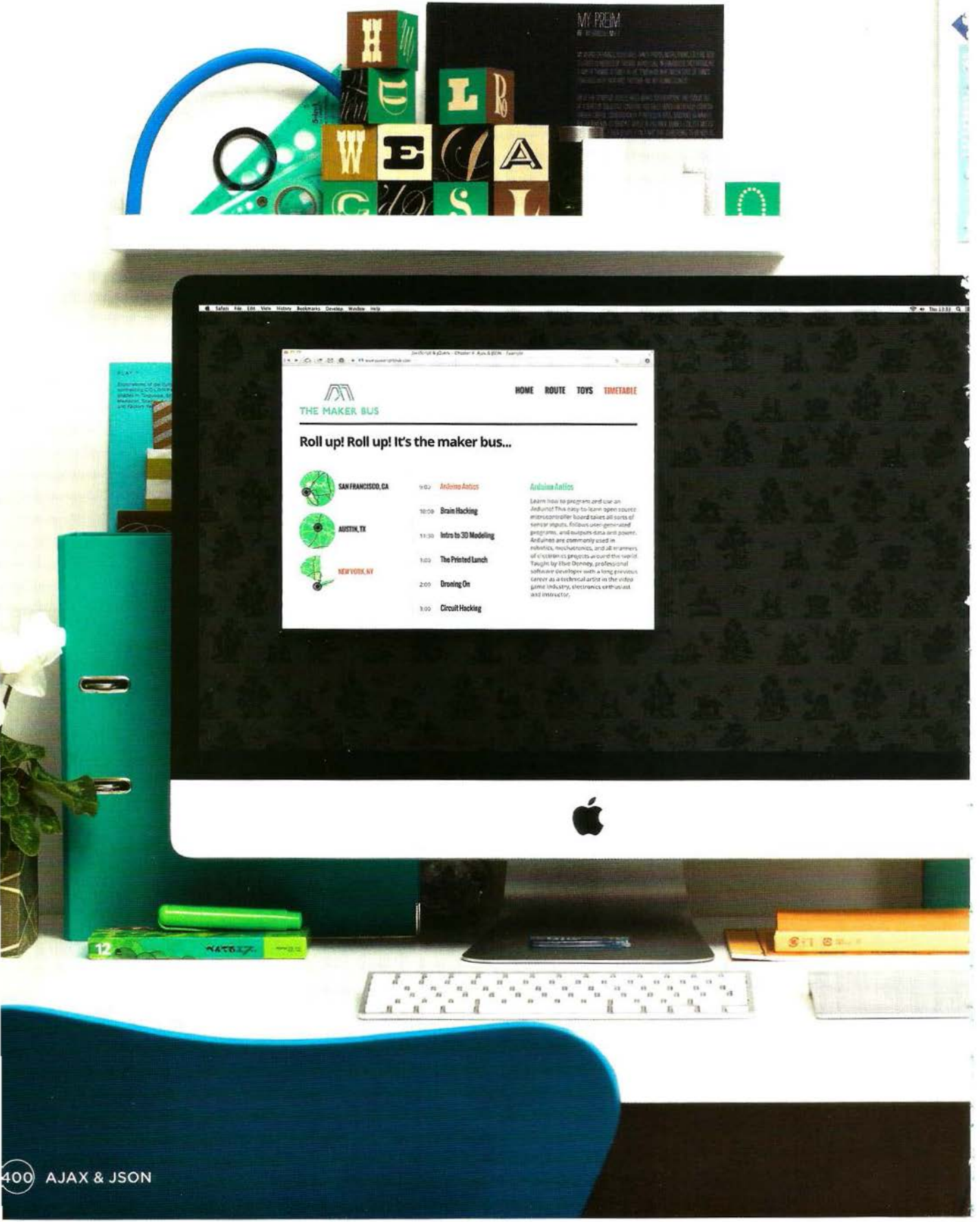
3. The code also adds elements into the page to show that data is loading. You may not see them appear if the request is handled quickly, but you will see them if the page is slower to load.

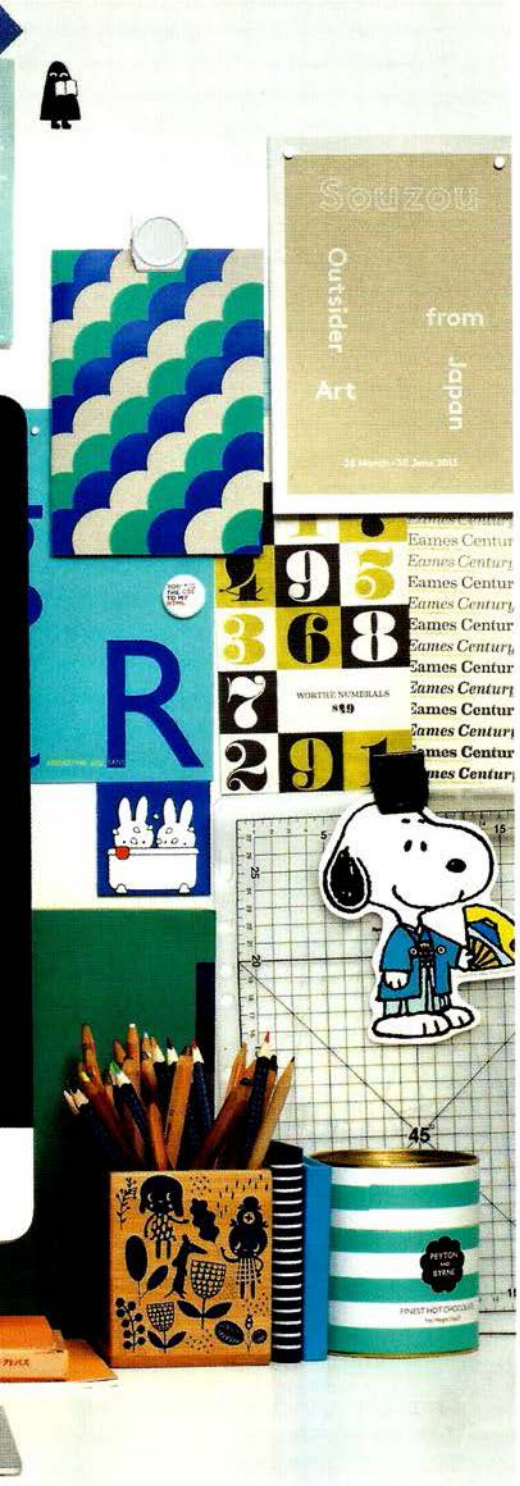
4. If the Ajax request fails, then an error message will be shown to the user.

JAVASCRIPT

c08/js/jq-ajax.js

```
① $('nav a').on('click', function(e) {
    e.preventDefault();
    var url = this.href;                                    // URL to load
    var $content = $('#content');                           // Cache selection

    $('nav a.current').removeClass('current');              // Update links
    $(this).addClass('current');
    $('#container').remove();                               // Remove content

    $.ajax({
      type: "POST",                                         // GET or POST
      url: url,                                             // Path to file
②     timeout: 2000,                                        // Waiting time
      beforeSend: function() {                              // Before Ajax
③       $content.append('<div id="load">Loading</div>');    // Load message
      },
      complete: function() {                                // Once finished
        $('#loading').remove();                             // Clear message
      },
      success: function(data) {                             // Show content
        $content.html( $(data).find('#container') ).hide().fadeIn(400);
      },
      fail: function() {                                    // Show error msg
④       $('#panel').html('<div class="loading">Please try again soon.</div>');
      }
    });

  });
```


HOME ROUTE TOYS TIMETABLE
THE MAKER BUS
Roll up! Roll up! It's the maker bus...
SAN FRANCISCO, CA
AUSTIN, TX
NEW YORK, NY
9:00 Arduino Antics
10:00 Brain Hacking
11:30 Intro to 3D Modeling
1:00 The Printed Lunch
2:00 Droning On
3:00 Circuit Hacking
Arduino Antics
Learn how to program and use an Arduino! This easy-to-learn open source microcontroller board takes all sorts of sensor inputs, follows user-generated programs, and outputs data and power. Arduinos are commonly used in robotics, mechatronics, and all manners of electronics projects around the world. Taught by Elsie Donney, professional software developer with a long previous career as a technical artist in the video game industry, electronics enthusiast and instructor.



EXAMPLE
AJAX & JSON

This example shows information about three events. The data used comes from three different sources.

1) When the page loads, event locations are coded into the HTML. Users click on an event in the left-hand column; it updates the timetable in the middle column.

In the left column, the links have an `id` attribute whose value is a two-letter identifier for the state the event is in:

```
<a id="tx" href="tx.html">... Austin, TX</a>
```

2) The timetables are stored in a JSON object, in an external file collected when the DOM has loaded. When users click on a session in the middle column, its description is shown in the right-hand column.

In the middle column showing timetables, the title of each session is used inside a link that will show the description for the session.

```
<a href="descriptions.html#Circuit-Hacking">
Circuit Hacking</a>
```

3) Descriptions of all sessions are stored in one HTML file. Individual descriptions are selected using jQuery's `.load()` method (and the # selector shown on p390).

In the right column, the session description is taken from an HTML file. Each session is stored in an element whose `id` attribute contains the title of the session (with spaces replaced by dashes).

```
<div id="Intro-to-3D-Modeling">
  <h3>Intro to 3D Modeling</h3>
  <p>Come learn how to create 3D models of ...</p>
</div>
```

Because links are added and removed, event delegation is used.

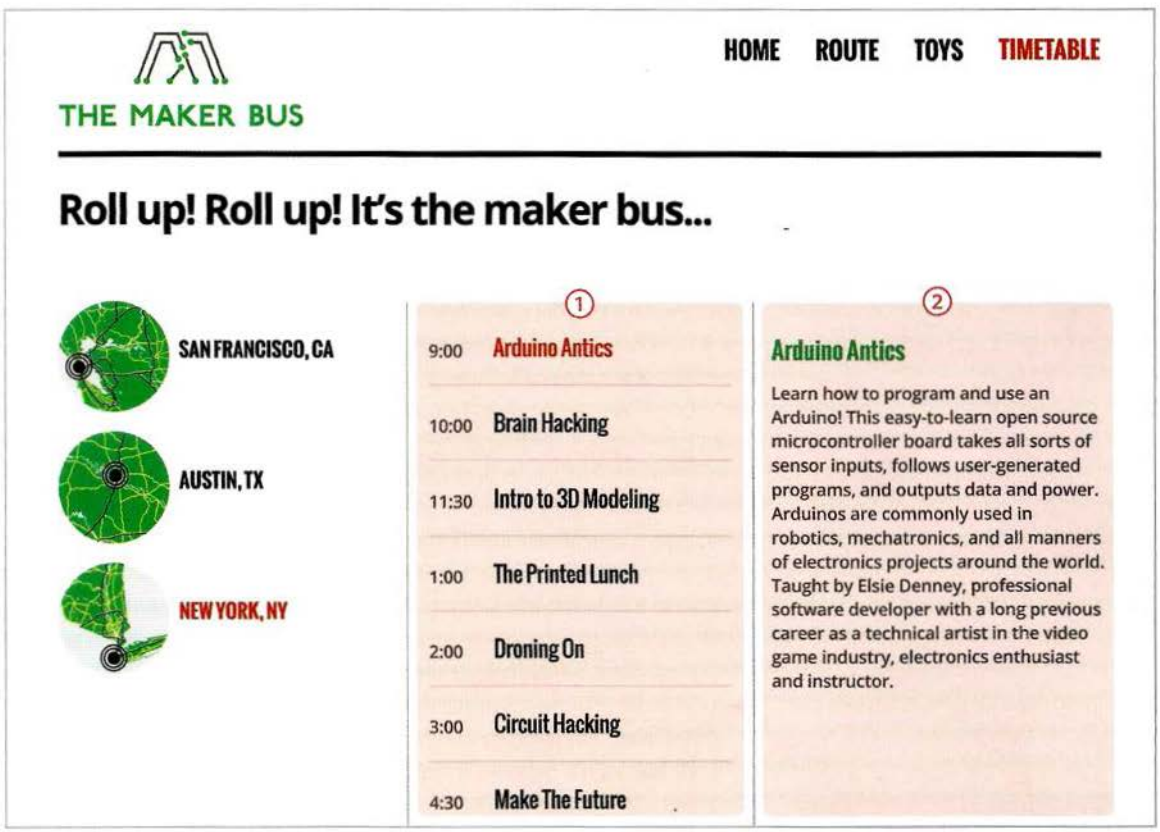
EXAMPLE
AJAX & JSON

This example uses data from three separate sources to demonstrate Ajax techniques.

In the left-hand column you can see three locations for an event. These are written into the HTML for the timetable page. Each one is a link.

1. Clicking on an event loads the session times for that event. They are stored in a file called `example.json`, which is collected when the DOM has loaded.

2. Clicking on a session will load its description. They are stored in `descriptions.html`, which is loaded when a user clicks on a session title.



EXAMPLE
AJAX & JSON

HTML

c08/example.html

```
<body>
  <header>
    <h1>THE MAKER BUS</h1>
    <nav>
      <a href="jq-load.html">HOME</a>
      <a href="jq-load2.html">ROUTE</a>
      <a href="jq-load3.html">TOYS</a>
      <a href="example.html" class="current">TIMETABLE</a>
    </nav>
  </header>

  <section id="content">
    <div id="container">
      <div class="third">
        <div id="event">
          <a id="ca" href="ca.html">
            <img src="img/map-ca.png" alt="SF, CA" />San Francisco, CA</a>
          <a id="tx" href="tx.html">
            <img src="img/map-tx.png" alt="Austin, TX" />Austin, TX</a>
          <a id="ny" href="ny.html">
            <img src="img/map-ny.png" alt="New York, NY" />New York, NY</a>
        </div>
      </div>
      <div class="third">
        <div id="sessions">Select an event from the left</div>
      </div>
      <div class="third">
        <div id="details>Details</div>
      </div>
    </div><!-- #container -->
  </section><!-- #content -->

  <script src="js/jquery-1.11.0.min.js"></script>
  <script src="js/example.js"></script>
</body>
```

Here you can see the HTML page. It has a header, followed by three columns. Two scripts appear before the closing </body> tag.

Left column: list of the events
Middle column: timetable of the sessions
Right column: description of the sessions

EXAMPLE
AJAX & JSON

cNN/data/example.json

JAVASCRIPT

```
{
    "CA": [
        {
            "time": "09.00",
            "title": "Intro to 3D Modeling"
        },
        {
            "time": "10.00",
            "title": "Circuit Hacking"
        },
        {
            "time": "11.30",
            "title": "Arduino Antics"
        }...
```

c08/descriptions.html

HTML

```
<div id="Intro-to-3D-Modeling">
  <h3>Intro to 3D Modeling</h3>
  <p>Come learn how to create 3D models of parts you can then make...</p>
</div>
<div id="Circuit-Hacking">
  <h3>Circuit Hacking</h3>
  <p>Head to the Electro-Tent for a free introductory soldering...</p>
</div>
<div id="Arduino-Antics">
  <h3>Arduino Antics</h3>
  <p>Learn how to program and use an Arduino! This easy-to-learn...</p>
</div>
```

When the script is run, the `loadTimetable()` function loads the timetables for all three events from a file formatted using JSON, stored in `example.json`. The data is cached in a variable called `times`.

Events are identified by a two-letter code for the state. You can see a sample of the JSON-formatted data above and a sample of the HTML that will be created using that data.

EXAMPLE
AJAX & JSON

JAVASCRIPT c08/js/example.js

```
① $(function() {                                          // When the DOM is ready

②   var times;                                            // Declare global variable
    $.ajax({                                              // Setup request
      beforeSend: function(xhr){                          // Before requesting data
③       if (xhr.overrideMimeType) {                       // If supported
          xhr.overrideMimeType("application/json");       // set MIME to prevent errors
        }
      }
    });

    // FUNCTION THAT COLLECTS DATA FROM THE JSON FILE
④   function loadTimetable() {                            // Declare function
      $.getJSON('data/example.json')                      // Try to collect JSON data
⑤     .done( function(data){                              // If successful
        times = data;                                     // Store it in a variable
⑥     }).fail( function() {                               // If a problem: show message
        $('#event').html('Sorry! We could not load the timetable at the moment');
      });
    }

⑦ loadTimetable();                                        // Call the function
```

1. The script that does all the work is in `example.js`. It runs when the DOM has loaded.

2. The `times` variable will be used to store the session timetables for all of the events.

3. Before the browser requests the JSON data, the script checks if the browser supports the `overrideMimeType()` method. This is used to indicate that the response from the server should be treated as JSON data. This method can be used in case the server is accidentally set up to indicate that the data being returned is in any other format.

4. Next you can see a function called `loadTimetable()`, which is used to load the timetable data from a file called `example.json`.

5. If the data loads successfully, the data for the timetables will be stored in a variable called `times`.

6. If it fails to load, an error message will be shown to the users.

7. The `loadTimetable()` function is then called to load the data.

EXAMPLE
AJAX & JSON

c08/js/example.js

JAVASCRIPT

```
    // CLICK ON THE EVENT TO LOAD A TIMETABLE
①   $('#content').on('click', '#event a', function(e) {  // User clicks on place

②     e.preventDefault();                                 // Prevent loading page
③     var loc = this.id.toUpperCase();                    // Get value of id attr

④     var newContent = '';                                // To build up timetable
      for (var i = 0; i < times[loc].length; i++) {       // loop through sessions
⑤       newContent += '<li><span class="time">' + times[loc][i].time + '</span>';
⑥       newContent += '<a href="descriptions.html#';
⑦       newContent += times[loc][i].title.replace(/ /g, '-') + '">';
⑧       newContent += times[loc][i].title + '</a></li>';
      }

⑨     $('#sessions').html('<ul>' + newContent + '</ul>'); // Display time

⑩     $('#event a.current').removeClass('current');       // Update selected link
      $(this).addClass('current');

⑪     $('#details').text('');                             // Clear third column
    });
```

1. A jQuery event helper method waits for users to click on the name of an event. It will load the timetable for that event into the middle column.

2. The `preventDefault()` method prevents the link from opening a page (because it is will show the AJAX data instead).

3. A variable called `loc` is created to hold the name of the event location. It is collected from the `id` attribute of the link that was clicked.

4. The HTML for the timetables will be stored in a variable called `newContent`. It is set to a blank string.

5. Each session is stored inside an `<li>` element, which starts by displaying the time of the session.

6. A link is added to the timetable, which will be used to load the description. The link points to the `descriptions.html` file. It is followed by a # symbol so it links to the correct part of the page.

7. The session title is added after the # symbol. The `.replace()` method replaces spaces in the title with a dash to match the value of the `id` attribute in the `descriptions.html` file for each session.

8. Inside the link you can see the title of the session.

9. The new content is added into the middle column.

10. The `class` attributes on the event links are updated to shows which event is the current event.

11. The third column is emptied if it had content.

EXAMPLE
AJAX & JSON

JAVASCRIPT c08/js/example.js

```
    // CLICK ON A SESSION TO LOAD THE DESCRIPTION
①   $('#content').on('click', '#sessions li a', function(e) { // Click on session
②     e.preventDefault();                                    // Prevent loading
③     var fragment = this.href;                              // Title is in href

④     fragment = fragment.replace('#', ' #');                // Add space after#
⑤     $('#details').load(fragment);                          // To load info

⑥     $('#sessions a.current').removeClass('current');       // Update selected
      $(this).addClass('current');
    });

    // CLICK ON PRIMARY NAVIGATION
    $('nav a').on('click', function(e) {                     // Click on nav
      e.preventDefault();                                    // Prevent loading
      var url = this.href;                                   // Get URL to load

⑦     $('nav a.current').removeClass('current');             // Update nav
      $(this).addClass('current');

      $('#container').remove();                              // Remove old
      $('#content').load(url + ' #container').hide().fadeIn('slow'); // Add new
    });

});
```

1. Another jQuery event helper method is set up to respond when a user clicks on a session in the middle column. It loads a description of the session.

2. `preventDefault()` stops the link opening.

3. A variable called `fragment` is created to hold the link to the session. This is collected from the `href` attribute of the link that was clicked.

4. A space is added before the # symbol so that it is the correct format for the jQuery `load()` method to collect part (not all) of the HTML page, e.g., `description.html #Arduino-Antics`

5. A jQuery selector is used to find the element whose id attribute has a value of `details` in the third column. The `.load()` method is then used to load the session description into that element.

6. The links are updated so that they highlight the appropriate session in the middle column.

7. The main navigation is set up as shown on p391.

SUMMARY

AJAX & JSON

- Ajax refers to a group of technologies that allow you to update just one part of the page (rather than reload a whole page).
- You can incorporate HTML, XML, or JSON data into your pages. (JSON is becoming increasingly popular.)
- To load JSON from a different domain, you can use JSONP but only if the code is from a trusted source.
- jQuery has methods that make it easier to use Ajax.
- `.load()` is the simplest way to load HTML into your pages and allows you to update just a part of the page.
- `.ajax()` is more powerful and more complex. (Several shorthand methods are also offered.)
- It is important to consider how the site will work if the user does not have JavaScript enabled, or if the page is not able to access the data from a server.

9

APIS

User interfaces allow humans to interact with programs. Application Programming Interfaces (APIs) let programs (including scripts) talk to each other.

Browsers, scripts, websites, and other applications frequently open up some of their functionality so that programmers can interact with them. For example:

BROWSERS

The DOM is an API. It allows scripts to access and update the contents of a web page while loaded in the browser. In this chapter you will meet some HTML5 JavaScript APIs that provide access to other browser features.

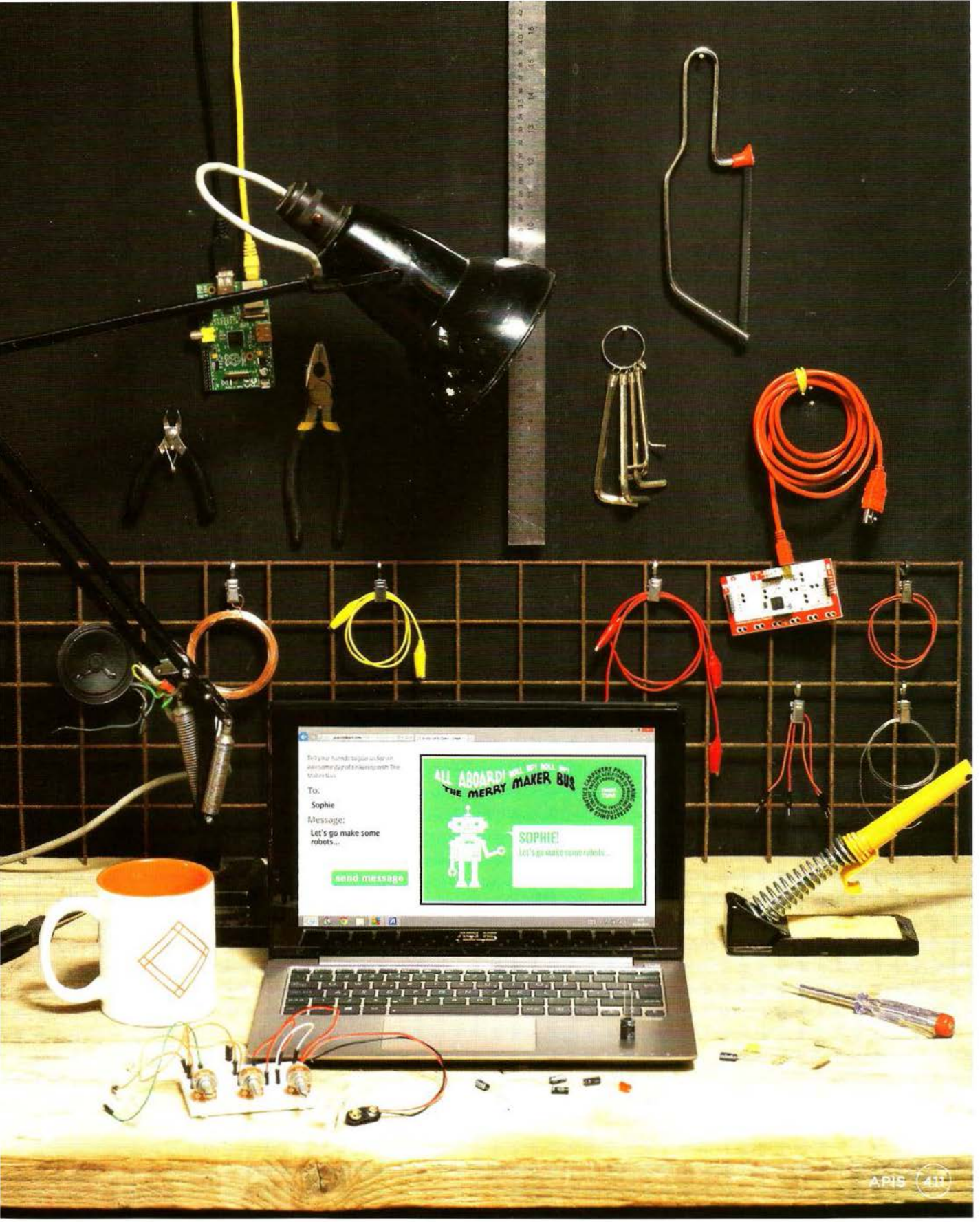
SCRIPTS

jQuery is a JavaScript file with an API. It allows you to select elements, then use its methods to work with those elements. It is just one of many scripts that let you to perform powerful tasks using their code.

PLATFORMS

Sites such as Facebook, Google, and Twitter open up their platforms so that you can access and update data they store (via websites and apps). In this chapter you see how Google lets you to add their maps to your sites.

You do not need to know *how* the other script or program achieves its task; you only need to know what it does, how to ask it to do something, and how to understand its replies. Therefore, this chapter will familiarize you with the form in which APIs are described.


To:
Sophie
Message:
Let's go make some robots...
send message
ALL ABOARD!
THE MERRY MAKER BUS
SOPHIE!

PLAYING NICELY WITH OTHERS

You do not always need to know *how* a script or program works, as long you know how to ask it to do something, and how to process its response. The questions you can ask and the format of the answers form the API.

WHAT THE API CAN DO

If there is a script or program that offers functionality you need, consider using it rather than writing something from scratch.

Because each script, program, or platform has different features, the first thing you need to do is understand what the API allows you to do. For example:

- The DOM and jQuery APIs allow you to access and update a web page that is loaded in the browser and respond to events.
- Facebook, Google+, and Twitter APIs let you to access and update profiles and create status updates on their platforms.

When you know what the API allows you to do, you can decide if it is the right tool for the job.

HOW TO ACCESS IT

Next you need to know how to access the functionality of the API in order to use it.

The DOM's functionality is built into the JavaScript interpreter in the browser.

With jQuery you need to include the jQuery script from your server or a CDN in your pages.

Facebook, Google+, Twitter, and other sites provide various ways to access the functionality of their platforms using APIs.

THE SYNTAX

Finally, you need to learn how to ask the API to do something and the format in which you should expect any replies.

As long as you know how to call a function, create an object, and access the properties and methods of an object, you will be able to use any JavaScript API.

This chapter introduces you to a range of APIs so you gain the confidence to learn more about them and other APIs.

HTML5 JAVASCRIPT APIS

First, we will look at some of the new HTML5 APIs. Along with the markup in the HTML5 specification, a set of APIs define that describe how to interact with features of web browsers.

WHY HTML5 HAS APIS

As technologies evolve, so does the browsing experience. For example, smartphones may have smaller screens and less power than the latest desktop computers; but they include features that are rarely found on desktop machines such as accelerometers and GPS.

The HTML5 specification has not only added new markup, but also includes a new set of JavaScript APIs that standardize how you can make use of these new features in any device that implements them.

WHAT THEY COVER

Each of the HTML5 APIs focuses on one or more objects that browsers implement to deliver specific functionality.

For example, the geolocation API describes a `geolocation` object that lets you ask users for their location and two objects that handle the browsers response.

There are also APIs that offer improvements over existing functionality. For example, the web storage API lets you store information within the browser without relying on cookies.

WHAT YOU'LL LEARN

There is not space for an exhaustive reference of each of the HTML5 APIs (there have been whole books dedicated to these new HTML5 features). But you will meet three of the APIs and see examples of how to work with them.

This should get you used to using the HTML5 APIs so that you can then go on and learn more about them as you need them. You will also learn how you can test to see whether or not a browser supports the functionality in any of the APIs.

API	DESCRIPTION	
`geolocation`	How to tell where the user is located	p418
`localStorage`	Store information in the browser (even when user closes tab/window)	p420
`sessionStorage`	Store information in the browser while a tab/window is open	
`history`	How to access items from the browser's history	p424

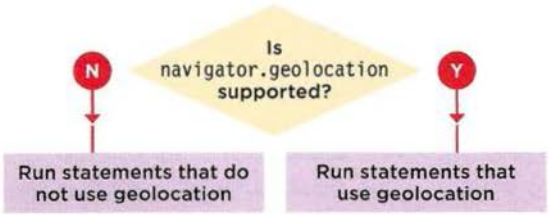
FEATURE DETECTION

When you write code that uses the HTML5 APIs (or any other new feature in a web browser), you may need to check if the browser supports that feature before your code tries to use it.

The HTML5 APIs describe objects that browsers use to implement new functionality. For example, you are about to meet an object called the `geolocation` object that is used to determine a user's location. However, this object is only supported in modern browsers, so you need to check whether a browser supports this it before trying to use the object.

It is possible to check whether a browser supports an object using a conditional statement.

If the browser supports the object, then the condition will return a truthy value and the first set of statements are run. If it is not implemented, the second set of statements is run.




```
if (navigator.geolocation) {
  // Returns truthy so it is supported
  // Run statements in this code block
} else {
  // Not supported / turned off
  // Or user rejected request
}
```

You may not be surprised to hear that there are some cross-browser issues with feature detection.

For example, in the case of the code above, there was a bug in IE9 which could result in a memory leak when you check for the `geolocation` object. This could slow down your pages.

Luckily, there is a library called Modernizr, which takes away the hassles of cross-browser issues (like jQuery for feature detection). It is a better way to check if the browser supports recent features. The script is regularly updated and refined to deal with cross-browser issues as they are discovered, so they are less likely to affect you.

MODERNIZR

Modernizr is a script you can use in your pages to tell whether the browser supports features of HTML, CSS, and JavaScript. It will be used in the coming HTML5 API examples.

HOW TO GET MODERNIZR

First, you need to download the script from the Modernizr.com website, where you will see:

- A development version of the script.
  It is uncompressed and features every check that the script is capable of performing.
- A tool (see screenshot below) that lets you select which features you want to test for.
  You can then download a custom version of the script that only contains the checks you *need*.
  On a live site, you should not test for features that you do not use as it would slow your site down.

In our examples, Modernizr is used near the end of the page just before the script that uses it. But you may see Modernizr included in the <head> of an HTML page (if the content of the page is uses features that you are testing for).

HOW MODERNIZR WORKS

When you include the Modernizr script in your page, it adds an object called `Modernizr`, which tests whether the browser supports the features that you specified that it should test for. Each feature you want it to test becomes a property of the `Modernizr` object. Their values are a Boolean (`true` or `false`) that tell you if a feature is supported.

You can use Modernizr as a condition like this:
If `Modernizr`'s `geolocation` property returns `true` run the code in the curly braces:

```
if (Modernizr.geolocation) {
  // Geolocation is supported
}
```

MODERNIZR PROPERTIES

In the screenshot on the left, you can see some of the features that Modernizr can check for. To see a full list of `Modernizr`'s properties, visit: modernizr.github.io/Modernizr/test/index.html

GEOLOCATION API: FINDING USERS' LOCATIONS

An increasing number of sites offer extra functionality to users who disclose their location. The users' location can be requested using the geolocation API.

WHAT THE GEOLOCATION API DOES

Browsers that implement the geolocation API let users share their location with websites. The location data is provided in the form of longitude and latitude points. There are several ways for the browser to determine its location, including using data from its IP address, wireless network connection, cell towers, or GPS hardware.

In some devices, the geolocation API can give you more data along with longitude and latitude. But, we focus on these features because they have the most support. Having seen how to use them, if you need to work with the other features, you will be able to.

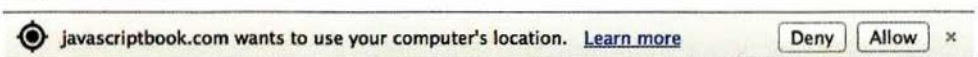
HOW TO ACCESS GEOLOCATION

The geolocation API is available by default in any browser that supports it (just like the DOM is). It was first supported in IE9, Firefox 3.5, Safari 5, Chrome 5, Opera 10.6, iOS3, and Android 2.

Browsers that support geolocation allow users to turn the feature on and off. If it is on, the browser will ask users if they want to share data for each individual web site that requests that information.

The way in which the browser asks the user if they will share location data differs from one browser to the next and one device to the next.

CHROME ON MAC

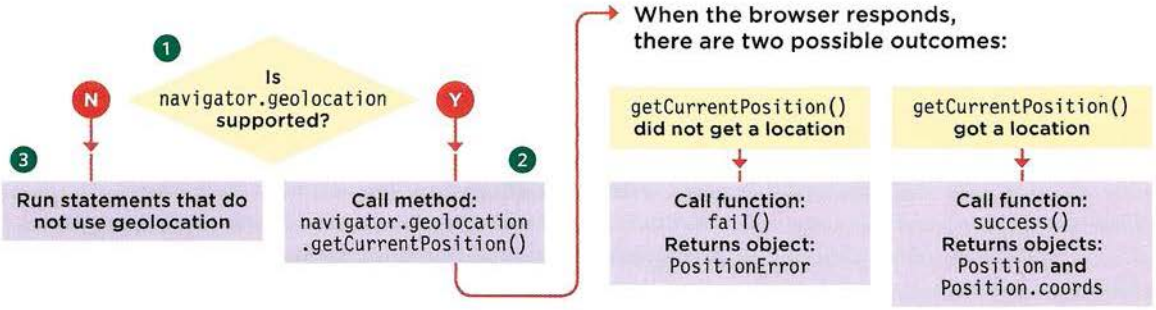


IOS ON IPHONE

FIREFOX ON PC

REQUESTING A USER'S LOCATION

PROCESSING THE RESPONSE



The geolocation API relies on an object called `geolocation`. If you want to try and make use of the user's location, first you need to check if the browser supports this object. This example will use the Modernizr script is used to perform this check.

1. A conditional statement is used to check whether the browser supports geolocation.

2. If geolocation is supported, the browser returns a truthy value and the first set of statements run. They request the user's location using the `geolocation` object's `getCurrentPosition()` method.

3. If geolocation is not supported, then a second set of statements is run.

```
if (Modernizr.geolocation) {
  // Returns truthy so it is supported
  // Run statements in this code block
} else {
  // Not supported / turned off
  // Or user rejected request
}
```

Once you call the `getCurrentPosition()` method, the code continues onto the next line because it is an asynchronous request (like the Ajax calls in the last chapter). The request is asynchronous because the browser will take a while to determine the user's location (and you do not want the rest of the page to stop loading while the browser works out where the user is). Therefore, the method has two parameters:
`getCurrentPosition(`*`success`*`, `*`fail`*`)`

`success` is the name of a function to call if the longitude and latitude are successfully returned. This method will automatically be passed an object called `position`, which holds the user's location.

`fail` is the name of a function called if the details cannot be obtained. This method will automatically be passed an object called `PositionError` containing details about the error.

So in all, there are three new objects you need to use in order to work with the geolocation API: `geolocation`, `position`, and `PositionError`. Their syntax is shown on the next page.

THE GEOLOCATION API

There are three objects involved in adding geolocation to your web page. The tables demonstrate how API documentation typically describes the objects, properties, and the methods you can use.

`geolocation` OBJECT

The `geolocation` object is used to request location data. It is a child of the `navigator` object.

METHOD	RETURNS
`getCurrentPosition(`*`success`*`, `*`fail`*`)`	Requests the position of the user and, if the user permits, returns the user's latitude / longitude plus other location information *`success`* is the name of a function to call if coordinates are retrieved *`fail`* is the name of a function to call if coordinates are not returned

`Position` OBJECT

If a user's location is found, a `Position` object is sent to the callback function. It has a child object called `coords` whose properties hold the user's location. If a device supports `geolocation`, it must provide a minimum amount of data (see the required column); other properties are optional (they may depend on the device's capabilities).

PROPERTY	RETURNS	REQUIRED
`Position.coords.latitude`	Latitude in decimal degrees	Yes
`Position.coords.longitude`	Longitude in decimal degrees	Yes
`Position.coords.accuracy`	Accuracy of latitude and longitude in meters	Yes
`Position.coords.altitude`	Meters above sea level	Yes (value can be `null`)
`Position.coords.altitudeAccuracy`	Accuracy of altitude in meters	Yes (value can be `null`)
`Position.coords.heading`	Degrees clockwise from north	No (up to device)
`Position.coords.speed`	Speed traveling in meters per second	No (up to device)
`Position.coords.timestamp`	Time since created (formatted as `Date` object)	No (up to device)

`PositionError` OBJECT

If location is not determined, the callback function is passed the `PositionError` object.

PROPERTY	RETURNS	REQUIRED
`PositionError.code`	An error number with the following values: 1 Permission denied 2 Unavailable 3 Timeout	Yes
`PositionError.message`	A message (not intended for the end user)	Yes

WORKING WITH LOCATION

1. In this example, Modernizr checks if geolocation is supported by the browser and enabled by the user.
2. When `getCurrentPosition()` is called, the user will be asked for permission to share their location.
3. If the location is gained, the user's latitude and longitude are written into the page.
4. If it is not supported, then the user will see a message that shows their location could not be found.
5. If the location is not gained (for any reason), again the message will say that a location cannot be found. The error code is logged to the browser console.

JAVASCRIPT c09/js/geolocation.js

```
var elMap = document.getElementById('loc');                    // HTML element
var msg = 'Sorry, we were unable to get your location.';       // No location msg

① if (Modernizr.geolocation) {                                 // Is geo supported
②   navigator.geolocation.getCurrentPosition(success, fail);   // Ask for location
    elMap.textContent = 'Checking location...';                // Say checking...
  } else {                                                     // Not supported
④   elMap.textContent = msg;                                   // Add manual entry
  }

function success(position) {                                   // Got location
  msg = '<h3>Longitude:<br>';                                  // Create message
  msg += position.coords.latitude + '</h3>';                   // Add latitude
③ msg += '<h3>Latitude:<br>';                                  // Create message
  msg += position.coords.longitude + '</h3>';                  // Add longitude
  elMap.innerHTML = msg;                                       // Show location
}

function fail(msg) {                                           // Not got location
⑤   elMap.textContent = msg;                                   // Show text input
  console.log(msg.code);                                       // Log the error
}
```

HTML c09/geolocation.html

```
<script src="js/geolocation.js"></script>
```

If you are unable to see a result on a desktop browser, try the example on a smart phone.
You can try all examples directly from the website for the book, `http://www.javascriptbook.com/`.
To support older browsers, search for a script called `geoPosition.js`

WEB STORAGE API: STORING DATA IN BROWSERS

Web storage (or HTML5 storage) lets you store data in the browser. There are two different types of storage: **local** and **session** storage.

HOW TO ACCESS THE STORAGE API

Before HTML5, cookies were the main mechanism for storing information in the browser. But cookies have several limitations, most notably they are:

- Not able to hold much data.
- Sent to the server every time you request a page from that domain.
- Not considered secure.

Therefore, HTML5 introduced a **storage object**. There are two different flavors of the storage object, `localStorage` and `sessionStorage`. Both use the same methods and properties. The differences are how long the data is stored for and whether all tabs can access the data that is being stored.

STORAGE	LOCAL	SESSION
Is the data stored when you close a window/tab?	✔	✘
Can all open windows/tabs access the data?	✔	✘

Commonly, browsers store 5MB of data per domain in a storage object. If a site tries to store more than 5mb of data, the browser will usually ask the user whether they want to allow this site to store more information (never rely on users agreeing to give a site more space).

The data is stored as properties of the storage objects (using in key/value pairs). The value in the pair is always a string. To protect the information that a website stores in these storage objects, browsers employ a **same origin policy**, which means data can only be accessed by other pages in the same domain.

```
http://www.google.com:80
└─①─┘② └──③──┘④
```

These four parts of the URL must match:

1. **Protocol:** The protocol must be a match. If data was stored by a page that starts `http`, the storage object cannot be accessed via `https`.
2. **Subdomain:** The subdomain name must match. For example, `maps.google.com` cannot access data stored by `www.google.com`.
3. **Domain:** The domain name must match. For example, `google.com` cannot access local storage from `facebook.com`.
4. **Port:** The port number must match. Web servers can have many ports. Usually a port number is not specified in a URL, and the site uses port 80 for web pages, but the port number *can* be changed.

The storage objects are just one of the new HTML5 APIs for storing data. Others include access to the file system (through the FileSystem API) and client side databases such as the Web SQL database.

HOW TO ACCESS THE STORAGE API

Both of these objects are implemented on the `window` object, so you do not need to prefix the method names with any other object name.

To save an item into the storage object, you use the `setItem()` method, which takes two parameters: the name of the key and the value associated with it.

To retrieve a value from the storage object you use the `getItem()` method, passing it the key.

```
// Store information
localStorage.setItem('age', '12');
localStorage.setItem('color', 'blue');
// Access information and store in variable
var age   = localStorage.getItem('age');
var color = localStorage.getItem('color');
// Number of items stored
var items = localStorage.length;
```

You can also set and retrieve keys and values of the storage objects as you might with other objects using dot notation.

The storage objects are commonly used to store JSON-formatted data. The `JSON` object's:

- `parse()` method is used to turn the JSON-formatted data into a JavaScript object
- `stringify()` method is used to transform objects into JSON-formatted strings

```
// Store information (object notation)
localStorage.age   = 12;
localStorage.color = 'blue';
// Access information (object notation)
var age = localStorage.age;
var color = localStorage.color;
// Number of items stored
var items = localStorage.length;
```

Data for the storage objects is stored and accessed in a synchronous manner: all other processing stops while the script accesses or saves the data. Therefore, if a lot of data is regularly accessed or stored, the site can appear slower to use.

Below, you can see a table that shows the methods and property of the storage objects. This table is very similar to the one you saw for the geolocation API and is indicative of the types of tables you see in documentation for APIs.

METHOD	DESCRIPTION
`setItem(key, value)`	Creates a new key/value pair
`getItem(key)`	Gets the value for the specified key
`removeItem(key)`	Removes the key/value pair for the specified key
`clear()`	Clears all information from that storage object

PROPERTY	DESCRIPTION
`length`	Number of keys

LOCAL STORAGE

The examples on this page and the right-hand page store what the user enters into text boxes, but both examples store it for different lengths of time.

1. A conditional statement is used to check if the browser supports the relevant storage API.
2. References to the inputs for the username and answer are stored in variables.
3. The script checks to see if the storage object has a value for either of these elements using the `getItem()` method. If so, it is written into the appropriate input by updating its `value` property.
4. Each time an `input` event fires on one of the inputs, the form will save the data to the `localStorage` or `sessionStorage` object. It will automatically be shown if you refresh the page.

c09/js/local-storage.js — JAVASCRIPT

```
① if (window.localstorage) {

②   var txtUsername = document.getElementById('username');// Get form elements
    var txtAnswer = document.getElementById('answer');

③   txtUsername.value = localStorage.getItem('username'); // Elements populated
    txtAnswer.value = localStorage.getItem('answer');     // by localStorage data

④   txtUsername.addEventListener('input', function () {   // Data saved
      localStorage.setItem('username', txtUsername.value);
    }, false);

    txtAnswer.addEventListener('input', function () {     // Data saved
      localStorage.setItem('answer', txtAnswer.value);
    }, false);
  }
```

c09/local-storage.html (The only difference in session-storage.html is the link to the script.) — HTML

```
<div class="two-thirds">
  <form id="application" action="apply.php">
    <label for="username">Name</label>
    <input type="text" id="username" name="username" /><br>
    <label for="answer">Answer</label>
    <textarea id="answer" name="answer"></textarea>
    <input type="submit" />
  </form>
</div>
<script src="js/local-storage.js"></script>
```

SESSION STORAGE

`sessionStorage` is more suited to information that:

- Changes frequently (each time the user visits the site - such as whether they are logged in or location data).
- Is personal and should not be viewed by other users of the device.

`localStorage` is best suited to information that:

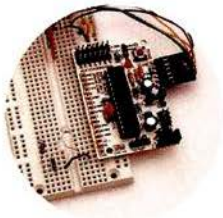
- Only changes at set intervals (such as timetables / price lists), which can be helpful to store offline.
- The user might want to come back and use again (such as saving preferences / settings).

JAVASCRIPT c09/js/session-storage.js

```
① if (window.sessionstorage) {

②   var txtUsername = document.getElementById('username');  // Get form elements
    var txtAnswer = document.getElementById('answer');

③   txtUsername.value = sessionStorage.getItem('username'); // Elements populated
    txtAnswer.value = sessionStorage.getItem('answer');     // by sessionStorage

④   txtUsername.addEventListener('input', function () {     // Save data
      sessionStorage.setItem('username', txtUsername.value);
    }, false);

    txtAnswer.addEventListener('input', function () {       // Save data
      sessionStorage.setItem('answer', txtAnswer.value);
    }, false);
  }
```

RESULT

What would you like to make?



Name

Answer

Submit

HISTORY API & PUSHSTATE

If you move from one page to another, the browser's history remembers which pages you visited. But Ajax applications do not load new pages, so they can use the `history` API to update the location bar and history.

WHAT THE HISTORY API DOES

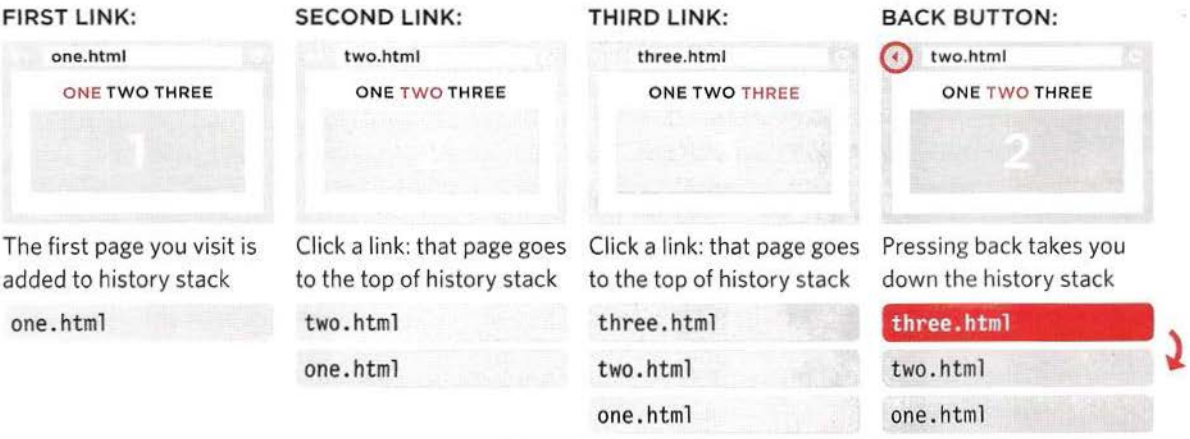
Each tab or window in the browser keeps its own history of pages you have viewed. When you visit a new page in that tab or window, the URL is added to the list of pages you have visited in the history.

Because of this, you can use the back and forward buttons in a browser to move between pages you have visited in that tab or window. However, on sites that use Ajax to load information, the URL is not automatically updated (and the back button might not show the last thing that the user was viewing).

HTML5's history API can help fix this problem. It lets you interact with the browser's `history` object:

- You can update the browser history stack using the `pushState()` and `replaceState()` methods.
- Extra information can be stored with each item.

As you will see, information can be added to the `history` object when an Ajax request is made, and the user can be shown the right content when they press back or forward buttons.



Browsing pages:
As you browse, the URL in your web browser's address bar updates. The page is also added to the top of something called the **history stack**.

Pressing back: takes you back down the stack
Pressing forward: takes you up the stack (where possible)
New page: if you request a new page, it will replace anything above the current page in the stack

State refers to the condition that something is in at a particular time. The browser history is like a pile (or stack) of states, one on top of the other. The three methods on this page allow you to manipulate the state in browsers.

ADDING INFORMATION TO THE HISTORY OBJECT

`pushState()` adds an entry to the `history` object. `replaceState()` updates the current entry. Both take the same three parameters (below), each of which updates the `history` object.

Because the `history` object is a child of the `window` object, you can use its name directly in the script; you can write `history.pushState()`, you do not need to write `window.history.pushState()`.

```
history.pushState(state, title, url);
                  ①      ②      ③
```

1. The `history` object can store information with each item in the history. This is provided in the `state` parameter and can be retrieved when you go back to that page.

2. Currently unused by most browsers, the `title` parameter is intended to change the title of the page. (You can specify a string for this value, ready for when browsers support it.)

3. The URL that you want the browser to show for this page. It must be on the same origin as the current URL and it should show the correct content if the user goes back to that URL.

GETTING INFORMATION FROM THE HISTORY OBJECT

Adding content to the browser history is only part of the solution; the other half is loading the right content when the user presses the back or forward buttons. To help show the right content, the `onpopstate` event fires whenever the user requests a new page.

This `onpopstate` event is used to trigger a function that will load the appropriate content into the page. There are two ways to determine what content should be loaded into the page:

- The `location` object (which represents the browser's location bar)
- The `state` information in the `history` object

The `location` object:
If the user presses back or forward, the address bar will update itself, so you can get the URL for the page that should be loaded using `location.pathname` (the `location` object is a child of the `window` object and its `pathname` property is the current URL). This works well when you are updating an entire page.

The `state`:
Because the first parameter of the `pushState()` method stores data with the `history` object for that page, you can use it to store JSON-formatted data. That data can then be loaded directly into the page. (This is used when the new content loads data rather than a traditional web page.)

THE HISTORY OBJECT

The HTML5 history API describes the functionality of the `history` object in modern web browsers. It lets you access and update the browser history (but only for pages the user visited on your site).

Even if the visitor is not taken to a new page in the browser window (for example, when only a part of the page is updated using Ajax), you can modify the `history` object to ensure that the back and forward buttons work as the user would expect them to on non-Ajax pages.

Again, the table below is indicative of the kind you might see in API documentation. As you become comfortable using the methods, properties, and events of an object you will find it easier to work with all kinds of APIs.

`history` OBJECT

METHOD	DESCRIPTION
`history.back()`	Takes you back in the history, like the browser's back button
`history.forward()`	Takes you forward in the history, like the browser's forward button
`history.go()`	Takes you to a specific page in the history. It is an index number, starting at 0. `.go(1)` is like clicking the forward button and `.go(-1)` is like clicking back
`history.pushState()`	Adds an item to the history stack (Clicking on a relative link in a page usually triggers a `hashchange` event, rather than `load`, but no event fires if you use `pushState()` and the url contains a hash)
`history.replaceState()`	Does the same as `pushState()` except it modifies the current history entry

PROPERTY	DESCRIPTION
`length`	Tells you how many items are in the `history` object

EVENT	DESCRIPTION
`window.onpopstate`	Used to handle the user moving backwards or forwards

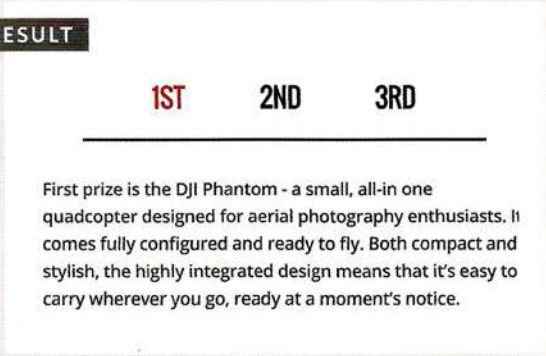
WORKING WITH HISTORY

1. The `loadContent()` function uses jQuery's `.load()` method (see p390) to load content into the page.
2. If a link is clicked on, an anonymous function runs.
3. The page to load is held in a variable called `href`.
4. The current links are updated.
5. The `loadContent()` function is called (see step 1).
6. The `pushState()` method of the `history` object updates the history stack.

JAVASCRIPT c09/js/history.js

```
$(function() {                                    // DOM has loaded
  function loadContent(url){                      // Load new content into page
    $('#content').load(url + ' #container').hide().fadeIn('slow');
  }

  $('nav a').on('click', function(e) {            // Click handler
    e.preventDefault();                           // Stop link loading new page
    var href = this.href;                         // Get href attribute of link
    var $this = $(this);                          // Store link in jQuery object
    $('a').removeClass('current');                // Remove current from links
    $this.addClass('current');                    // Update current link
    loadContent(href);                            // Call function: loads content
    history.pushState('', $this.text, href);      // Update history
  });

  window.onpopstate = function() {                // Handle back/forward buttons
    var path = location.pathname;                 // Get the file path
    loadContent(path);                            // Call function to load page
    var page = path.substring(location.pathname.lastIndexOf("/") + 1);
    $('a').removeClass('current');                // Remove current from links
    $('[href="' + page + '"]').addClass('current'); // Update current link
  };
});
```

RESULT



7. When the user clicks backwards or forwards, the `onpopstate` event fires. This is used to trigger an anonymous function.
8. The browser's location bar will display the corresponding page from the history stack, so `location.pathname` is used to obtain the path for the page that needs to be loaded.
9. The `loadContent()` function (in step 1) is called again, to retrieve the specified page.
10. The file name is retrieved so that the current link can be updated.

SCRIPTS WITH APIS

There are hundreds of scripts available for free on the web. Many have an API you need to use to get them to work for you.

SCRIPT APIS

Lots of developers share their scripts through a range of websites. Some are relatively simple scripts with a single purpose (such as sliders, lightboxes, and table sorters). Others are far more complicated and can be used for a range of purposes (such as jQuery).

In this section, you will meet two different types of scripts whose code you can make use of when you have learned their API:

- A set of jQuery plugins known as jQuery UI.
- A script that makes it easier to create web apps called AngularJS.

JQUERY PLUGINS

Many developers have written code that adds extra functionality to jQuery. These scripts add methods to extend the jQuery object, which are known as **jQuery plugins**.

When you use these plugins, first you include the jQuery script, followed by the plugin script. Then, when you select elements (as you do with standard in jQuery methods), the plugin allows you to apply new methods that it has defined to that selection, offering new functionality that was not in the original jQuery script.

ANGULAR

Angular.js is another JavaScript library, but it is *very* different from jQuery. Its purpose is to make it easier to develop web applications.

One of the most striking things is that it allows you to access and update the contents of a page without writing code to handle events, select elements, or update the content of an element. We only have space to provide a very basic introduction to Angular in this chapter, but it does help demonstrate the variety of scripts available.

THIRD-PARTY SCRIPTS

Before writing your own script it can pay to check if someone else has already done the hard work for you (there is no point reinventing the wheel).

It is always a good idea to check:

- Whether it has been updated fairly recently
- That the JavaScript is separate from the HTML
- Reviews of the script if they are available

This helps to ensure that the script uses modern practices and is still being updated.
It is also worth noting that the instructions for using a script are not always called an API.

JQUERY UI

The jQuery foundation maintain its own set of jQuery plugins called jQuery UI. They help create user interfaces.

WHAT JQUERY UI DOES

jQuery UI is a suite of jQuery plugins that extends jQuery with a set of methods to create:

- Widgets (such as accordions and tabs)
- Effects (that make elements appear and disappear)
- Interactions (such as drag and drop functionality)

jQuery UI not only provides JavaScript you can use, but it also comes with a set of themes that help control how the plugins look on the page.

If you want fine-grained control over how the jQuery plugins look in the browser, you can also use the **theme roller**, which gives you more precise control over the appearance of the elements.

HOW TO ACCESS IT

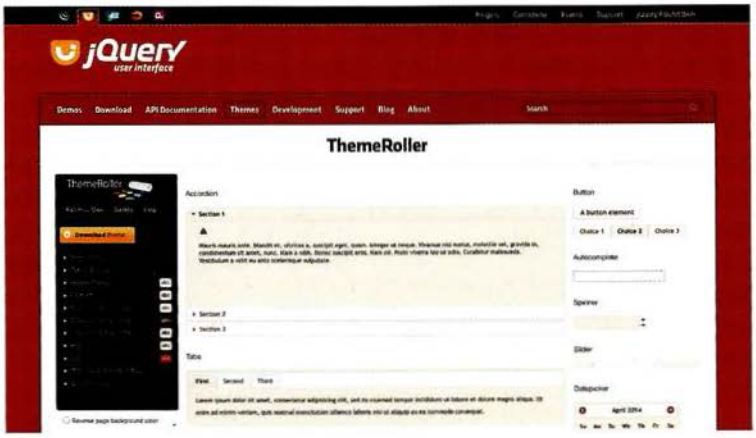
To use jQuery UI, first you must include jQuery in your page; then you must include the jQuery UI script (after the jQuery file).

Versions of jQuery UI are available on the same CDNs as the main jQuery file. But, if you only need part of the jQuery UI functionality, you can just download the relevant parts from the `jqueryui.com` website. This creates a smaller JavaScript file, which in turn makes the script faster to download.

SYNTAX

Once you have included the jQuery and jQuery UI scripts in the page, the syntax is very similar to using other jQuery methods. You create a jQuery selection and then call a method that will be defined in the plugin.

As you will see, the jQuery UI documentation not only has to explain the JavaScript methods and properties it uses, but also how to structure your HTML if you want to use many of its widgets and interactions.



JQUERY UI ACCORDION

Creating an accordion with jQuery UI is very simple. You only need to know:

- How to structure your HTML
- What element(s) should be used in the jQuery selector
- The jQuery UI method to call

1. In this example, the HTML for an accordion is contained within a `<div>` element (its `id` attribute has a value of `prizes`, which will be used in the script). Each panel of the accordion has:

2. An `<h3>` element for the clickable heading
3. A `<div>` element for the content of that panel

4. Before the closing `</body>` tag the jQuery and jQuery UI scripts are both included in the page.

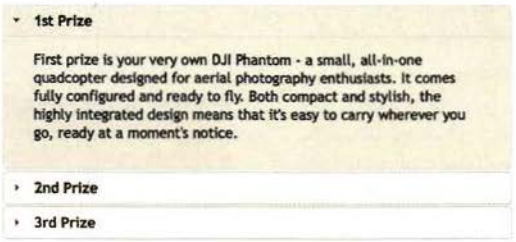
5. Finally, you can see a third `<script>` element containing an anonymous function that runs when the page has loaded.

6. Inside that function, a standard jQuery selector picks the containing `<div>` element that contains the accordion (using the value of its `id` attribute). The accordion functionality is triggered by calling the `.accordion()` method on that selection.

c09/jqui-accordion.html — HTML

```
<body>
  <div id="prizes">
    <h3>1st Prize</h3>
    <div><p>First prize is the DJI...</p></div>
    <h3>2nd Prize</h3>
    <div><p>Second prize is the...</p></div>
    <h3>3rd Prize</h3>
    <div><p>Third prize is a...</p></div>
  </div>
  <script src="js/jquery-1.9.1.js"></script>
  <script src="js/1.10.3/jquery-ui.js"></script>
  <script>
    $(function() {
      $('#prizes').accordion();
    });
  </script>
</body>
```

RESULT



You do not need to know how the jQuery plugin achieves this, as long as you know how to:

- Structure your HTML
- Create the jQuery selection
- Call the new method defined in the jQuery plugin

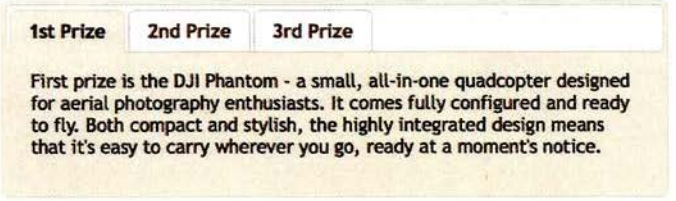
Note: On a live site, the JavaScript should be kept in an external file to maintain a separation of concerns. It is shown here for convenience and to show how little work needs to be done to achieve this effect.

JQUERY UI TABS

HTML c09/jqui-tabs.html

```
① )<div id="prizes">
    <ul>
      <li><a href="#tab-1">1st Prize</a></li>
②     <li><a href="#tab-2">2nd Prize</a></li>
      <li><a href="#tab-3">3rd Prize</a></li>
    </ul>
③   <div id="tab-1"><p>First prize is...</p></div>
    <div id="tab-2"><p>Second prize is...</p></div>
    <div id="tab-3"><p>Third prize is...</p></div>
  </div>
  <script src="js/jquery-1.9.1.js"></script>
  <script src="js/jquery-ui.js"></script>
  <script>
    $(function() {
④     $('#prizes').tabs();
    });
  </script>
```

RESULT



This structure is common in most jQuery plugins:

1. jQuery is loaded.
2. The plugin is loaded.
3. An anonymous function runs when the page is ready.

The anonymous function will create a jQuery selection and applies the method defined in the jQuery plugin to that selection. Some methods will also require parameters in order to do their job.

The tabs are a similar concept to the accordion.

1. They are kept in a containing `<div>` element that will be used in the jQuery selector. The content, however, is slightly different.

2. The tabs are created using an unordered list. The link inside each list item points to a `<div>` element lower down the page that holds content for that tab.

3. Note that the `id` attributes on the `<div>` elements must match the value of the `href` attribute on the tabs.

Once you have included jQuery and jQuery UI in the page, there is a third script tag with an anonymous function that runs when the DOM has loaded.

4. A jQuery selector picks the element whose `id` attribute has a value of `prizes` (this is the containing element for the tabs). Then it calls the `.tabs()` method is called on that selection.

On a live site, the JavaScript should be kept in an external file to maintain a separation of concerns, but it is shown here for convenience and to show how little work needs to be done to achieve this effect.

JQUERY UI FORM

jQuery UI introduces several form controls that make it easier for people to enter data into forms. This example demonstrates two of them:

Slider input: This allows people to select a numeric value using a draggable slider. This slider has two handles that allow the user to set a range between two numbers. As you can see on the right, the HTML for the slider is made up of two components:
1. A normal label and text input that would allow users to enter a number.
2. An extra `<div>` element used to hold the slider that you see on the page.

Date picker: This allows people to pick a date from a pop-up calendar, which helps ensure that users provide the date in the correct format that you need.
3. It is just a text input, and does not need any additional markup.

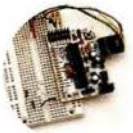
Before the closing `</body>` tag, you can see that there are three `<script>` elements: the first is the jQuery script, the second is jQuery UI, and the third contains the instructions to setup these two form controls (see right-hand page). If JavaScript is not enabled, these controls look like normal form controls without the jQuery's enhancements.

c09/jqui-form.html

HTML

```
<body> ...
  <h2>Find Accommodation</h2> ...
  <p id="price">
(1)   <label for="amount">Price range:</label>
    <input type="text" id="amount" />
  </p>
(2) <div id="price-range"></div>
  <p>
    <label for="arrival">Arrival date:</label>
(3)   <input type="text" id="arrival" />
  </p>
  <input type="submit" value="Find a hotel"/>

  <script src="js/jquery-1.9.1.js"></script>
  <script src="js/jquery-ui.js"></script>
  <script src="js/form-init.js"></script>
</body>
```

RESULT



Price range:
$175 - $300

Arrival date:
05/30/2015

Most jQuery scripts live within the `.ready()` function or its shortcut (used on the next page). As you saw in Chapter 7, this ensures that the script only runs when the DOM has loaded.

If you include more than one jQuery plugin, each of which uses the `.ready()` method, you do not repeat the function - you combine the code from inside both functions into the one.

1. The JavaScript is contained within the shortcut for the jQuery `.ready()` method. It contains the setup instructions for both of the form controls.

2. To turn a text input into a date picker, all you need to do is select the text input and then call the `datepicker()` method on that selection.

3. Cache the inputs for price.

4. The slider uses an object literal to set the properties of the `.slider()` method (see below).

JAVASCRIPT c09/js/form-init.js

```
① $(function() {

②   $('#arrival').datepicker();          // Turn input to JQUI datepicker

③   var $amount = $('#amount');          // Cache the price input
    var $range = $('#price-range');      // Cache the <div> for the price range

    $('#price-range').slider({           // Turn price-range input into a slider
      range: true,                       // If it is a range it gets two handles
      min: 0,                            // Minimum value
      max: 400,                          // Maximum value
④     values: [175, 300],                // Values to use when the page loads
      slide: function(event, ui) {       // When slider used update amount element
        $amount.val('$' + ui.values[0] + ' - $' + ui.values[1]);
      }
    });
    $amount                              // Set initial values of amount element
⑤     .val('$' + $range.slider('values', 0)        // A $ sign then lower range
      + ' - $' + $range.slider('values', 1));      // A $ sign then higher range

});
```

5. When the form loads, the text input that shows the amount as text needs to know the initial range for the slider. The value of that input is made up of:

a) A dollar sign: $ followed by the lower range value.

b) A dash and dollar sign: - $ followed by the higher range value.

The script is called `form-init.js`. Programmers often use **init** as a shorthand for initialize; and this script is used to set an initial state for the form.

When a jQuery plugin has settings that vary each time it is used, it is common to pass the settings in an object literal. You can see this with the `.slider()` method; it is passed several parameters and a method:

PROPERTY	DESCRIPTION
`range`	A Boolean to give the slider two handles (not just a single value)
`min`	The minimum value for the slider
`max`	The maximum value for the slider
`values`	An array containing two values to specify an initial range in the slider when the page first loads

METHOD	DESCRIPTION
`slider()`	Updates the text input which shows the text values for the slider (the documentation shows examples for this)

ANGULARJS

AngularJS is a framework that makes it easier to create web apps. In particular, it assists in creating apps that write, read, update, and delete data in a database on a server.

Angular is based on a software development approach called **model view controller** or **MVC**. (It is actually variant on MVC, not strict MVC). To use Angular, first you include the `angular.js` script in your page, and then it makes a set of tools available to you (just like jQuery does).

The point of MVC is that it separates out parts of a web application, in the same way that front-end developers should separate content (HTML), presentation (CSS), and behavior (JavaScript).

We do not have space to go into Angular in *detail*, but it introduces another example of a very different script with an API, as well as concepts such as the MVC approach, templating, and data binding. You can download Angular and view the full API at `http://angularjs.org`.

The **View** is what the user sees. In a web app, it is the HTML page. Angular lets you create templates with spaces for particular types of content. If the user changes values in the view, **commands (1)** are sent to up the chain to update the model. There can be different views of the same data, e.g., users and administrators.

This **ViewModel** (or *controller*) will update the view if there are changes to the model, and will update the model if there are changes in the view. The task of keeping data synchronized between the two is known as **data binding** (2). For example, if a form in the view is updated, it reflects the changes and updates the server.

In a web app, the **Model** is usually stored in the database, and managed by server-side code that can access and update the model.

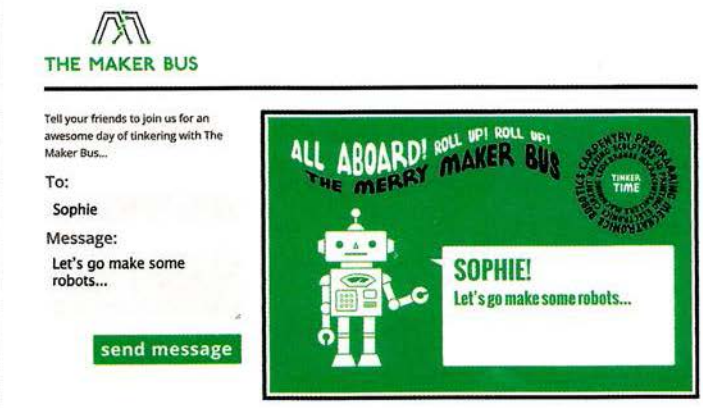
When the model has been updated, **change notifications (3)** are sent to the ViewModel. This info can be passed onto the View to keep it updated.

USING ANGULAR

HTML c09/angular-introduction.html

```
<!DOCTYPE html>
<html ng-app>
<head> ...
  <script src="https://ajax.googleapis.com/ajax/
  libs/angularjs/1.0.2/angular.min.js"></script>
</head>
<body> ...
  <form>
    To:<br>
    <input ng-model="name" type="text"/><br>
    Message:<br>
    <textarea ng-model="message"></textarea>
    <input type="submit" value="send message" />
  </form> ...
  <div class="postcard">
    <div>{{ name }}</div>
    <p>{{ message }}</p>
  </div> ...
</body>
</html>
```

RESULT



This example takes the content of the <input> and <textarea> elements and writes it into another part of the page (where you can see the double curly braces in the HTML file).

First, include the Angular script in your page. You can store it locally or use the version on Google's CDN. Until you understand more about Angular, place it in the <head> element.

Note the new markup in the HTML. There are attributes that start with ng- (which is short for Angular). These are called **directives**. There is one on the opening <html> tag and one on each of the form elements. The value of the ng-model attribute on the text inputs matches the values inside the double curly braces. Angular *automatically* takes the content of the form elements and writes it into the page where the corresponding curly braces are.

No more JavaScript is needed to achieve this, whereas in jQuery, this would involve four steps:

1. Writing an event handler for the form elements
2. Using that to trigger code to get the elements' content
3. Selecting new element nodes that represent the postcard
4. Writing the data into the page

VIEW & VIEWMODEL

Below, look at the `angular-controller.js` file. It uses a a constructor function to create an object called `BasketCtrl`. This object is known as a **controller** or **ViewModel**. It is passed another object called `$scope` as an argument. Properties of the `$scope` object are set in the constructor function.

1. Note the object's name (`BasketCtrl`) matches the value of the `ng-controller` attribute on the opening `<table>` tag. In this example, there is no database, so the controller will also act as the model: sharing data with the view.

The HTML file (the view) gets its data from the `BasketCtrl` object in the JavaScript controller. In the HTML, note how the names in curly braces, e.g., `{{ cost }}` and `{{ qty }}`, match the properties of the $scope object in the JavaScript.

The HTML file is now called a **template** because it will display whatever data is in the corresponding controller. The names in curly braces are like variables that match the data in the object. If the JavaScript object had different values, the HTML would show those values.

c09/angular-controller.html

HTML

```
<!DOCTYPE html>
<html ng-app>
  <head>
    <title>JavaScript &amp;amp; jQuery - Chapter 9 ...</title>
    <script src="https://ajax.googleapis.com/.../angular.min.js"></script>
    <script src="js/angular-controller.js"></script>
    <link rel="stylesheet" href="css/c09.css">
  </head>
  <body> ...
(1)  <table ng-controller="BasketCtrl">
       <tr><td>Item:</td><td>{{ description }}</td></tr>
(2)    <tr><td>Cost:</td><td>${{ cost }}</td></tr>
       <tr><td>Qty:</td><td><input type="number" ng-model="qty"></td></tr>
(3)    <tr><td>Subtotal:</td><td>{{qty * cost | currency}}</td></tr>
     </table> ...
  </body>
</html>
```

c09/js/angular-controller.js

JAVASCRIPT

```
(1) function BasketCtrl($scope) {
      $scope.description = 'Single ticket';
(2)   $scope.cost = 8;
      $scope.qty = 1;
(3) }
```

2. It is also possible to evaluate expressions inside the curly braces. In step 3, the subtotal is calculated in the template. This is then formatted as a currency. Furthermore, if you update the quantity in the form, the underlying data model (in the JavaScript object) is updated along with the subtotal. Try updating the values in the JavaScript file, then refreshing the HTML to see the connection. This is an example of something programmers call **data binding**; the data in the JavaScript file is bound to the HTML and vice-versa. If the ViewModel changes, the view updates. If the view changes, the ViewModel updates.

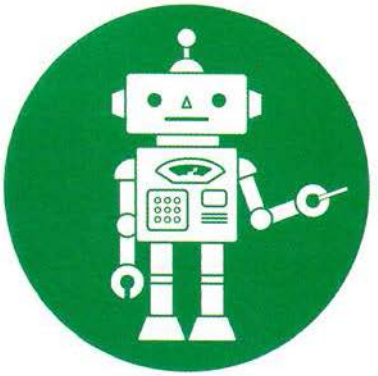
As this shows, Angular is particularly helpful when you load data from a separate file into the view.

A page can have multiple controllers, each of which has its own **scope**. In the HTML, the `ng-controller` attribute is used on an element to define the scope of that controller. This is similar to variable scope. For example, a different element might have a different controller (e.g., `StoreCtrl`), and both controllers would be able to have a property called `description`. Because the scope is only within that element, each controller's `description` property would only be used within that controller's scope.

RESULT

THE MAKER BUS

Buy tickets



Item: Single ticket

Cost: $8

Qty: 1

Subtotal: $8.00

GETTING EXTERNAL DATA

Here, the controller (the JavaScript file) collects the model (the JSON data) from a file on the server. (In a web app, the JSON data would usually come from a database.) This updates the view in the HTML.

To collect the data, Angular uses what it calls the **`$http` service**. Inside the `angular.js` file, the code uses the `XMLHttpRequest` object to make Ajax requests (like those you saw in Chapter 8).

1. The path to the JSON file is relative to the HTML template, *not* the JavaScript file (even though the path is written in the JavaScript).

Just like jQuery's `.ajax()` method, the `$http` service has several shortcuts to make it easier to create some requests. To fetch data it uses `get()`, `post()`, and `jsonp()`; to delete data it uses `delete()`; and to create new records: `put()`. This example uses `get()`.

c09/angular-external-data.html

HTML

```
<table ng-controller="TimetableCtrl">
  <tr><th>time</th><th>title</th><th>detail</th></tr>
⑤ <tr ng-repeat="session in sessions">
    <td>{{ session.time }}</td>
    <td>{{ session.title }}</td>
    <td>{{ session.detail }}</td>
  </tr>
</table>
```

c09/js/angular-external-data.js

JAVASCRIPT

```
function TimetableCtrl($scope, $http) {
① $http.get('js/items.json')
② .success(function(data) { $scope.sessions = data.sessions; })
③ .error(function(data) { console.log('error') });
  // The error could show a friendly message to users...
}
```

c09/js/items.json

JAVASCRIPT

```
{
  "sessions": [
④   {"time": "09.00", "title": "Intro to 3D Modeling", "detail": "Come..."}
    {"time": "10.00", "title": "Circuit Hacking", "detail": "Head to the..."}
    {"time": "11.30", "title": "Arduino Antics", "detail": "Learn how..."}
  ]
}
```

LOOP THROUGH RESULTS

2. If the request successfully fetches data, the code in the `success()` function runs. In this case, if it is successful the `$scope` object is passed the data from the `JSON` object. This allows the template to display the data.

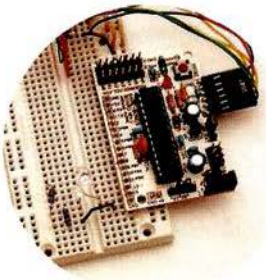
3. If it fails, the `error()` function is run instead. This would to show an error message to users. Here it writes to the console (which you meet on p464).

4. The JSON data contains several objects, each of which is displayed in the page. Note, there is no JavaScript loop written in the controller. Instead, the HTML template (or view) is where the loop occurs.

5. The `ng-repeat` directive on the opening `<tr>` tag indicates that the table row should act like a loop. It should go through each object in the `sessions` array and create a new table row for each of them.

RESULT

Session Times



TIME	TITLE	DETAIL
09.00	Intro to 3D Modeling	Come learn how to create 3D models of parts you can then make on our bus! You'll get to know the same 3D modeling software that used worldwide in professional settings like engineering, product design, and more. Develop and test ideas in a fun and informative session hosted by Bella Stone, professional roboticist.
10.00	Circuit Hacking	Head to the Electro-Tent for a free introductory soldering lesson. There will be electronics kits on hand for those who wish to make things, and experienced hackers and engineers around to answer all your

In the HTML, the value of the `ng-repeat` directive is: `session in sessions`

- `sessions` matches the JSON data; it corresponds with the object name.
- `session` is the identifier used in the template to indicate the name of each individual object within the `sessions` object.

If the `ng-repeat` attribute used different names than `session`, the value in the curly braces in the HTML would have to change to reflect that name. For example, if it said `lecture in sessions`, then the curly braces would change to reflect that: `{{ lecture.time }}`, `{{ lecture.title }}`, etc.

This is just a very high-level introduction to Angular, but does demonstrate some popular techniques when using JavaScript to develop web apps, such as:

- The use of templates that take content from JavaScript and update the HTML page.
- The rise in MVC-influenced frameworks for web-based application development.
- The use of libraries to save developers having to write so much code.

For more on Angular, see `http://angularjs.org`

Another very popular alternative is Backbone `http://backbonejs.org`

PLATFORM APIS

Many large websites expose their APIs that allow you to access and update the data on their sites, including Facebook, Google, and Twitter.

WHAT YOU CAN DO

Each site offers different capabilities, for example:

- Facebook offers features such as allowing people to like sites or add comments and discussion to the bottom of a web page.
- Google Maps lets you to include various types of maps in your pages.
- Twitter allows you to display your latest tweets on your web pages or send new tweets.

By exposing some of the functionality of their platforms these companies are advertising their sites and encouraging people back to them. This in turn increase their total amount of activity (and their revenue).

Be aware that companies can change either how you access APIs or change what you are allowed to use the APIs for.

HOW TO ACCESS

On the web, you can access several of these platform APIs by including a script they provide in your page. That script will typically create an object (just like the jQuery script adds a `jQuery` object). In turn, that object will have methods and properties that you can use to access (and sometimes update) the data on that platform.

Most sites that offer an API will also provide documentation that explains how to use its objects, methods, and properties (along with some basic examples).

Some of the larger sites provide pages where you can get code that you can copy and paste into your site without even needing to understand the API.

Facebook, Google, and Twitter have all made changes to both how you access their APIs *and* what you can use them for.

THE SYNTAX

The syntax of an API will vary from platform to platform. But they will be documented using tables of objects, methods, and properties like those you saw in the first section of this chapter. You may also see sample code that demonstrates tasks people commonly use the API for (like the examples you have seen in this chapter).

Some platforms offer APIs in multiple languages, so that you can interact with them using server-side languages such as PHP / C# as well as using JavaScript.

In the rest of this chapter we will be focusing on the Google Maps API as an example of what you can do with platform APIs.

If you work on a site for a client, make them aware that APIs can change (and that could result in recoding pages that use them).

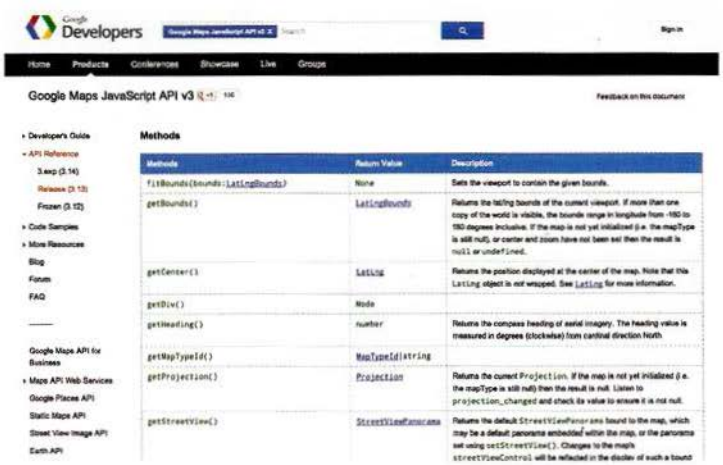
GOOGLE MAPS API

Currently, one of the most popular APIs in use on the web is the Google Maps API, which allows you to add maps to web pages.

WHAT IT DOES

The Google Maps JavaScript API allows you to show Google maps in your web pages. It also allows you to customize the look of the maps and what information is shown on them.

You may find it helpful to look at the documentation for the Google Maps API while going through this example. It will show you other things that you can do with the API. `https://developers.google.com/maps/`



WHAT YOU'LL SEE

We only have space to show a few of the features of the Google Maps API, as it is very powerful and contains a lot of advanced features. But the examples in this chapter will get you used to working with its API.

You will start by seeing how to add a map to your web pages, then you will see how to change the controls, and finally how to change the colors and add markers on top of the map.

API KEY

Some APIs require that you register and request an API key in order to get data from their servers. An API key is a set of letters and numbers that uniquely identify you to the application so the owners of the site can track how much you use the API and what you use it for.

At the time of writing, Google allowed websites to call their maps API 25,000 times per day for free without an API key, but sites that consistently make more requests are required to use a key and pay for the service.

If you run a busy site, or the map is part of the core application, it is good practice to use an API key with Google Maps because:

- You can see how many times your site requests the API
- Google can contact you if they change terms of service or charge for use

To get a Google API key, see `https://cloud.google.com/console`

BASIC MAP SETTINGS

Once you have included the Google Maps script in your page, you can use their `maps` object. It lets you display Google maps in your pages.

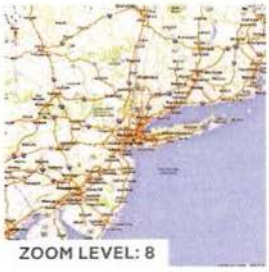
CREATING A MAP

The `maps` object is stored within an object called `google`. This creates scope for all Google objects.

To add a map to your page, you create a new map object using a constructor: `Map()`. The constructor is part of the `maps` object, and it has two parameters:

- The element into which you want the map drawn
- A set of map options that control how it is displayed given using object literal notation

Zoom level is typically set using a number between 0 (the full earth) and 16. (Some cities can go higher.)



MAP OPTIONS

The settings that control how the map should look are stored inside another JavaScript object called `mapOptions`. It is created as an object literal before you call the `Map()` constructor. In the JavaScript on the right, you can see that the `mapOptions` object uses three pieces of data:

- Longitude and latitude of the center of the map
- The zoom level for the map
- The type of map data you want to show

The images that make up the map are called tiles. Four **map types** each show a different style of map.

A BASIC GOOGLE MAP

HTML c09/google-map.html

```
    <div id="map"></div>
    <script src="js/google-map.js"></script>
  </body>
```

JAVASCRIPT c09/js/google-map.js

```
function init() {
  var mapOptions = {                                    // Set up the map options
   center: new google.maps.LatLng(40.782710,-73.965310),
   mapTypeId: google.maps.MapTypeId.ROADMAP,
   zoom: 13
  };
  var venueMap;                                         // Map() draws a map
  venueMap = new google.maps.Map(document.getElementById('map'), mapOptions);
}

function loadScript() {
  var script = document.createElement('script');        // Create <script> element
  script.src = 'http://maps.googleapis.com/maps/api/js?
                        sensor=false&callback=initialize';
  document.body.appendChild(script);                    // Add element to page
}

window.onload = loadScript;                             // Onload call
```

RESULT

1. Starting at the bottom of the script, when the page has loaded, the `onload` event will call the `loadScript()` function.

2. `loadScript()` creates a `<script>` element to load the Google Maps API. When it has loaded, it calls `init()`, to initialize the map.

3. `init()` loads the map into the HTML page. First it creates a `mapOptions` object with three properties.

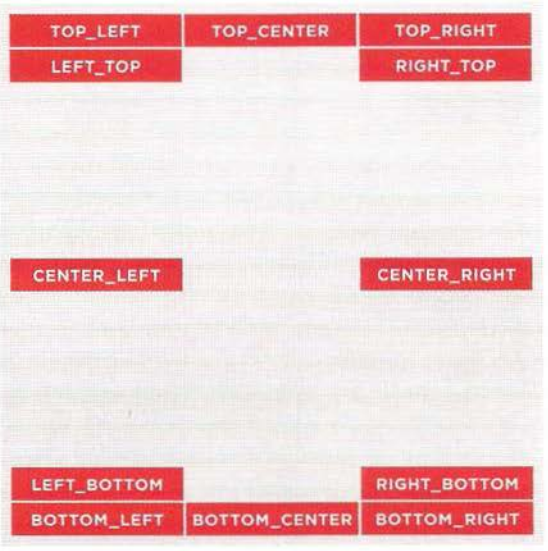
4. Then it uses the `Map()` constructor to create a map and draw the map into the page. The constructor takes two parameters:

- The element that the map will appear inside
- The `mapOptions` object

CHANGING CONTROLS

VISIBILITY OF MAP CONTROLS

POSITION OF MAP CONTROLS



To show or hide the controls, use the control name followed by a value of `true` (to show it) or `false` (to hide it). Although Google Maps tries to prevent overlaps, use judgement to position controls on your map.

CONTROL	DESCRIPTION	DEFAULT
`zoomControl` (1)	Sets the zoom level of the map. It uses a slider (for large maps) "+/-" buttons (for small maps)	On
`panControl` (2)	Allows panning across the map	On for non-touch devices
`scaleControl` (3)	Shows the scale of the map	Off
`mapTypeControl` (4)	Switch map types (e.g., `ROADMAP` and `SATELLITE`)	On
`streetViewControl` (5)	A Pegman icon that can be dragged and dropped onto the map to show a street view	On
`rotateControl`	Rotates maps that have oblique imagery (not shown)	On when available
`overviewMapControl`	A thumbnail showing a larger area, that reflects where the current map is within that wider area (not shown)	On when map is collapsed, e.g., street view

GOOGLE MAP WITH CUSTOM CONTROLS

APPEARANCE OF CONTROLS

To alter the appearance and position of map controls, you add to the `mapOptions` object.

1. To show or hide a control, the key is the name of the control, and the value is a Boolean (`true` will show the control; `false` will hide it).

POSITION OF THE CONTROL

2. Each control has its own options object used to control its style and position. The word `Options` follows the control name, e.g., `zoomControlOptions`. Styles are discussed below. The diagram on the left-hand page shows options for the `position` property.

JAVASCRIPT

c09/js/google-map-controls.js

```
var mapOptions = {
  zoom: 14,
  center: new google.maps.LatLng(40.782710,-73.965310),
  mapTypeId: google.maps.MapTypeId.ROADMAP,

(1) panControl: false,
(1) zoomControl: true,
    zoomControlOptions: {
(3)   style: google.maps.ZoomControlStyle.SMALL,
(2)   position: google.maps.ControlPosition.TOP_RIGHT
    },
(1) mapTypeControl: true,
    mapTypeControlOptions: {
(3)   style: google.maps.MapTypeControlStyle.DROPDOWN_MENU,
(2)   position: google.maps.ControlPosition.TOP_LEFT
    },
(1) scaleControl: true,
    scaleControlOptions: {
(2)   position: google.maps.ControlPosition.TOP_CENTER
      },
(1)   streetViewControl: false,
(1)   overviewMapControl: false
};
```

STYLE OF MAP CONTROLS

3. You can change the appearance of the zoom and map type controls using the following options:

`zoomControlStyle`:

`SMALL`	Small +/- buttons
`LARGE`	Vertical slider
`DEFAULT`	The default for that device

`MapTypeControlStyle`:

`HORIZONTAL_BAR`	Buttons side-by-side
`DROPDOWN_MENU`	Dropdown select box
`DEFAULT`	The default for that device

STYLING A GOOGLE MAP

To style the map you need to specify three things:

- `featureTypes`: the map feature you want to style: e.g., roads, parks, waterways, public transport.
- `elementTypes`: the part of that feature you want to style, such as its geometry (shapes) or labels.
- `stylers`: properties that allow you to adjust the color or visibility of items on the map.

The `styles` property in the `mapOptions` object sets the map style. It's value is an array of objects. Each object affects a different feature fo the map.

The first `stylers` property alters the colors of the map as a whole. It, too, contains an array of objects.

- `hue` property adjusts color, its value is a hex code
- `lightness` or `saturation` can take a value from -100 to 100

Then each feature that shows up on the map can have its own object, and its own `stylers` property. In it, the `visibility` property can have three values:

- `on` to show the feature type
- `off` to hide it
- `simplified` to show a more basic version

c09/js/google-map-styled.js

JAVASCRIPT

```
styles: [                          // styles property is an array of objects
  {
    stylers: [                     // stylers property holds array of objects
      { hue: "#00ff6f" },          // Overall map colors
      { saturation: -50 }          // Overall map saturation
    ]
  }, {
    featureType: "road",           // Road features
    elementType: "geometry",       // Their geometry (lines)
    stylers: [
      { lightness: 100 },          // Lightness of roads
      { visibility: "simplified" } // Level of road detail
    ]
  }, {
    featureType: "transit",        // Public transport features
    elementType: "geometry",       // Their geometry (lines)
    stylers: [
      { hue: "#ff6600" },          // Color of public transport
      { saturation: +80 }          // Saturation of public transport
    ]
  }, {
    featureType: "transit",        // Public transport features
    elementType: "labels",         // Their labels
    stylers: [
      { hue: "#ff0066" },          // Label color
      { saturation: +80 }          // Label saturation
    ]
  } ...                            // More stylers shown in the code download
```

ADDING MARKERS

Here you can see how to add a **marker** to a map. The map has been created, and its name is `venueMap`.

1. Create a `LatLng` object to store the position of the marker using object constructor syntax. Below that object is called `pinLocation`.
2. The `Marker()` constructor creates a `marker` object. It has one parameter: an object that contains settings using object literal notation.

The `settings` object contains three properties:

3. `position` is the object storing the location of the marker (`pinLocation`).
4. `map` is the map that the marker should be added to (because a page can have more than one map).
5. `icon` is the path to the image that should be displayed as the marker on the map (this should be provided relative to the HTML page).

JAVASCRIPT c09/js/google-map-styled.js

```
① var pinLocation = new google.maps.LatLng(40.782710,-73.965310);

② var startPosition = new google.maps.Marker({          // Create a new marker
③   position: pinLocation,                              // Set its position
④   map: venueMap,                                      // Specify the map
⑤   icon: "img/go.png"                                  // Path to image from HTML
});
```

RESULT

THE MAKER BUS

Naumberg Bandshell
Central Park
New York, NY 10019

SUMMARY
APIS

- APIs are used in browsers, scripts, and by websites that share functionality with other programs or sites.
- APIs let you write code that will make a **request**, asking another program or script to do something.
- APIs also specify the format in which the **response** will be given (so that the response can be understood).
- To use an API on your website, you will need to include a script in the relevant web pages.
- An API's documentation will usually feature tables of objects, methods, and properties.
- Providing you know how to create an object and call its methods, access its properties, and respond to its events, you should be able to learn any JavaScript API.

10

ERROR HANDLING & DEBUGGING

JavaScript can be hard to learn and everyone makes mistakes when writing it. This chapter will help you learn how to find the errors in your code. It will also teach you how to write scripts that deal with potential errors gracefully.

When you are writing JavaScript, do not expect to write it perfectly the first time. Programming is like problem solving: you are given a puzzle and not only do *you* have to solve it, but you also need to create the instructions that allow the *computer* to solve it, too.

When writing a long script, nobody gets everything right in their first attempt. The error messages that a browser gives look cryptic at first, but they can help you determine what went wrong in your JavaScript and how to fix it. In this chapter you will learn about:

THE CONSOLE & DEV TOOLS

Tools built into the browser that help you hunt for errors.

COMMON PROBLEMS

Common sources of errors, and how to solve them.

HANDLING ERRORS

How code can deal with potential errors gracefully.


Find the area of a wall:
width
height
Calculate area

ORDER OF EXECUTION

To find the source of an error, it helps to know how scripts are processed. The order in which statements are executed can be complex; some tasks cannot complete until another statement or function has been run:

```
function greetUser() {
  return 'Hello ' + getName();
}

function getName() {
  var name = 'Molly';
  return name;
}

var greeting = greetUser();
alert(greeting);
```

This script above creates a greeting message, then writes it to an alert box (see right-hand page). In order to create that greeting, two functions are used: `greetUser()` and `getName()`.

You might think that the **order of execution** (the order in which statements are processed) would be as numbered: one through to four. However, it is a little more complicated.

To complete step one, the interpreter needs the results of the functions in steps two *and* three (because the message contains values returned by those functions). The order of execution is more like this: 1, 2, 3, 2, 1, 4.

1. The `greeting` variable gets its value from the `greetUser()` function.

2. `greetUser()` creates the message by combining the string `'Hello '` with the result of `getName()`.

3. `getName()` returns the name to `greetUser()`.

2. `greetUser()` now knows the name, and combines it with the string. It then returns the message to the statement that called it in step 1.

1. The value of the `greeting` is stored in memory.

4. This `greeting` variable is written to an alert box.

EXECUTION CONTEXTS

The JavaScript interpreter uses the concept of **execution contexts**. There is one global execution context; plus, each function creates a new new execution context. They correspond to variable scope.

EXECUTION CONTEXT

Every statement in a script lives in one of three execution contexts:

- **GLOBAL CONTEXT**
  Code that is in the script, but not in a function. There is only one global context in any page.

- **FUNCTION CONTEXT**
  Code that is being run within a function. Each function has its own function context.

- **EVAL CONTEXT (NOT SHOWN)**
  Text is executed like code in an internal function called `eval()` (which is not covered in this book).

VARIABLE SCOPE

The first two execution contexts correspond with the notion of scope (which you met on p98):

- **GLOBAL SCOPE**
  If a variable is declared outside a function, it can be used anywhere because it has global scope. If you do not use the `var` keyword when creating a variable, it is placed in global scope.

- **FUNCTION-LEVEL SCOPE**
  When a variable is declared within a function, it can only be used within that function. This is because it has function-level scope.

THE STACK

The JavaScript interpreter processes one line of code at a time. When a statement needs data from another function, it **stacks** (or piles) the new function on top of the current task.

When a statement has to call some other code in order to do its job, the new task goes to the top of the pile of things to do.

Once the new task has been performed, the interpreter can go back to the task in hand.

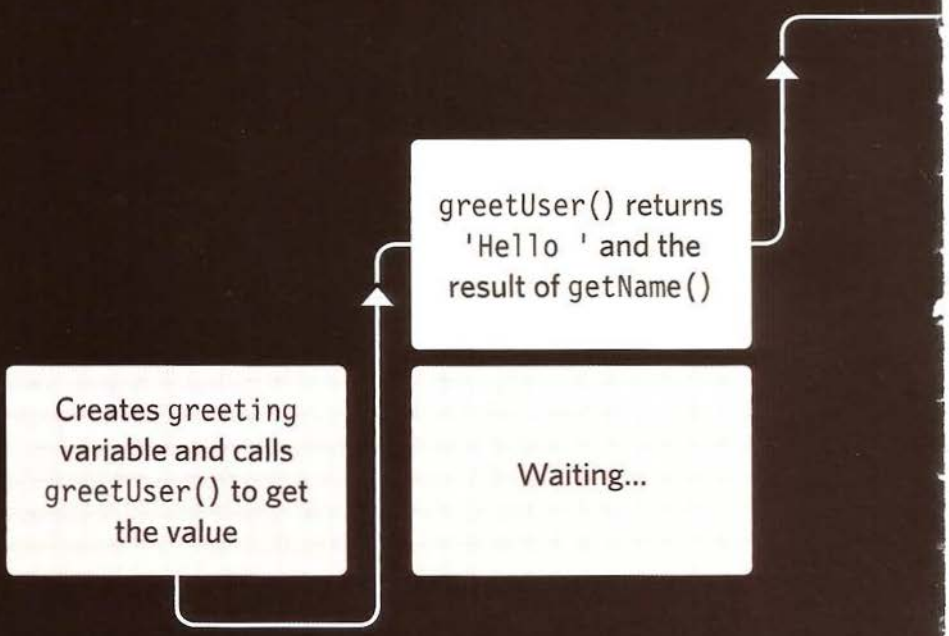
Each time a new item is added to the stack, it creates a new execution context.

Variables defined in a function (or execution context) are only available in that function.

If a function gets called a second time, the variables can have different values.

You can see how the code that you have been looking at so far in this chapter will end up with tasks being stacked up on each other in the diagram to the right.

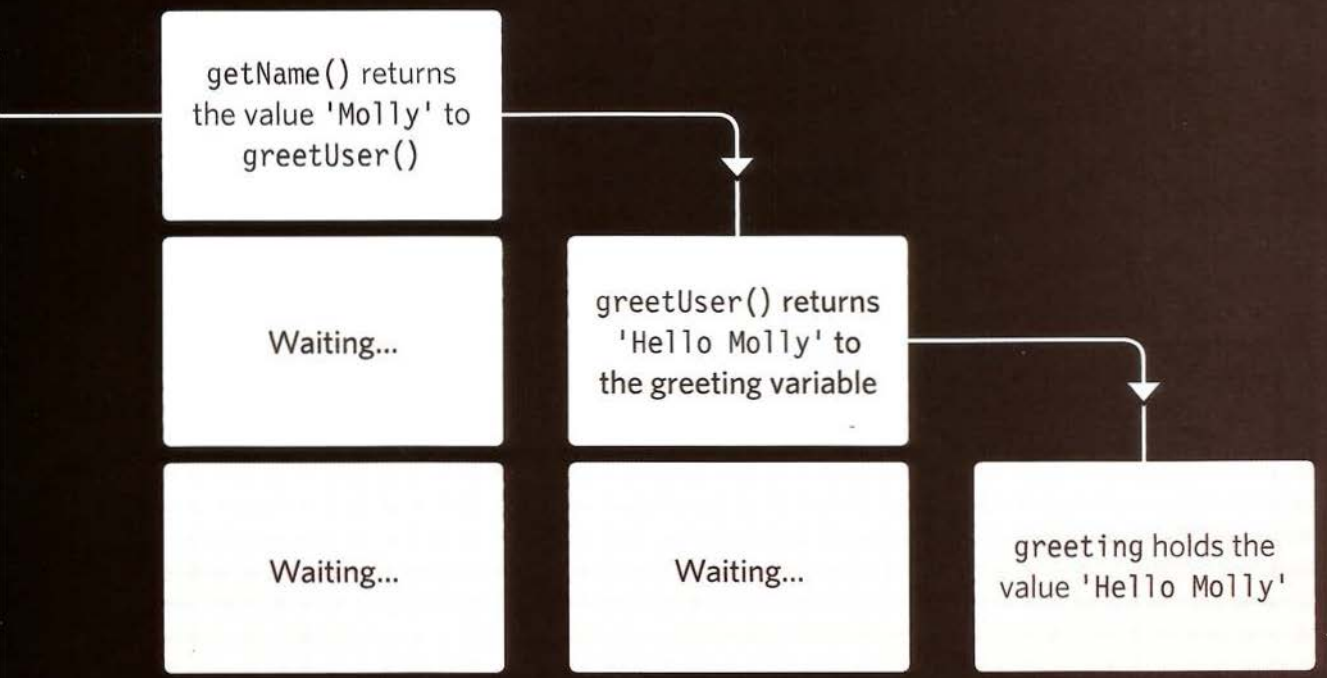
(The code is shown at the top of the right-hand page.)



The value for the **greeting** variable is obtained by calling the **greetUser()** function. So the variable cannot be assigned until the **greetUser()** function has done its job.

The statement is effectively put on hold, and the **greetUser()** task gets stacked on top it. In turn, the **greetUser()** function cannot return a value until the **getName()** function has completed *its* task.

```
function greetUser() {
  return 'Hello ' + getName();
}

function getName() {
  var name = 'Molly';
  return name;
}

var greeting = greetUser();
alert(greeting);
```



So, **getName()** is stacked on top of the **greetUser()** function. You can see the stack starting to build up. When **getName()** has done its job, a value is returned back to the **greetUser()** function.

Since **getName()** has done its job, it is removed from the stack. In turn, the **greetUser()** function can now finish its job and return a value to the **greeting** variable.

The **greetUser()** function has finished its work and it is removed from the stack and the value is finally assigned to the **greeting** variable.

EXECUTION CONTEXT & HOISTING

Each time a script enters a new execution context, there are two phases of activity:

1: PREPARE

- The new scope is created
- Variables, functions, and arguments are created
- The value of the `this` keyword is determined

2: EXECUTE

- Now it can assign values to variables
- Reference functions and run their code
- Execute statements

Understanding that these two phases happen helps with understanding a concept called **hoisting**. You may have seen that you *can*:

- Call functions *before* they have been declared (if they were created using function declarations – not function expressions, see p96)
- Assign a value to a variable that has not yet been declared

This is because any variables and functions within each execution context are created before they are executed.

The preparation phase is often described as taking all of the variables and functions and hoisting them to the top of the execution context. Or you can think of them as having been *prepared*.

Each execution context also creates its own **`variables` object**. This object contains details of all of the `variables`, functions, and parameters for that execution context.

You may expect the following to fail, because `greetUser()` is called before it has been defined:

```
var greeting = greetUser();
function greetUser() {
  // Create greeting
}
```

It works because the function and first statement are in the same execution context, so it is treated like this:

```
function greetUser() {
  // Create greeting
}
var greeting = greetUser();
```

The following would would fail because `greetUser()` is created within the `getName()` function's context:

```
var greeting = greetUser();
function getName() {
  function greetUser() {
    // Create greeting
  }
  // Return name with greeting
}
```

UNDERSTANDING SCOPE

In the interpreter, each execution context has its own `variables` object. It holds the variables, functions, and parameters available within it. Each execution context can also access its parent's `variables` object.

Functions in JavaScript are said to have **lexical scope**. They are linked to the object they were defined *within*. So, for each execution context, the scope is the current execution context's `variables` object, *plus* the `variables` object for each parent execution context.

Imagine that each function is a nesting doll. The children can ask the parents for information in their variables. But the parents cannot get variables from their children. Each child will get the same answer from the same parent.

```
var greeting = (function() {
  var d = new Date();
  var time = d.getHours();
  var greeting = greetUser();

  function greetUser() {
    if (time < 12) {
      var msg = 'Good morning ';
    } else {
      var msg = 'Welcome ';
    }
    return = msg + getName();

    function getName() {
      var name = 'Molly';
      return name;
    }

  }

});
alert(greeting);
```

If a variable is not found in the `variables` object for the current execution context, it can look in the `variables` object of the parent execution context. But it is worth knowing that looking further up the stack can affect performance, so ideally you create variables inside the functions that use them.

If you look at the example on the left, the inner functions can access the outer functions and their variables. For example, the `greetUser()` function can access the `time` variable that was declared in the outer `greeting()` function.

Each time a function is called, it gets its own execution context and `variables` object.

Each time an outer function calls an inner function, the inner function can have a new `variables` object. But variables in the outer function remain the same.

Note: you cannot access this `variables` object from your code; it is something the interpreter is creating and using behind the scenes. But understanding what goes on helps you understand scope.

UNDERSTANDING ERRORS

If a JavaScript statement generates an error, then it throws an **exception**. At that point, the interpreter stops and looks for exception-handling code.

If you are anticipating that something in your code may cause an error, you can use a set of statements to **handle** the error (you meet them on p480). This is important because if the error is not handled, the script will just stop processing and the user will not know why. So exception-handling code should inform users when there is a problem.

```
function greetUser() {
2 // Interpreter looks here
}

function getName() {
3 // Imagine this had an error
// It was caused by greetUser()
}

1 var greeting = greetUser();
4 alert(greeting);
```

Whenever the interpreter comes across an error, it will look for error-handling code. In the diagram below, the code has the same structure as the code you saw in the diagrams at the start of the chapter. The statement at step 1 uses the function in step 2, which in turn uses the function in step 3. Imagine that there has been an error at step 3.

When an exception is thrown, the interpreter stops and checks the current execution context for exception-handling code. So if the error occurs in the `getName()` function (3), the interpreter starts to look for error handling code in that function.

If an error happens in a function and the function does not have an exception handler, the interpreter goes to the line of code that called the function. In this case, the `getName()` function was called by `greetUser()`, so the interpreter looks for exception-handling code in the `greetUser()` function (2). If none is found, it continues to the next level, checking to see if there is code to handle the error in that execution context. It can continue until it reaches the global context, where it would have to it terminate the script, and create an `Error` object.

So it is going through the stack looking for error-handling code until it gets to the global context. If there is still no error handler, the script stops running and the `Error` object is created.

ERROR OBJECTS

Error objects can help you find where your mistakes are and browsers have tools to help you read them.

When an `Error` object is created, it will contain the following properties:

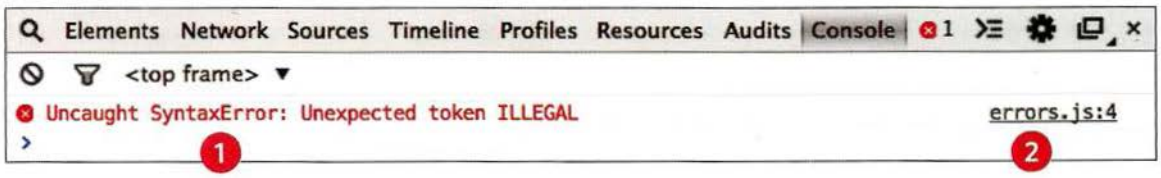
PROPERTY	DESCRIPTION
`name`	Type of execution
`message`	Description
`fileNumber`	Name of the JavaScript file
`lineNumber`	Line number of error

When there is an error, you can see all of this information in the JavaScript console / Error console of the browser.

You will learn more about the console on p464, but you can see an example of the console in Chrome in the screen shot below.

There are seven types of built-in error objects in JavaScript. You'll see them on the next two pages:

OBJECT	DESCRIPTION
`Error`	Generic error - the other errors are all based upon this error
`SyntaxError`	Syntax has not been followed
`ReferenceError`	Tried to reference a variable that is not declared/within scope
`TypeError`	An unexpected data type that cannot be coerced
`RangeError`	Numbers not in acceptable range
`URIError`	`encodeURI()`, `decodeURI()`, and similar methods used incorrectly
`EvalError`	`eval()` function used incorrectly



1. In the red on the left, you can see this is a `SyntaxError`. An unexpected character was found.

2. On the right, you can see that the error happened in a file called `errors.js` on line 4.

ERROR OBJECTS CONTINUED

Please note that these error messages are from the Chrome browser. Other browsers' error messages may vary.

SyntaxError

SYNTAX IS NOT CORRECT

This is caused by incorrect use of the rules of the language. It is often the result of a simple typo.

MISMATCHING OR UNCLOSED QUOTES

```
document.write("Howdy');
```

```
SyntaxError: Unexpected EOF
```

MISSING CLOSING BRACKET

```
document.getElementById('page'
```

```
SyntaxError: Expected token ')'
```

MISSING COMMA IN ARRAY

Would be same for missing] at the end

```
var list = ['Item 1', 'Item 2' 'Item 3'];
```

```
SyntaxError: Expected token ']'
```

MALFORMED PROPERTY NAME

It has a space but is not surrounded by quote marks

```
user = {first name: "Ben", lastName: "Lee"};
```

```
SyntaxError: Expected an identifier but
found 'name' instead
```

ReferenceError

VARIABLE DOES NOT EXIST

This is caused by a variable that is not declared or is out of scope.

VARIABLE IS UNDECLARED

```
var width = 12;
var area = width * height;
```

```
ReferenceError: Can't find variable:
height
```

NAMED FUNCTION IS UNDEFINED

```
document.write( randomFunction() );
```

```
ReferenceError: Can't find variable:
randomFunction
```

EvalError

INCORRECT USE OF `eval()` FUNCTION

The `eval()` function evaluates text through the interpreter and runs it as code (it is not discussed in this book). It is rare that you would see this type of error, as browsers often throw other errors when they are supposed to throw an `EvalError`.

URIError

INCORRECT USE OF URI FUNCTIONS

If these characters are not escaped in URIs, they will cause an error: / ? & # : ;

CHARACTERS ARE NOT ESCAPED

```
decodeURI('http://bbc.com/news.php?a=1');
```

```
URIError: URI error
```

These two pages show JavaScript's seven different types of error objects and some common examples of the kinds of errors you are likely to see. As you can tell, the errors shown by the browsers can be rather cryptic.

TypeError

VALUE IS UNEXPECTED DATA TYPE

This is often caused by trying to use an object or method that does not exist.

INCORRECT CASE FOR **document** OBJECT

```
Document.write('Oops!');
```

```
TypeError: 'undefined' is not a function
(evaluating 'Document.write('Oops!')')
```

INCORRECT CASE FOR **write()** METHOD

```
document.Write('Oops!');
```

```
TypeError: 'undefined' is not a function
(evaluating 'document.Write('Oops!')')
```

METHOD DOES NOT EXIST

```
var box = {};      // Create empty object
box.getArea();     // Try to access getArea()
```

```
TypeError: 'undefined' is not a function
(evaluating 'box.getArea()')
```

DOM NODE DOES NOT EXIST

```
var el = document.getElementById('z');
el.innerHTML = 'Mango';
```

```
TypeError: 'null' is not an object
(evaluating 'el.innerHTML = 'Mango'')
```

RangeError

NUMBER OUTSIDE OF RANGE

If you call a function using numbers outside of its accepted range.

CANNOT CREATE ARRAY WITH -1 ITEMS

```
var anArray = new Array(-1);
```

```
RangeError: Array size is not a small
enough positive integer
```

NUMBER OF DIGITS AFTER DECIMAL IN **toFixed()** CAN ONLY BE 0-20

```
var price = 9.99;
price.toFixed(21);
```

```
RangeError: toFixed() argument must be
between 0 and 20
```

NUMBER OF DIGITS IN **toPrecision()** CAN ONLY BE 1-21

```
num = 2.3456;
num.toPrecision(22);
```

```
RangeError: toPrecision() argument must
be between 1 and 21
```

Error

GENERIC ERROR OBJECT

The generic Error object is the template (or prototype) from which all other error objects are created.

NaN

NOT AN ERROR

Note: If you perform a mathematical operation using a value that is not a number, you end up with the value of NaN, not a type error.

NOT A NUMBER

```
var total = 3 * 'Ivy';
```

HOW TO DEAL WITH ERRORS

Now that you know what an error is and how the browser treats them, there are two things you can do with the errors.

1: DEBUG THE SCRIPT TO FIX ERRORS

If you come across an error while writing a script (or when someone reports a bug), you will need to debug the code, track down the source of the error, and fix it.

You will find that the developer tools available in every major modern browser will help you with this task. In this chapter, you will learn about the developer tools in Chrome and Firefox. (The tools in Chrome are identical to those in Opera.)

IE and Safari also have their own tools (but there is not space to cover them all).

2: HANDLE ERRORS GRACEFULLY

You can handle errors gracefully using `try`, `catch`, `throw`, and `finally` statements.

Sometimes, an error *may* occur in the script for a reason beyond your control. For example, you might request data from a third party, and their server may not respond. In such cases, it is particularly important to write error-handling code.

In the latter part of the chapter, you will learn how to gracefully check whether something will work, and offer an alternative option if it fails.

A DEBUGGING WORKFLOW

Debugging is about deduction: eliminating potential causes of an error. Here is a workflow for techniques you will meet over the next 20 pages. Try to narrow down where the problem might be, then look for clues.

WHERE IS THE PROBLEM?

First, should try to can narrow down the area where the problem seems to be. In a long script, this is especially important.

1. Look at the error message, it tells you:

- The relevant script that caused the problem.
- The line number where it became a problem for the interpreter. (As you will see, the *cause* of the error may be earlier in a script; but this is the point at which the script could not continue.)
- The type of error (although the underlying cause of the error may be different).

2. Check how far the script is running.
Use tools to write messages to the console to tell how far your script has executed.

3. Use breakpoints where things are going wrong. They let you pause execution and inspect the values that are stored in variables.

If you are stuck on an error, many programmers suggest that you try to describe the situation (talking out loud) to another programmer. Explain what should be happening and where the error appears to be happening. This seems to be an effective way of finding errors in all programming languages. (If nobody else is available, try describing it to yourself.)

WHAT EXACTLY IS THE PROBLEM?

Once you think that you might know the rough area in which your problem is located, you can then try to find the actual line of code that is causing the error.

1. When you have set breakpoints, you can see if the variables around them have the values you would expect them to. If not, look earlier in the script.

2. Break down / break out parts of the code to test smaller pieces of the functionality.

- Write values of variables into the console.
- Call functions from the console to check if they are returning what you would expect them to.
- Check if objects exist and have the methods / properties that you think they do.

3. Check the number of parameters for a function, or the number of items in an array.

And be prepared to repeat the whole process if the above solved one error just to uncover another...

If the problem is hard to find, it is easy to lose track of what you *have* and *have not* tested. Therefore, when you start debugging, keep notes of what you have tested and what the result was. No matter how stressful the circumstances are, if you can, stay calm and methodical, the problem will feel less overwhelming and you will solve it faster.

BROWSER DEV TOOLS & JAVASCRIPT CONSOLE

The JavaScript console will tell you when there is a problem with a script, where to look for the problem, and what kind of issue it seems to be.

These two pages show instructions for opening the console in all of the main browsers (but the rest of this chapter will focus on Chrome and Firefox).

Browser manufacturers occasionally change how to access these tools. If they are not where stated, search the browser help files for "console."

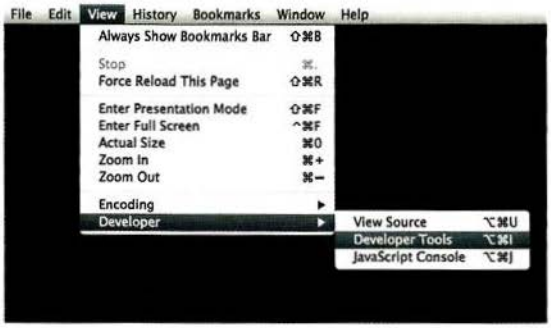
CHROME / OPERA

On a PC, press the F12 key or:

1. Go to the options menu (or three line menu icon)
2. Select ***Tools*** or ***More tools***.
3. Select ***JavaScript Console*** or ***Developer Tools***

On a Mac press Alt + Cmd + J. Or:

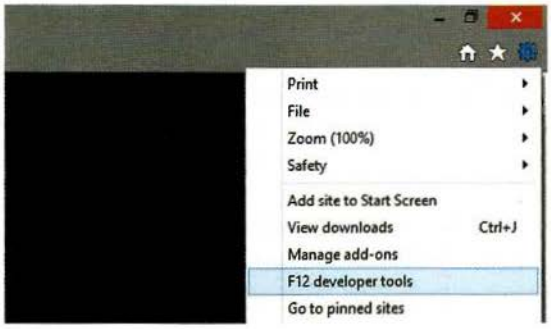
4. Go to the ***View*** menu.
5. Select ***Developer***.
6. Open the ***JavaScript Console*** or ***Developer Tools*** option and select ***Console***.



INTERNET EXPLORER

Press the F12 key or:

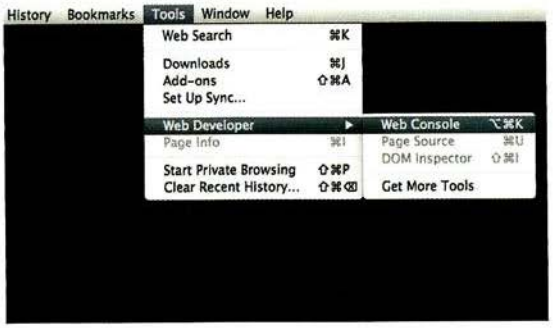
1. Go to the settings menu in the top-right.
2. Select ***developer tools***.



The JavaScript console is just one of several developer tools that are found in all modern browsers.

When you are debugging errors, it can help if you look at the error in more than one browser as they can show you different error messages.

If you open the `errors.html` file from the sample code in your browser, and then open the console, you will see an error is displayed.



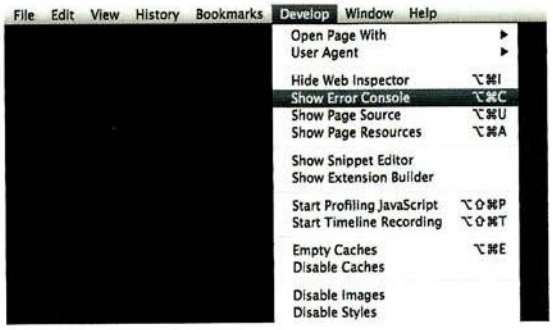
FIREFOX

On a PC, press Ctrl + Shift + K or:

1. Go to the ***Firefox*** menu.
2. Select ***Web Developer***.
3. Open the ***Web Console***.

On a Mac press Alt + Cmd + K. Or:

1. Go to the ***Tools*** menu.
2. Select ***Web Developer***.
3. Open the ***Web Console***.



SAFARI

Press Alt + Cmd + C or:

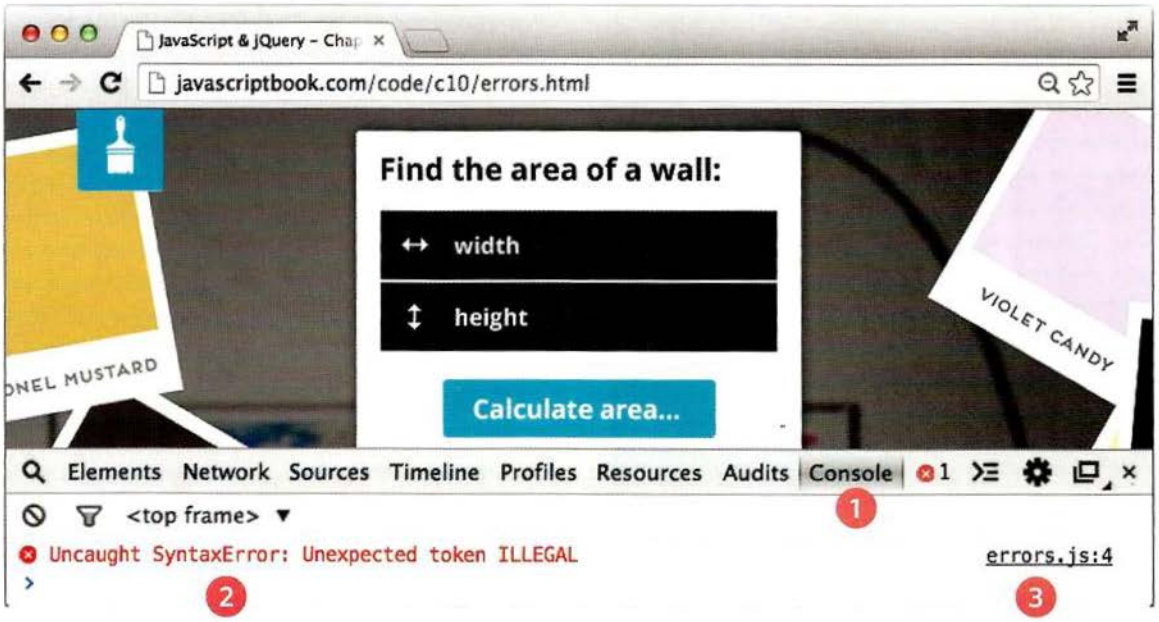
1. Go to the ***Develop*** menu.
2. Select ***Show Error Console***.

If the ***Develop*** menu is not shown:

1. Go to the ***Safari*** menu.
2. Select ***Preferences***.
3. Select ***Advanced***.
4. Check the box that says ***"Show Develop menu in menu bar."***

HOW TO LOOK AT ERRORS IN CHROME

The console will show you when there is an error in your JavaScript. It also displays the line where it became a problem for the interpreter.

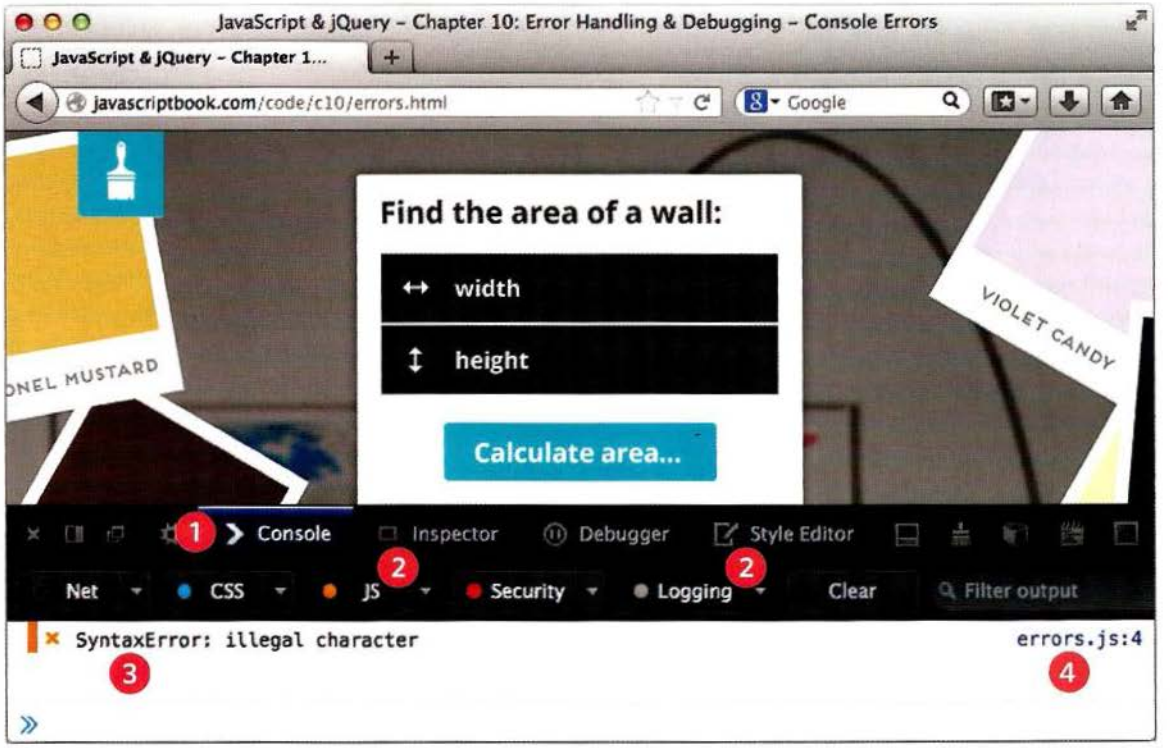


1. The ***Console*** option is selected.
2. The type of error and the error message are shown in red.
3. The file name and the line number are shown on the right-hand side of the console.

Note that the line number does not always indicate where the error is. Rather, it is where the interpreter noticed there was a problem with the code.

If the error stops JavaScript from executing, the console will show only one error – there may be more to troubleshoot once this error is fixed.

HOW TO LOOK AT ERRORS IN FIREFOX



1. The ***Console*** option is selected.
2. Only the ***JavaScript*** and ***Logging*** options need to be turned on. The ***Net***, ***CSS***, and ***Security*** options show other information.
3. The type of error and the error message are shown on the left.
4. On the right-hand side of the console, you can see the name of the JavaScript file and the line number of the error.

Note that when debugging any JavaScript code that has been minified, it will be easier to understand if you expand it first.

TYPING IN THE CONSOLE IN CHROME

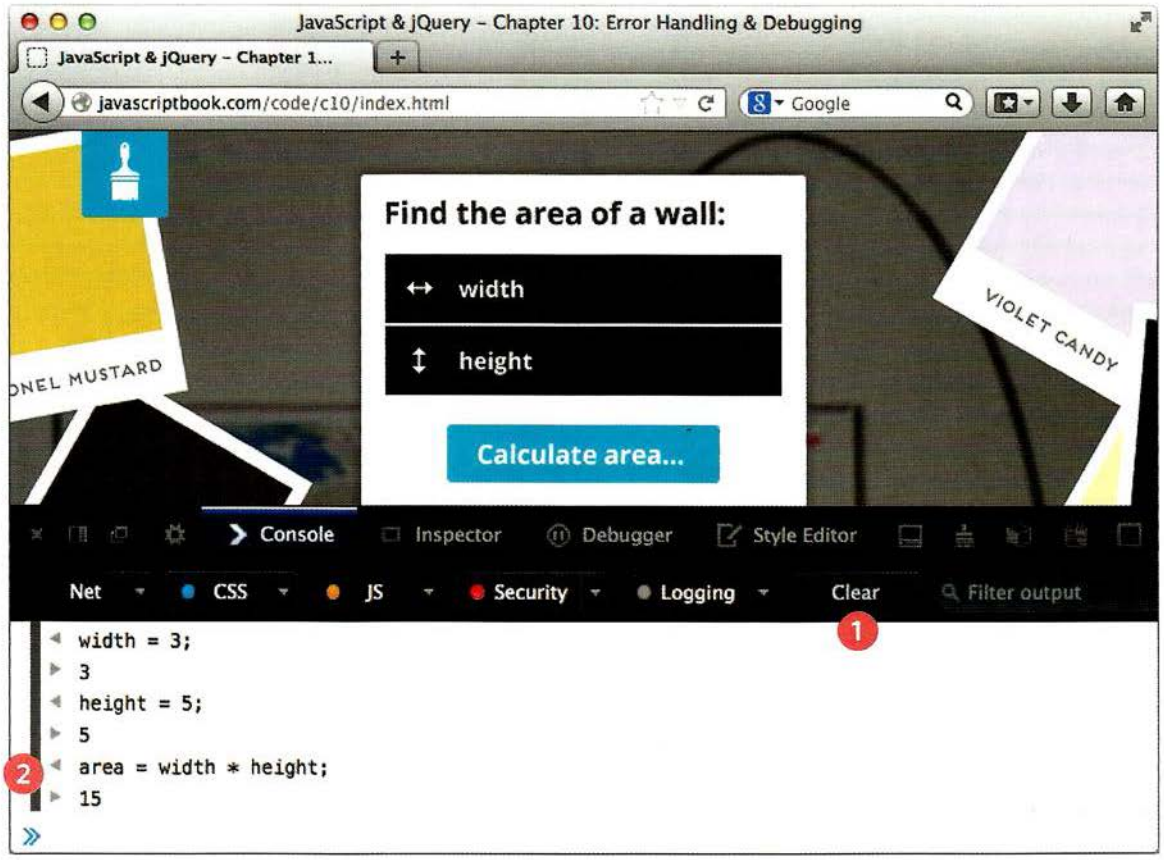
You can also just type code into the console and it will show you a result.

Above, you can see an example of JavaScript being written straight into the console. This is a quick and handy way to test your code.

Each time you write a line, the interpreter may respond. Here, it is writing out the value of each variable that has been created.

Any variable that you create in the console will be remembered until you clear the console.
1. In Chrome, the ***no-entry sign*** is used to clear the console.

TYPING IN THE CONSOLE IN FIREFOX

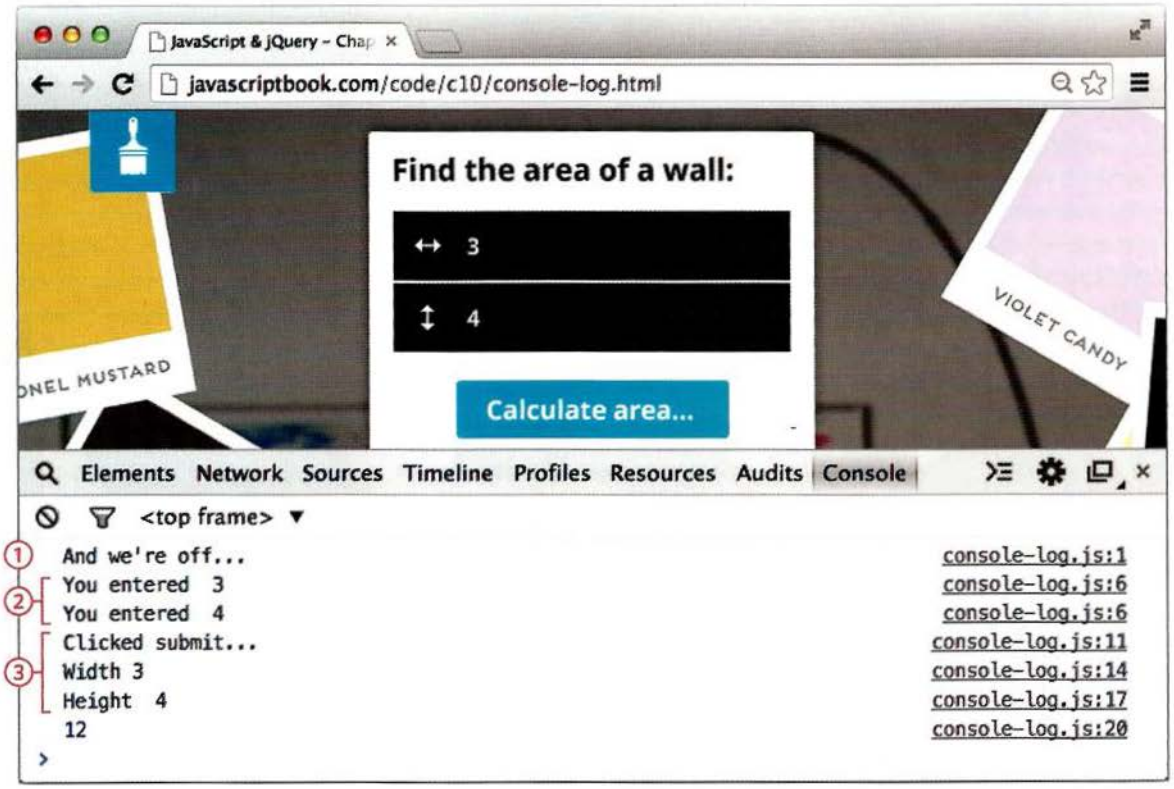


1. In Firefox, the ***Clear*** button will clear the contents of the console. This tells the interpreter that it no longer needs to remember the variables you have created.

2. The left and right arrows show which lines *you* have written, and which are from the interpreter.

WRITING FROM THE SCRIPT TO THE CONSOLE

Browsers that have a console have a `console` object, which has several methods that your script can use to display data in the console. The object is documented in the Console API.



1. The `console.log()` method can write data from a script to the console. If you open `console-log.html`, you will see that a note is written to the console when the page loads.

2. Such notes can tell you how far a script has run and what values it has received. In this example, the `blur` event causes the value entered into a text input to be logged in the console.

3. Writing out variables lets you see what values the interpreter holds for them. In this example, the console will write out the values of each variable when the form is submitted.

LOGGING DATA TO THE CONSOLE

This example shows several uses of the `console.log()` method.

1. The first line is used to indicate the script is running.

2. Next an event handler waits for the user leaving a text input, and logs the value that they entered into that form field.

When the user submits the form, four values are displayed:

3. That the user clicked submit
4. The value in the width input
5. The value in the height input
6. The value of the area variable

They help check that you are getting the values you expect.

The `console.log()` method can write several values to the console at the same time, each separated by a comma, as shown when displaying the height (**5**).

You should always remove this kind of error handling code from your script before you use it on a live site.

JAVASCRIPT

c10/js/console-log.js

```
① console.log('And we\'re off...');                  // Indicates script is running
  var $form, width, height, area;
  $form = $('#calculator');

  $('form input[type="text"]').on('blur', function() { // When input loses focus
②   console.log('You entered ', this.value );          // Write value to console
  });

  $('#calculator').on('submit', function(e) {        // When the user clicks submit
    e.preventDefault();                              // Prevent the form submitting
③   console.log('Clicked submit...');                // Indicate button was clicked

    width = $('#width').val();
④   console.log('Width ' + width);                   // Write width to console

    height = $('#height').val();
⑤   console.log('Height ', height);                  // Write height to console

    area = width * height;
⑥   console.log(area);                               // Write area to console

    $form.append('<p>' + area + '</p>')
  });
```

MORE CONSOLE METHODS

To differentiate between the types of messages you write to the console, you can use three different methods. They use various colors and icons to distinguish them.

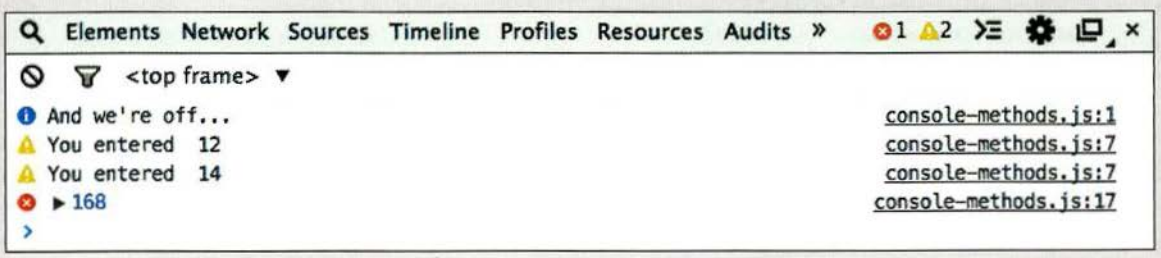
1. `console.info()` can be used for general information
2. `console.warn()` can be used for warnings
3. `console.error()` can be used to hold errors

This technique is particularly helpful to show the nature of the information that you are writing to the screen. (In Firefox, make sure you have the logging option selected.)

c10/js/console-methods.js

JAVASCRIPT

```
① console.info('And we\'re off...');                    // Info: script running

  var $form, width, height, area;
  $form = $('#calculator');

  $('form input[type="text"]').on('blur', function() {  // On blur event
②   console.warn('You entered ', this.value);          // Warn: what was entered
  });

  $('#calculator').on('submit', function(e) {           // When form is submitted
    e.preventDefault();

    width = $('#width').val();
    height = $('#height').val();

    area = width * height;
③   console.error(area);                                // Error: show area

    $form.append('<p class="result">' + area + '</p>');
  });
```



GROUPING MESSAGES

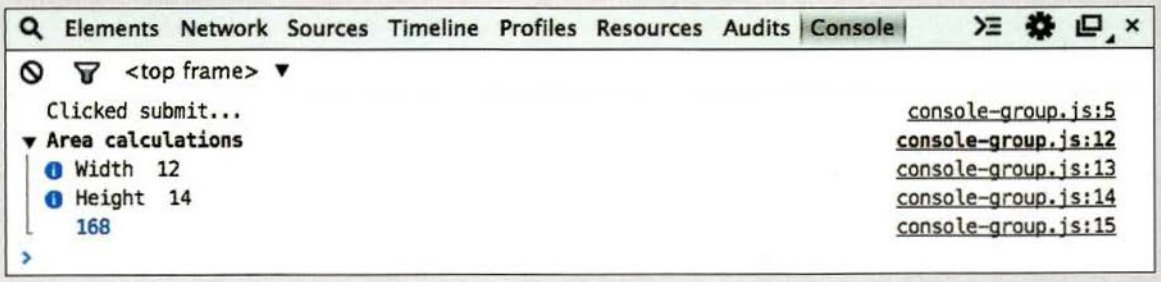
1. If you want to write a set of related data to the console, you can use the `console.group()` method to group the messages together. You can then expand and contract the results. It has one parameter; the name that you want to use for the group of messages. You can then expand and collapse the contents by clicking next to the group's name as shown below.

2. When you have finished writing out the results for the group, to indicate the end of the group the `console.groupEnd()` method is used.

JAVASCRIPT

c10/js/console-group.js

```
var $form = $('#calculator');

$form.on('submit', function(e) {                    // Runs when submit is pressed
  e.preventDefault();
  console.log('Clicked submit...');                 // Show the button was clicked

  var width, height, area;
  width = $('#width').val();
  height = $('#height').val();
  area = width * height;

① console.group('Area calculations');               // Start group
    console.info('Width ', width);                  // Write out the width
    console.info('Height ', height);                // Write out the height
    console.log(area);                              // Write out the area
② console.groupEnd();                               // End group

  $form.append('<p>' + area + '</p>');
});
```



WRITING TABULAR DATA

In browsers that support it, the `console.table()` method lets you output a table showing:

- objects
- arrays that contain other objects or arrays

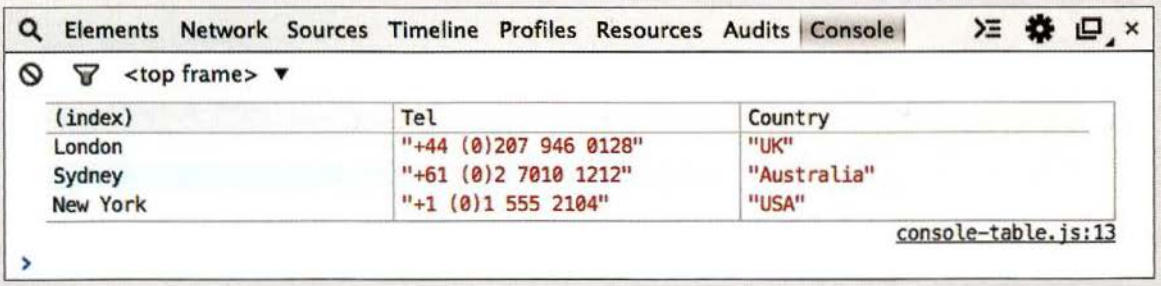
The example below shows data from the `contacts` object. It displays the city, telephone number, and country. It is particularly helpful when the data is coming from a third party.

The screen shot below shows the result in Chrome (it looks the same in Opera). Safari will show expanding panels. At the time of writing Firefox and IE did not support this method.

c10/js/console-table.js

JAVASCRIPT

```
var contacts = {                    // Store contact info in an object literal
  "London": {
    "Tel": "+44 (0)207 946 0128",
    "Country": "UK"},
  "Sydney": {
    "Tel": "+61 (0)2 7010 1212",
    "Country": "Australia"},
  "New York": {
     "Tel": "+1 (0)1 555 2104",
     "Country": "USA"}
}

① console.table(contacts);                          // Write data to console

var city, contactDetails;                           // Declare variables for page
contactDetails = '';                                // Hold details written to page

$.each(contacts, function(city, contacts) {         // Loop through data to
  contactDetails += city + ': ' + contacts.Tel + '<br />';
});
$('h2').after('<p>' + contactDetails + '</p>'); // Add data to the page
```



WRITING ON A CONDITION

Using the `console.assert()` method, you can test if a condition is met, and write to the console only if the expression evaluates to `false`.

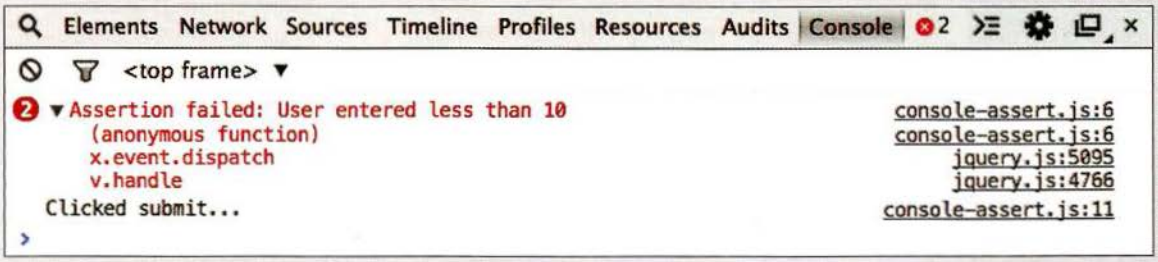
1. Below, when users leave an input, the code checks to see if they entered a value that is 10 or higher. If not, it will write a message to the screen.

2. The second check looks to see if the calculated area is a numeric value. If not, then the user must have entered a value that was not a number.

JAVASCRIPT

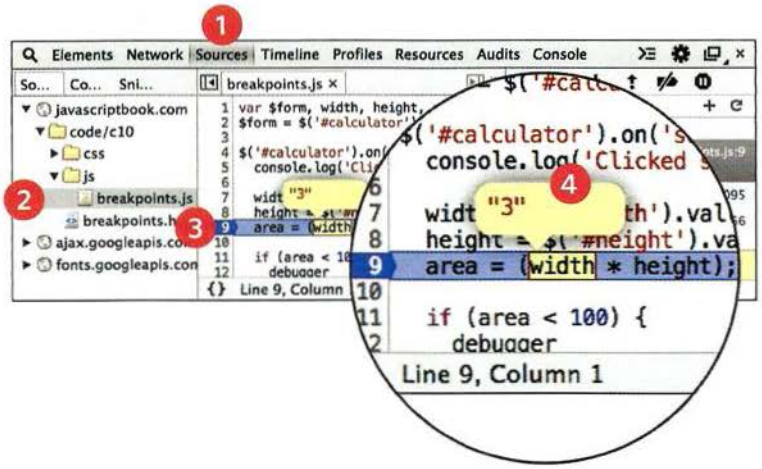
c10/js/console-assert.js

```
var $form, width, height, area;
$form = $('#calculator');

$('form input[type="text"]').on('blur', function() {
  // The message only shows if user has entered number less than 10
① console.assert(this.value > 10, 'User entered less than 10');
});

$('#calculator').on('submit', function(e) {
  e.preventDefault();
  console.log('Clicked submit...');

  width = $('#width').val();
  height = $('#height').val();
  area = width * height;
  // The message only shows if user has not entered a number
② console.assert($.isNumeric(area), 'User entered non-numeric value');

  $form.append('<p>' + area + '</p>');
});
```



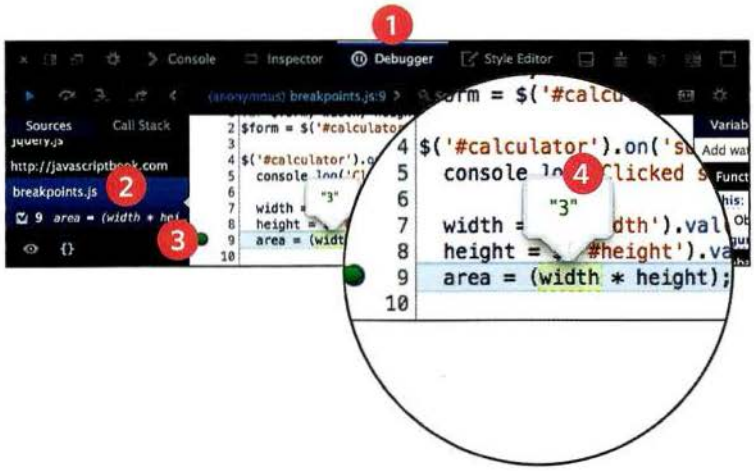
BREAKPOINTS

You can pause the execution of a script on any line using breakpoints. Then you can check the values stored in variables at that point in time.



CHROME

1. Select the ***Sources*** option.
2. Select the script you are working with from the left-hand pane. The code will appear to the right.
3. Find the line number you want to stop on and click on it.
4. When you run the script, it will stop on this line. You can now hover over any variable to see its value at that time in the script's execution.



FIREFOX

1. Select the ***Debugger*** option.
2. Select the script you are working with from the left-hand pane. The code will appear to the right.
3. Find the line number you want to stop on and click on it.
4. When you run the script, it will stop on this line. You can now hover over any variable to see its value at that time in the script's execution.

STEPPING THROUGH CODE

If you set multiple breakpoints, you can step through them one-by-one to see where values change and a problem might occur.

When you have set breakpoints, you will see that the debugger lets you step through the code line by line and see the values of variables as your script progresses.

When you are doing this, if the debugger comes across a function, it will move onto the next line after the function. (It does not move to where the function is defined.) This behavior is sometimes called **stepping over** a function.

If you want to, it is possible to tell the debugger to **step into** a function to see what is happening inside the function.

Chrome and Firefox both have very similar tools for letting you step through the breakpoints.

1. A pause sign shows until the interpreter comes across a breakpoint. When the interpreter stops on a breakpoint, a play-style button is then shown. This lets you tell the interpreter to resume running the code.

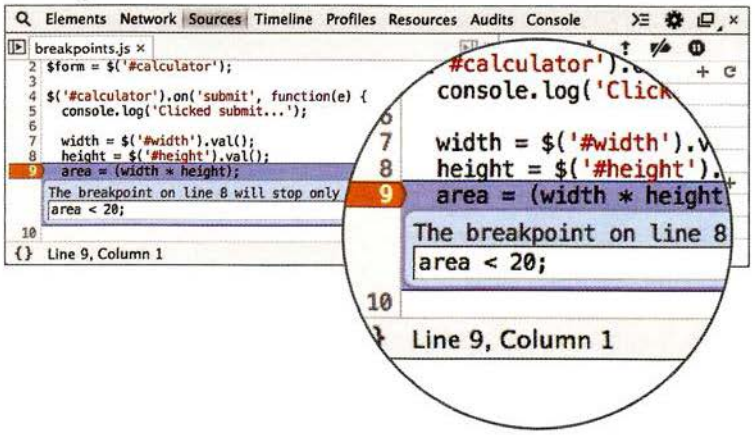
2. Go to the next line of code and **step through** the lines one-by-one (rather than running them as fast as possible).

3. Step into a function call. The debugger will move to the first line in that function.

4. Step out of a function that you stepped into. The remainder of the function will be executed as the debugger moves to its parent function.

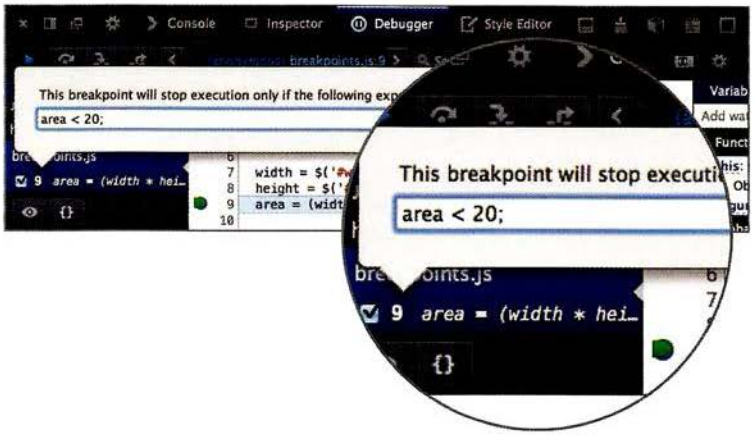
CONDITIONAL BREAKPOINTS

You can indicate that a breakpoint should be triggered only if a condition that you specify is met. The condition can use existing variables.



CHROME

1. Right-click on a line number.
2. Select ***Add Conditional Breakpoint...***
3. Enter a condition into the popup box.
4. When you run the script, it will only stop on this line if the condition is true (e.g., if `area` is less than 20).



FIREFOX

1. Right-click on a line of code.
2. Select ***Add conditional breakpoint***.
3. Enter a condition into the popup box.
4. When you run the script, it will stop on this line only if the condition is true (e.g., if `area` is less than 20).

DEBUGGER KEYWORD

You can create a breakpoint in your code using just the `debugger` keyword. When the developer tools are open, this will automatically create a breakpoint.

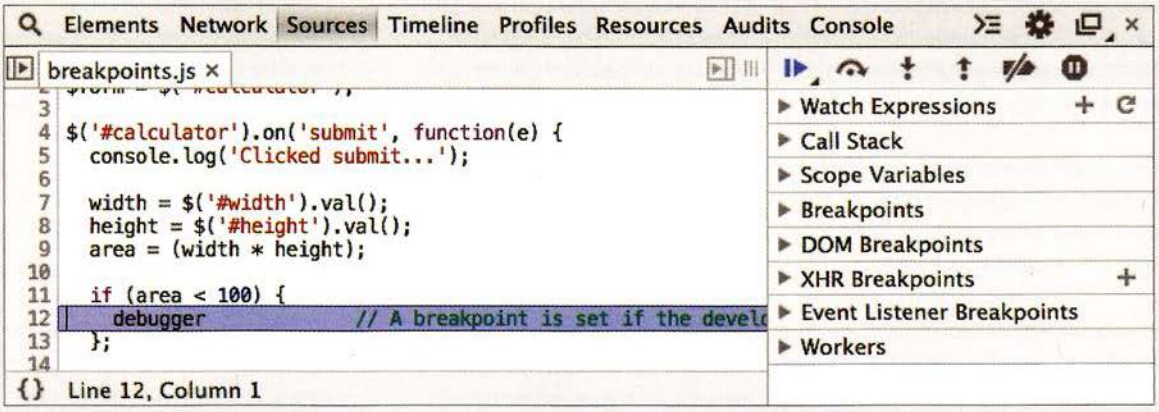
You can also place the `debugger` keyword within a conditional statement so that it only triggers the breakpoint if the condition is met. This is demonstrated in the code below.

It is particularly important to remember to remove these statements before your code goes live as this could stop the page running if a user has developer tools open.

JAVASCRIPT

c10/js/breakpoints.js

```
var $form, width, height, area;
$form = $('#calculator');

$('#calculator').on('submit', function(e) {
  e.preventDefault();
  console.log('Clicked submit...');

  width = $('#width').val();
  height = $('#height').val();
  area = (width * height);

  if (area < 100) {
    debugger;            // A breakpoint is set if the developer tools are open
  }

  $form.append('<p>' + area + '</p>');
});
```



If you have a development server, your debugging code can be placed in conditional statements that check whether it is running on a specific server (and the debugging code only runs if it is on the specified server).

HANDLING EXCEPTIONS

If you know your code might fail, use `try`, `catch`, and `finally`. Each one is given its own code block.

```
try {
  // Try to execute this code
} catch (exception) {
  // If there is an exception, run this code
} finally {
  // This always gets executed
}
```

TRY

First, you specify the code that you think might throw an exception within the `try` block.

If an exception occurs in this section of code, control is automatically passed to the corresponding `catch` block.

The `try` clause must be used in this type of error handling code, and it should always have either a `catch`, `finally`, or both.

If you use a `continue`, `break`, or `return` keyword inside a `try`, it will go to the `finally` option.

CATCH

If the `try` code block throws an exception, `catch` steps in with an alternative set of code.

It has one parameter: the error object. Although it is optional, you are not *handling* the error if you do not catch an error.

The ability to catch an error can be very helpful if there is an issue on a live website.

It lets you tell users that something has gone wrong (rather than not informing them why the site stopped working).

FINALLY

The contents of the `finally` code block will run either way - whether the `try` block succeeded or failed.

It even runs if a `return` keyword is used in the `try` or `catch` block. It is sometimes used to clean up after the previous two clauses.

These methods are similar to the `.done()`, `.fail()`, and `.always()` methods in jQuery.

You can nest checks inside each other (place another `try` inside a `catch`), but be aware that it can affect performance of a script.

TRY, CATCH, FINALLY

This example displays JSON data to the user. But, imagine that the data is coming from a third party and there have been occasional problems with it that could cause the page to fail.

This script checks if the JSON can be parsed using a `try` block before trying to display the information to the users.

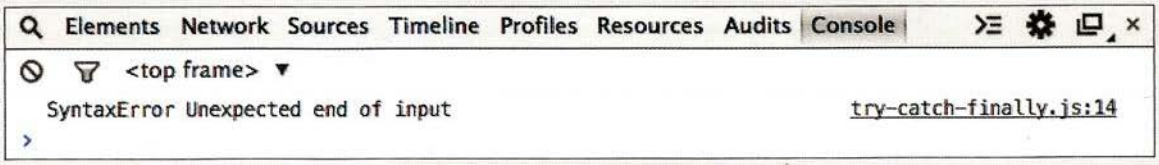
If the `try` statement throws an error (because the data cannot be parsed), the code in the `catch` code block will be run, and the error will not prevent the rest of the script from being executed.

The `catch` statement creates a message using the `name` and `message` properties of the `Error` object.

The error will be logged to the console, and a friendly message will be shown to the users of the site. You could also send the error message to the server using Ajax so that it could be recorded. Either way, the `finally` statement adds a link that allows users to refresh the data they are seeing.

JAVASCRIPT c10/js/try-catch-finally.js

```
response = ' {"deals": [{"title": "Farrow and Ball",... ' // JSON data

if (response) {
    try{
        var dealData = JSON.parse(response);              // Try to parse JSON
        showContent(dealData);                            // Show JSON data
    }catch(e) {
        var errorMessage = e.name + ' ' + e.message;      // Create error msg
        console.log(errorMessage);                        // Show devs msg
        feed.innerHTML = '<em>Sorry, could not load deals'</em>; // Users msg
    } finally {
        var link = document.createElement('a');           // Add refresh link
        link.innerHTML = ' <a href="try-catch-finally.html">reload</a>';
        feed.appendChild(link);
    }
}
```



THROWING ERRORS

If you know something might cause a problem for your script, you can generate your own errors before the interpreter creates them.

To create your own error, you use the following line:

```
throw new Error('message');
```

This creates a new `Error` object (using the default `Error` object). The parameter is the message you want associated with the error. This message should be as descriptive as possible.

Being able to throw an error at the time you know there might be a problem can be better than letting that data cause errors further into the script.

If you are working with data from a third party, you may come across problems such as:

- JSON that contains a formatting error
- Numeric data that occasionally has a non-numeric value
- An error from a remote server
- A set of information with one missing value

Bad data might not cause an error in the script straight away, but it could cause a problem later on. In such cases, it helps to report the problem straight away. It can be much harder to find the source of the problem if the data causes an error in a different part of the script.

For example, if a user enters a string when you expect a number, it might not throw an error immediately.

However, if you know that the application will try to use that value in a mathematical operation at some point in the future, you know that it will cause a problem later on.

If you add a number to a string, it will result in a string. If you use a string in any other mathematical calculations, the result would be `NaN`. In itself, `NaN` is not an error; it is a value that is not a number.

Therefore, if you throw an error when the user enters a value you cannot use, it prevents issues at some other point in the code. You can create an error that explains the problem, before the user gets further into the script.

THROW ERROR FOR NaN

If you try to use a string in a mathematical operation (other than in addition), you do not get an error, you get a special value called `NaN` (not a number).

In this example, a `try` block attempts to calculate the area of a rectangle. If it is given numbers to work with, the code will run. If it does not get numbers, a custom error is thrown and the `catch` block displays the error.

By checking that the results are numeric, the script can fail at a specific point and you can provide a detailed error about what caused the problem (rather than letting it cause a problem later in the script).

JAVASCRIPT c10/js/throw.js

```
var width = 12;                                    // width variable
var height = 'test';                               // height variable

function calculateArea(width, height) {
  try {
    var area = width * height;                     // Try to calculate area
    if (!isNaN(area)) {                            // If it is a number
      return area;                                 // Return the area
    } else {                                       // Otherwise throw an error
      throw new Error('calculateArea() received invalid number');
    }
  } catch(e) {                                     // If there was an error
    console.log(e.name + ' ' + e.message);         // Show error in console
    return 'We were unable to calculate the area.'; // Show users a message
  }
}

// TRY TO SHOW THE AREA ON THE PAGE
document.getElementById('area').innerHTML = calculateArea(width, height);
```

There are two different errors shown: one in the browser window for the users and another in the console for the developers.

This not only catches an error that would not have been thrown otherwise, but it also provides a more descriptive explanation of what caused the error.

Ideally, form validation, which you learn about in Chapter 13, would solve this kind of issue. It is more likely to occur when data comes from a third party.

DEBUGGING TIPS

Here are a selection of practical tips that you can try to use when debugging your scripts.

ANOTHER BROWSER

Some problems are browser-specific. Try the code in another browser to see which ones are causing a problem.

ADD NUMBERS

Write numbers to the console so you can see which the items get logged. It shows how far your code runs before errors stop it.

STRIP IT BACK

Remove parts of code, and strip it down to the minimum you need. You can do this either by removing the code altogether, or by just commenting it out using multi-line comments:

```
/* Anything between these
characters is a comment */
```

EXPLAINING THE CODE

Programmers often report finding a solution to a problem while explaining the code to someone else.

SEARCH

Stack Overflow is a Q+A site for programmers.

Or use a traditional search engine such as Google, Bing, or DuckDuckGo.

CODE PLAYGROUNDS

If you want to ask about problematic code on a forum, in addition to pasting the code into a post, you could add it to a code playground site (such as JSBin.com, `JSFiddle.com`, or `Dabblet.com`) and then post a link to it from the forum.

(Other popular playgrounds include `CSSDeck.com` and `CodePen.com` - but these sites place more emphasis on show and tell.)

VALIDATION TOOLS

There are a number of online validation tools that can help you try to find errors in your code:

JAVASCRIPT

`http://www.jslint.com`
`http://www.jshint.com`

JSON

`http://www.jsonlint.com`

JQUERY

There is a jQuery debugger plugin available for Chrome which can be found in the Chrome web store.

COMMON ERRORS

Here is a list of common errors you might find with your scripts.

GO BACK TO BASICS

JavaScript is case sensitive so check your capitalization.

If you did not use `var` to declare the variable, it will be a global variable, and its value could be overwritten elsewhere (either in your script or by another script that is included in the page).

If you cannot access a variable's value, check if it is out of scope, e.g., declared within a function that you are not within.

Do not use reserved words or dashes in variable names.

Check that your single / double quotes match properly.

Check that you have escaped quotes in variable values.

Check in the HTML that values of your `id` attributes are unique.

MISSED / EXTRA CHARACTERS

Every statement should end in a semicolon.

Check that there are no missing closing braces } or parentheses).

Check that there are no commas inside a `,}` or `,)` by accident.

Always use parentheses to-surround a condition that you are testing.

Check the script is not missing a parameter when calling a function.

`undefined` is not the same as `null`: `null` is for objects, `undefined` is for properties, methods, or variables.

Check that your script has loaded (especially CDN files).

Look for conflicts between different script files.

DATA TYPE ISSUES

Using = rather than == will assign a value to a variable, not check that the values match.

If you are checking whether values match, try to use strict comparison to check datatypes at the same time. (Use === rather than ==.)

Inside a `switch` statement, the values are not loosely typed (so their type will not be coerced).

Once there is a match in a switch statement, all expressions will be executed until the next `break` or `return` statement is executed.

The `replace()` method only replaces the first match. If you want to replace all occurrences, use the global flag.

If you are using the `parseInt()` method, you might need to pass a radix (the number of unique digits including zero used to represent the number).

SUMMARY
ERROR HANDLING & DEBUGGING

- If you understand execution contexts (which have two stages) and stacks, you are more likely to find the error in your code.
- Debugging is the process of finding errors. It involves a process of deduction.
- The console helps narrow down the area in which the error is located, so you can try to find the exact error.
- JavaScript has 7 different types of errors. Each creates its own error object, which can tell you its line number and gives a description of the error.
- If you know that you may get an error, you can handle it gracefully using the `try`, `catch`, `finally` statements. Use them to give your users helpful feedback.

11

CONTENT PANELS

Content panels allow you to showcase extra information within a limited space. In this chapter, you will see several examples of content panels that also give you practical insight into creating your own scripts using jQuery.

In this chapter, you will see how to create many types of content panels: accordions, tabbed panels, modal windows (also known as a lightboxes), a photo viewer, and a responsive slider. Each example of a content panel also demonstrates how to apply the code you have learned throughout the book so far in a practical setting.

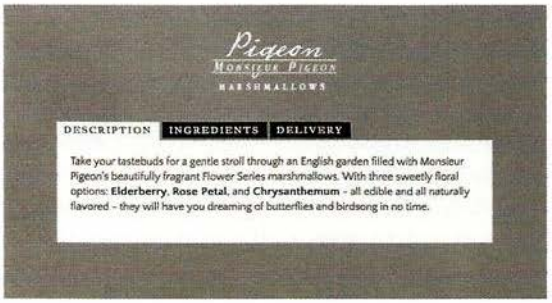
Throughout the chapter, reference will be made to more complex jQuery plugins that extend the functionality of the examples shown here. But the code samples in this chapter also show how it is possible to achieve techniques you will have seen on popular websites in relatively few lines of code (without needing to rely on plugins written by other people).

ACCORDION

An accordion features titles which, when clicked, expand to show a larger panel of content.

TABBED PANEL

Tabs automatically show one panel, but when you click on another tab, the panel is changed.



MODAL WINDOW

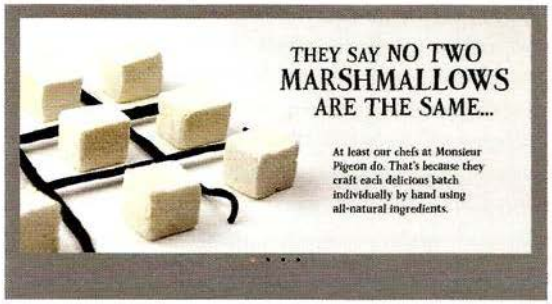
When you click on a link for a modal window (or "lightbox"), a hidden panel will be displayed.

PHOTO VIEWER

Photo viewers display different images within the same space when the user clicks on the thumbnails.

RESPONSIVE SLIDER

The slider allows you to show panels of content that slide into view as the user navigates between them.



CREATING A JQUERY PLUGIN

The final example revisits the accordion (the first example) and turns it into a jQuery plugin.

SEPARATION OF CONCERNS

As you saw in the introduction to this book, it is considered good practice to separate your content (in HTML markup), presentation (in CSS rules), and behaviors (in JavaScript).

In general, your code should reflect that:

- HTML is responsible for structuring content
- CSS is responsible for presentation
- JavaScript is responsible for behavior

Enforcing this separation produces code that is easier to maintain and reuse. While this may already be a familiar concept to you, it's important to remember as it is very easy to mix these concerns in with your JavaScript. As a rule, editing your HTML templates or stylesheets should not necessitate editing your scripts and vice versa.

You can also place event listeners and calls to functions in JavaScript files rather than adding them to the end of an HTML document.

If you need to change the styles associated with an element, rather than having styles written in the JavaScript, you can update the value of the `class` attributes for those elements. In turn, they can trigger new rules from the CSS file that change the appearance of those elements.

When your scripts access the DOM, you can uncouple them from the HTML by using `class` selectors rather than tag selectors.

ACCESSIBILITY & NO JAVASCRIPT

When writing any script, you should think about those who might be using a web page in different situations than you.

ACCESSIBILITY

Whenever a user can interact with an element:

- If it is a link, use <a>
- If it acts like a button, use a button

Both can gain focus, so users can move between them focusable elements using the Tab key (or other non-mouse solution). And although any element can become focusable by setting its `tabindex` attribute, only <a> elements and some input elements fire a `click` event when users press the Enter key on their keyboard (the ARIA `role="button"` attribute will not simulate this event).

NO JAVASCRIPT

This chapter's accordion menu, tabbed panels, and responsive slider all hide some of their content by default. This content would be inaccessible to visitors that do not have JavaScript enabled if we didn't provide alternative styling. One way to solve this is by adding a `class` attribute whose value is `no-js` to the opening <html> tag. This class is then removed by JavaScript (using the `replace()` method of the String object) if JavaScript is enabled.
The `no-js` class can then be used to provide styles targeted to visitors who do not have JavaScript enabled.

HTML — c11/no-js.html

```
<!DOCTYPE html><html class="no-js"> ...
  <body>
    <div class="js-warning">You must enable JavaScript to buy from us</div>
    <!-- Turn off your JavaScript to see the difference -->
    <script src="js/no-js.js"></script>
  </body>
</html>
```

JAVASCRIPT — c11/js/no-js.js

```
var elDocument = document.documentElement;
elDocument.className = elDocument.className.replace(/(^|\s)no-js(\s|$)/, '$1');
```

ACCORDION

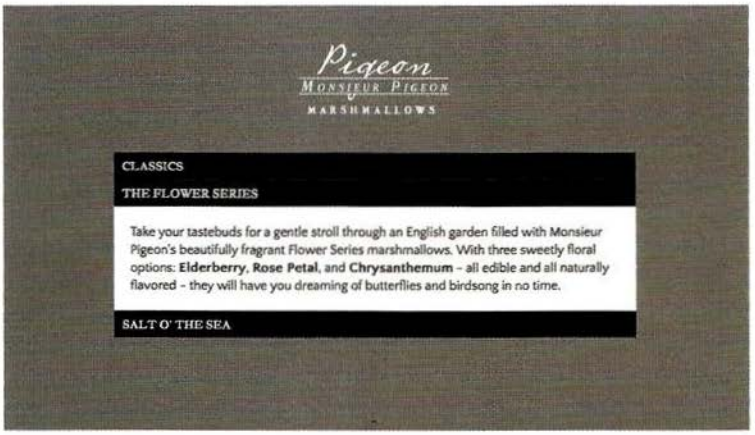
When you click on the title of an accordion, its corresponding panel expands to reveal the content.

An accordion is usually created within an unordered list (in a `<ul>` element). Each `<li>` element is a new item in the accordion. The items contain:

- A visible label (in this example, it is a `<button>`)
- A hidden panel holding the content (a `<div>`)

Clicking a label prompts the associated panel to be shown (or to be hidden if it is in view). To just hide or show a panel, you could change the value of the `class` attribute on the associated panel (triggering a new CSS rule to show or hide it). But, in this case, jQuery will be used to animate the panel into view or hide it.

HTML5 introduces `<details>` and `<summary>` elements to create a similar effect, but (at the time of writing) browser support was not widespread. Therefore, a script like this would still be used for browsers that do not support those features.



Other tabs scripts include liteAccordion and zAccordion. They are also included in jQuery UI and Bootstrap.

ACCORDION WITH ALL PANELS COLLAPSED

ACCORDION WITH SECOND PANEL EXPANDED

When the page loads, CSS rules are used to hide the panels.

Clicking a label prompts the hidden panel that follows it to animate and reveal its full height. This is done using jQuery.

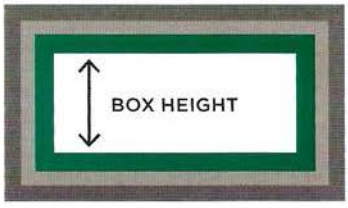
Clicking on the label again would hide the panel.

ANIMATING CONTENT WITH SHOW, HIDE, AND TOGGLE

jQuery's `.show()`, `.hide()`, and `.toggle()` methods animate the showing and hiding of elements.

jQuery calculates the size of the box, including its content, and any margins and padding. This helps if you do not know what content appears in a box.

(To use CSS animation, you would need to calculate the box's height, margin and padding.)



`.toggle()` saves you writing conditional code to tell whether the box is already being shown or not. (If a box is shown, it hides it, and if hidden, it will show it.)

The three methods are all shorthand for the `animate()` method. For example, the `show()` method is shorthand for:

```
$('.accordion-panel')
.animate({
    height: 'show',
    paddingTop: 'show',
    paddingBottom: 'show',
    marginTop: 'show',
    marginBottom: 'show'
});
```

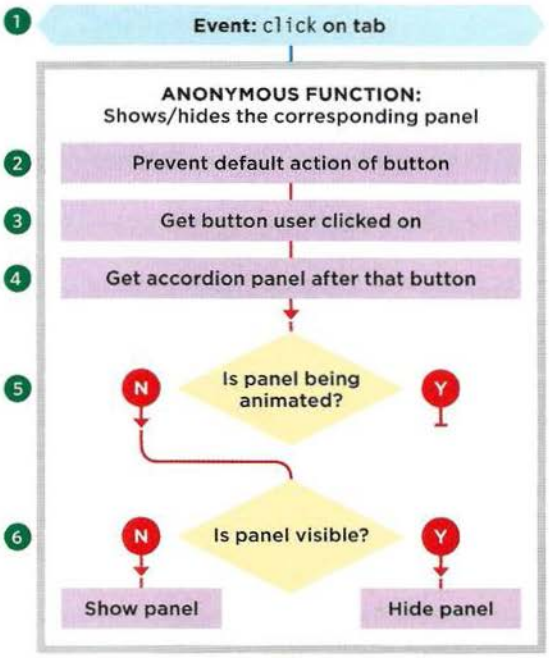
CREATING AN ACCORDION

Below you can see a diagram, rather like a flowchart. These diagrams have two purposes. They help you:

i) Follow the code samples; the numbers on the diagram correspond with the steps on the right, and the script on the right-hand page. Together, the diagrams, steps, and comments in the code should help you understand how each example works.

ii) Learn how to plan a script before coding it.

This is not a "formal" diagram style, but it gives you a visual idea of what is going on with the script. The diagrams show how a collection of small, individual instructions achieve a larger goal, and if you follow the arrows you can see how the data flows around the parts of the script.



Some programmers use Unified Modeling Language or class diagrams - but they have a steeper learning curve, and these flowcharts are here to help you see how the interpreter moves through the script.

Now let's take a look at how the diagram is translated into code. The steps below correspond to the numbers next to the JavaScript code on the right-hand page and the diagram on the left.

1. A jQuery collection is created to hold elements whose `class` attribute has a value of `accordion`. In the HTML you can see that this corresponds to the unordered list element (there could be several lists on the page, each acting as an accordion). An event listener waits for the user to click on one of the buttons whose `class` attribute has a value of `accordion-control`. This triggers an anonymous function.

2. The `preventDefault()` method prevents browsers treating the the button like a submit button. It can be a good idea to use the `preventDefault()` method early in a function so that anyone looking at your code knows that the form element or link does not do what they might expect it to.

3. Another jQuery selection is made using the `this` keyword, which refers to the element the user clicked upon. Three jQuery methods are applied to that jQuery selection holding the element the user clicked on.

4. `.next('.accordion-panel')` selects the next element with a class of `accordion-panel`.

5. `.not(':animated')` checks that it is not in the middle of being animated. (If the user repeatedly clicks the same label, this stops the `.slideToggle()` method from queuing multiple animations.)

6. `.slideToggle()` will show the panel if it is currently hidden and will hide the panel if it is currently visible.

HTML c11/accordion.html

```
<ul class="accordion">
  <li>
    <button class="accordion-control">Classics</button>
    <div class="accordion-panel">Panel content goes here...</div>
  </li>
  <li>
    <button class="accordion-control">The Flower Series</button>
    <div class="accordion-panel">Panel content goes here...</div>
  </li>
  <li>
    <button class="accordion-control">Salt O' the Sea</button>
    <div class="accordion-panel">Panel content goes here...</div>
  </li>
</ul>
```

CSS c11/css/accordion.css

```
.accordion-panel {
  display: none;}
```

JAVASCRIPT c11/js/accordion.js

```
① $('.accordion').on('click', '.accordion-control', function(e){ // When clicked
②   e.preventDefault();                    // Prevent default action of button
③   $(this)                                // Get the element the user clicked on
④     .next('.accordion-panel')            // Select following panel
⑤     .not(':animated')                    // If it is not currently animating
⑥     .slideToggle();                      // Use slide toggle to show or hide it
});
```

Note how steps 4, 5, and 6 are chained off the same jQuery selection.
You saw a screenshot of the accordion example on p492, at the start of this section.

TABBED PANEL

When you click on one of the tabs, its corresponding panel is shown. Tabbed panels look a little like index cards.

You should be able to see all of the tabs, but:

- Only one tab should look *active*.
- Only the panel that corresponds to the active tab should be shown (all other panels should be hidden).

The tabs are typically created using an unordered list. Each `<li>` element represents a tab and within each tab is a link.

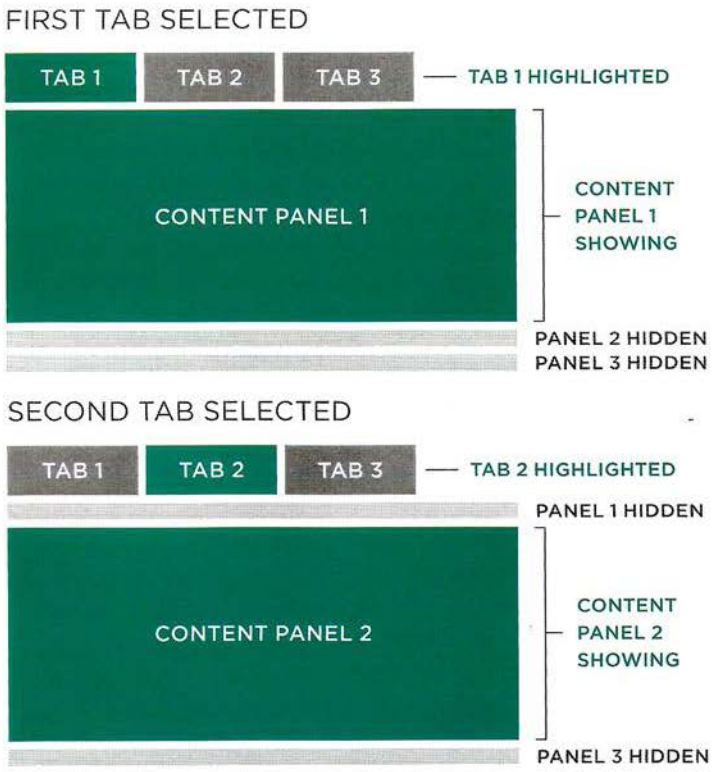
The panels follow the unordered list that holds the tabs, and each panel is stored in a `<div>`.

To associate the tab to the panel:

- The link in the tab, like all links, has an `href` attribute.
- The panel has an `id` attribute.

Both attributes share the same value. (This is the same principle as creating a link to another location within an HTML page.)

Other tabs scripts include Tabslet and Tabulous. They are also included in jQuery UI and Bootstrap.



When the page loads, CSS is used to make the tabs sit next to each other and to indicate which one is considered active.

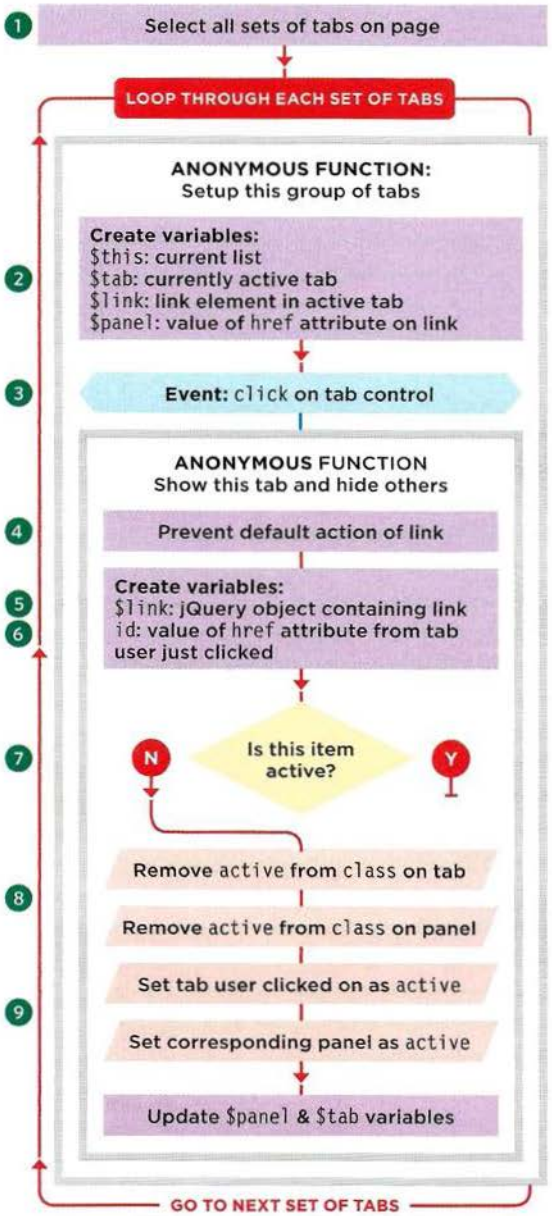
CSS also hides the panels, except for the one that corresponds with the active tab.

When the user clicks on the link inside a tab, the script uses jQuery to get the value of the `href` attribute from the link. This corresponds to the `id` attribute on the panel that should be shown.

The script then updates the values in the `class` attribute on that tab and panel, adding a value of `active`. It also removes that value from the tab and panel that had previously been active.

If the user does not have JavaScript enabled, the link in the tab takes the user to the appropriate part of the page.

CREATING TAB PANELS



The flowchart shows the steps that are involved in creating tabs when they are found in the HTML. Below, you can see how these steps can be translated into code:

1. A jQuery selection picks all sets of tabs within the page. The `.each()` method calls an anonymous function that is run for each set of tabs (like a loop). The code in the anonymous function deals with one set of tabs at a time, and the steps would be repeated for each set of tabs on the page.

2. Four variables hold details of the active tab:

i) `$this` holds the current set of tabs.

ii) `$tab` holds the currently active tab. The `.find()` method selects the active tab.

iii) `$link` holds the <a> element within that tab.

iv) `$panel` holds the value of the `href` attribute for the active tab (this variable will be used to hide the panel if the user selects a different one).

3. An event listener is set up to check for when the user clicks on any tab within that list. When they do, it runs another anonymous function.

4. `e.preventDefault()` prevents the link that users clicked upon taking them to that page.

5. Creates a variable called `$link` to hold the current link inside a jQuery object.

6. Creates a variable called `id` to hold the value of the `href` attribute from the tab that was clicked. It is called `id` because it is used to select the matching content panel (using its `id` attribute).

7. An `if` statement checks whether the `id` variable contains a value, and the current item is **not** active. If both conditions are met:

8. The previously active tab and panel have the `class` of `active` removed (which deactivates the tab and hides the panel).

9. The tab that was clicked on and its corresponding panel both have `active` added to their `class` attributes (which makes the tab look active and displays its corresponding panel, which was hidden). At the same time, references to these elements are stored in the `$panel` and `$tab` variables.

HTML

c11/tabs.html

```
<ul class="tab-list">
  <li class="active"><a class="tab-control" href="#tab-1">Description</a></li>
  <li><a class="tab-control" href="#tab-2">Ingredients</a></li>
  <li><a class="tab-control" href="#tab-3">Delivery</a></li>
</ul>
<div class="tab-panel active" id="tab-1">Content 1...</div>
<div class="tab-panel" id="tab-2">Content 2...</div>
<div class="tab-panel" id="tab-3">Content 3...</div>
```

CSS

c11/css/tabs.css

```
.tab-panel {
  display: none;}
.tab-panel.active {
  display: block;}
```

JAVASCRIPT

c11/js/tabs.js

```
① $('.tab-list').each(function(){                    // Find lists of tabs
    var $this     = $(this);                         // Store this list
②   var $tab      = $this.find('li.active');         // Get the active list item
    var $link     = $tab.find('a');                  // Get link from active tab
    var $panel    = $($link.attr('href'));           // Get active panel

③   $this.on('click', '.tab-control', function(e) { // When click on a tab
④     e.preventDefault();                            // Prevent link behavior
⑤     var $link = $(this);                           // Store the current link
⑥     var id    = this.hash;                         // Get href of clicked tab

⑦     if (id && !$link.is('.active')) {              // If not currently active
⑧       $panel.removeClass('active');                // Make panel inactive
        $tab.removeClass('active');                  // Make tab inactive

⑨       $panel = $(id).addClass('active');           // Make new panel active
        $tab   = $link.parent().addClass('active'); // Make new tab active
      }
    });
});
```

MODAL WINDOW

A modal window is any type of content that appears "in front of" the rest of the page's content. It must be "closed" before the rest of the page can be interacted with.

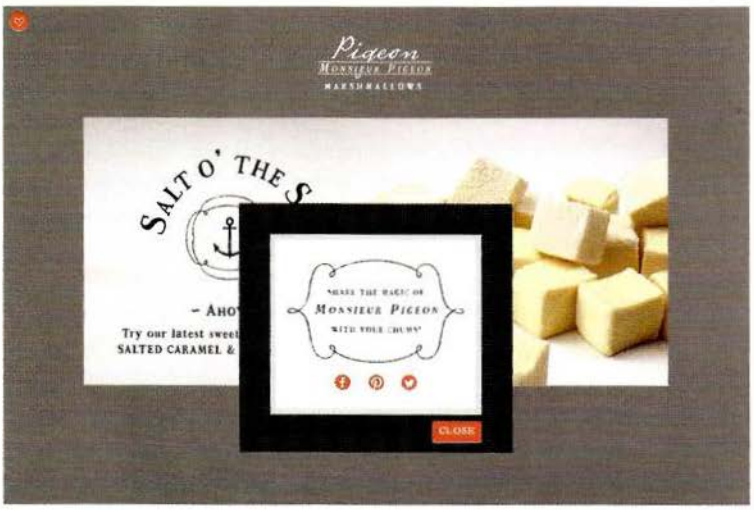
In this example, a modal window is created when the user clicks on the heart button in the top left-hand corner of the page.

The modal window opens in the center of the page, allowing users to share the page on social networks.

The content for the modal window will typically sit within the page, but it is hidden when the page loads using CSS.

JavaScript then takes that content and displays it inside `<div>` elements that create the modal window on top of the existing page.

Sometimes modal windows will dim out the rest of the page behind them. They can be designed to either appear automatically when the page has finished loading or they can be triggered by the user interacting with the page.



Other examples of modal window scripts include Colorbox (by Jack L. Moore), Lightbox 2 (by Lokesh Dhakar), and Fancybox (by Fancy Apps). They are also included in jQuery UI and Bootstrap.

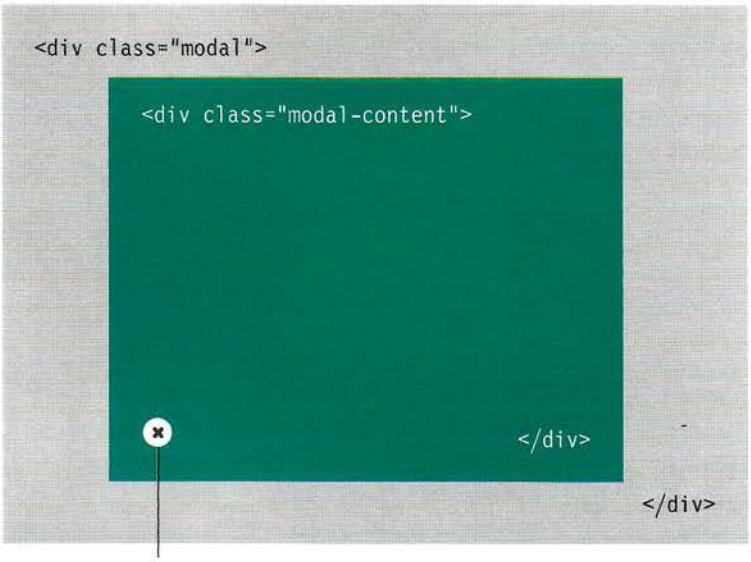
A **design pattern** is a term programmers use to describe a common approach to solving a range of programming tasks.

This script uses the **module pattern**. It is a popular way to write code that contains both **public** and **private** logic.

Once the script has been included in the page, other scripts can use its public methods: `open()`, `close()`, or `center()`. But users do not need to access the variables that create the HTML, so they remain private (on p505 the private code is shown on green).

Using modules to build parts of an application has benefits:

- It helps organize your code.
- You can test and reuse the individual parts of the app.
- It creates scope, preventing variable /method names clashing with other scripts.



This modal window script creates an object (called `modal`), which, in turn, provides three new methods you can use to create modal windows:

`open()` opens a modal window
`close()` closes the window
`center()` centers it on the page

Another script would be used to call the `open()` method and specify what content should appear in the modal window.

Users of this script only need to know how the `open()` method works because:

- `close()` is called by an event listener when the user clicks on the close button.
- `center()` is called by the `open()` method and also by an event listener if the user resizes the window.

When you call the `open()` method, you specify the content that you want the modal window to contain as a parameter (you can also specify its width and height if you want).

In the diagram, you can see that the script adds the content to the page inside `<div>` elements.

`div.modal` acts as a frame around the modal window.

`div.modal-content` acts as a container for the content being added to the page.

`button.modal-close` allows the user to close the modal window.

CREATING MODALS

The modal script needs to do two things:

1. Create the HTML for the modal window
2. Return the `modal` object itself, which consists of the `open()`, `close()`, and `center()` methods

Including the script in the HTML page does not have any visible effect (rather like including jQuery in your page does not affect the appearance of the page).

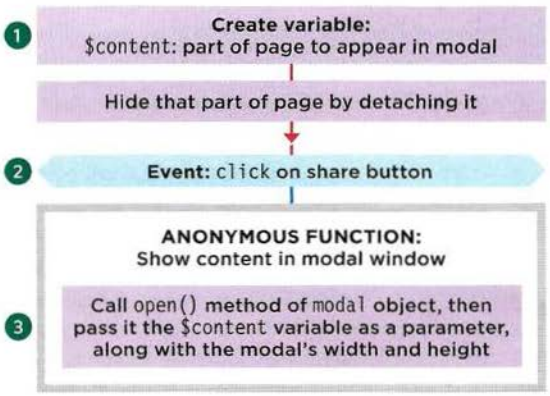
But it does allow any other script you write to use the functionality of the `modal` object and call its `open()` method to create a modal window (just like including jQuery script includes the `jQuery` object in your page and allows you to use its methods).

This means that people who use the script only need to know how to call the `open()` method and tell it what they want to appear in the modal window.

In the example on the right, the modal window is called by a script called `modal-init.js`. You will see how to create the `modal` object and its methods on the next double page spread, but for now consider that including this script is the equivalent of adding the following to your own script. It creates an object called `modal` and adds three methods to the object:

```
var modal = {
  center: function() {
    // Code for center() goes here
  },
  open: function(settings) {
    // Code for open() goes here
  },
  close: function() {
    // Code for close() goes here
  }
};
```

The `modal-init.js` file removes the share content from the HTML page. It then adds an event handler to call the `modal` object's `open()` method to open a modal window containing the content it just removed from the page. `init` is short for initialize and is commonly used in the name of files and functions that set up a page or other part of a script.



1. First the script gets the contents of the element that has an `id` attribute whose value is `share-options`. Note how the jQuery `.detach()` method removes this content from the page.
2. Next an event handler is set to respond to when the user clicks on the share button. When they do, an anonymous function is run.
3. The anonymous function uses the `open()` method of the `modal` object. It takes parameters in the form of an object literal:

- `content`: the content to be shown in the modal window. Here it is the content of the element whose `id` attribute has a value of `share-options`.
- `width`: the width of the modal window.
- `height`: the height of the modal window.

Step 1 uses the `.detach()` method because it keeps the elements and event handlers in memory so they can be used again later. jQuery also has a `.remove()` method but it removes the items completely.

USING THE MODAL SCRIPT

HTML

c11/modal-window.html

```
①     <div id="share-options">
        <!-- This is where the message and sharing buttons go -->
      </div>
      <script src="js/jquery.js"></script>
②     <script src="js/modal-window.js"></script>
③     <script src="js/modal-init.js"></script>
    </body>
  </html>
```

In the HTML above, you should note three things:

1. A `<div>` that contains the sharing options.
2. A link to the script that creates the `modal` object (`modal-window.js`).
3. A link to the script that will open a modal window using the `modal` object (`modal-init.js`), using it to display the sharing options.

The `modal-init.js` file below opens the modal window. Note how the `open()` method is passed three pieces of information in JSON format:

i) `content` for modal (required)
ii) `width` of modal (optional - overrides default)
iii) `height` of modal (optional - overrides default)

JAVASCRIPT

c11/js/modal-init.js

```
(function(){
①   var $content = $('#share-options').detach();   // Remove modal from page

②   $('#share').on('click', function() {           // Click handler to open modal
③     modal.open({content: $content, width:340, height:300});
    });                          ⓘ        ⓘⓘ       ⓘⓘⓘ
}());
```

The `z-index` of the modal window must be very high so that it appears on top of any other content.

These styles ensure the modal window sits on top of the page (there are more styles in the full example).

CSS

c11/css/modal-window.css

```
.modal {
  position: absolute;
  z-index: 1000;}
```