SWITCH STATEMENTS

A **switch** statement starts with a variable called the **switch value**. Each case indicates a possible value for this variable and the code that should run if the variable matches that value.

```
switch (level) {

  case 'One':
    title = 'Level 1';
    break;

  case 'Two':
    title = 'Level 2';
    break;

  case 'Three':
    title = 'Level 3';
    break;

  default:
    title = 'Test';
    break;

}
```

Here, the variable named **level** is the switch value. If the value of the **level** variable is the string **One**, then the code for the first case is executed. If it is **Two**, the second case is executed. If it is **Three**, the third case is executed. If it is none of these, the code for the **default** case is executed.

The entire statement lives in one code block (set of curly braces), and a colon separates the option from the statements that are to be run if the case matches the switch value.

At the end of each case is the **break** keyword. It tells the JavaScript interpreter that it has finished with this **switch** statement and to proceed to run any subsequent code that appears after it.

IF... ELSE

- There is no need to provide an **else** option. (You can just use an **if** statement.)
- With a series of **if** statements, they are all checked *even if* a match has been found (so it performs more slowly than **switch**).

VS.

SWITCH

- You have a **default** option that is run if none of the cases match.
- If a match is found, that code is run; then the **break** statement stops the rest of the **switch** statement running (providing better performance than multiple **if** statements).

USING SWITCH STATEMENTS

JAVASCRIPT c04/js/switch-statement.js

```
var msg;        // Message
var level = 2;  // Level

// Determine message based on level
switch (level) {
case 1:
    msg = 'Good luck on the first test';
    break;

case 2:
    msg = 'Second of three - keep going!';
    break;

case 3:
    msg = 'Final round, almost there!';
    break;

default:
    msg = 'Good luck!';
    break;
}

var el = document.getElementById('answer');
el.textContent = msg;
```

RESULT

In this example, the purpose of the `switch` statement is to present the user with a different message depending on which level they are at. The message is stored in a variable called `msg`.

The variable called `level` contains a number indicating which level the user is on. This is then used as the switch value. (The switch value could also be an expression.)

In the following code block (inside the curly braces), there are three options for what the value of the `level` variable might be: the numbers 1, 2, or 3.

If the value of the `level` variable is the number 1, the value of the `msg` variable is set to `'Good luck on the first test'`.

If the value is 2, it will read: `'Second of three - keep going!'`

If the value is 3, the message will read: `'Final round, almost there!'`

If no match is found, then the value of the `msg` variable is set to `'Good luck!'`

Each case ends with the `break` keyword which will tell the JavaScript interpreter to skip the rest of this code block and continue onto the next.

TYPE COERCION & WEAK TYPING

If you use a data type JavaScript did not expect, it tries to make sense of the operation rather than report an error.

JavaScript can convert data types behind the scenes to complete an operation. This is known as **type coercion**. For example, a string `'1'` could be converted to a number 1 in the following expression: (`'1'` > 0). As a result, the above expression would evaluate to `true`.

JavaScript is said to use **weak typing** because the data type for a value can change. Some other languages require that you specify what data type each variable will be. They are said to use **strong typing**.

Type coercion can lead to unexpected values in your code (and also cause errors). Therefore, when checking if two values are equal, it is considered better to use strict equals operators `===` and `!==` rather than `==` and `!=` as these strict operators check that the value and data types match.

DATA TYPE	PURPOSE
`string`	Text
`number`	Number
`Boolean`	`true` or `false`
`null`	Empty value
`undefined`	Variable has been declared but not yet assigned a value

`NaN` is a value that is counted as a number. You may see it when a number is expected, but is not returned, e.g., `('ten'/2)` results in `NaN`.

TRUTHY & FALSY VALUES

Due to type coercion, every value in JavaScript can be treated as if it were true or false; and this has some interesting side effects.

FALSY VALUES

VALUE	DESCRIPTION
`var highScore = false;`	The traditional Boolean `false`
`var highScore = 0;`	The number zero
`var highScore = '';`	`NaN` (Not a Number)
`var highScore = 10/'score';`	Empty value
`var highScore;`	A variable with no value assigned to it

Almost everything else evaluates to truthy...

TRUTHY VALUES

VALUE	DESCRIPTION
`var highScore = true;`	The traditional Boolean `true`
`var highScore = 1;`	Numbers other than zero
`var highScore = 'carrot';`	Strings with content
`var highScore = 10/5;`	Number calculations
`var highScore = 'true';`	`true` written as a string
`var highScore = '0';`	Zero written as a string
`var highScore = 'false';`	`false` written as a string

Falsy values are treated *as if* they are `false`. The table to the left shows a `highScore` variable with a series of values, all of which are falsy.

Falsy values can also be treated as the number 0.

Truthy values are treated *as if* they are `true`. Almost everything that is not in the falsy table can be treated as if it were true.

Truthy values can also be treated as the number 1.

In addition, the presence of an object or an array is usually considered truthy, too. This is commonly used when checking for the presence of an element in a page.

The next page will explain more about why these concepts are important.

CHECKING EQUALITY & EXISTENCE

Because the presence of an object or array can be considered truthy, it is often used to check for the existence of an element within a page.

A **unary operator** returns a result with just one operand. Here you can see an `if` statement checking for the presence of an element. If the element is found, the result is truthy, so the first set of code is run. If it is not found, the second set is run instead.

```
if (document.getElementById('header')) {
  // Found: do something
} else {
  // Not found: do something else
}
```

Those new to JavaScript often think the following would do the same:

```
if (document.getElementById('header') == true)
```

but `document.getElementById('header')` would return an object which is a truthy value but it is *not* equal to a Boolean value of `true`.

Because of type coercion, the strict equality operators `===` and `!==` result in fewer unexpected values than `==` and `!=` do.

If you use `==` the following values can be considered equal: `false`, `0`, and `''` (empty string). However, they are not equivalent when using the strict operators.

EXPRESSION	RESULT
`(false == 0)`	`true`
`(false === 0)`	`false`
`(false == '')`	`true`
`(false === '')`	`false`
`(0 == '')`	`true`
`(0 === '')`	`false`

Although `null` and `undefined` are both falsy, they are not equal to anything other than themselves. Again, they are not equivalent when using strict operators.

EXPRESSION	RESULT
`(undefined == null)`	`true`
`(null == false)`	`false`
`(undefined == false)`	`false`
`(null == 0)`	`false`
`(undefined == 0)`	`false`
`(undefined === null)`	`false`

Although `NaN` is considered falsy, it is not equivalent to anything; it is not even equivalent to itself (since `NaN` is an undefinable number, two cannot be equal).

EXPRESSION	RESULT
`(Nan == null)`	`false`
`(NaN == NaN)`	`false`

SHORT CIRCUIT VALUES

Logical operators are processed left to right. They short-circuit (stop) as soon as they have a result – but they return the value that stopped the processing (not necessarily `true` or `false`).

On line 1, the variable `artist` is given a value of `Rembrandt`.
On line 2, if the variable `artist` has a value, then `artistA` will be given the same value as `artist` (because a non-empty string is truthy).

```
var artist = 'Rembrandt';
var artistA = (artist || 'Unknown');
```

If the string is empty (see below), `artistA` becomes a string `'Unknown'`.

```
var artist = '';
var artistA = (artist || 'Unknown');
```

You could even create an empty object if artist does not have a value:

```
var artist = '';
var artistA = (artist || {});
```

Logical operators will not always return `true` or `false`, because:

- They return the value that stopped processing.
- That value might have been treated as truthy or falsy although it was not a Boolean.

Programmers use this creatively (for example, to set values for variables or even create objects).

Here are three values. If any one of them is considered truthy, the code inside the `if` statement will execute. When the script encounters `valueB` in the logical operator, it will short circuit because the number 1 is considered truthy and the subsequent code block is executed.

```
valueA = 0;
valueB = 1;
valueC = 2;

if (valueA || valueB || valueC) {
  // Do something here
}
```

This technique could also be used to check for the existence of elements within a page, as shown on p168.

As soon as a truthy value is found, the remaining options are not checked. Therefore, experienced programmers often:

- Put the code most likely to return true *first* in OR operations, and false answers first in AND operations.
- Place the options requiring the most processing power last, just in case another value returns `true` and they do not need to be run.

LOOPS

Loops check a condition. If it returns **true**, a code block will run. Then the condition will be checked again and if it still returns **true**, the code block will run again. It repeats until the condition returns **false**. There are three common types of loops:

FOR

If you need to run code a specific number of times, use a **for** loop. (It is the most common loop.) In a **for** loop, the condition is usually a counter which is used to tell how many times the loop should run.

WHILE

If you do not know how many times the code should run, you can use a **while** loop. Here the condition can be something other than a counter, and the code will continue to loop for as long as the condition is **true**.

DO WHILE

The **do...while** loop is very similar to the **while** loop, but has one key difference: it will always run the statements inside the curly braces at least once, even if the condition evaluates to **false**.

KEYWORD

CONDITION (COUNTER)

OPENING CURLY BRACE

```
for (var i = 0; i < 10; i++) {
    document.write(i);
}
```

CODE TO EXECUTE DURING LOOP

CLOSING CURLY BRACE

This is a **for** loop. The condition is a counter that counts to ten. The result would write "0123456789" to the page.

If the variable **i** is less than ten, the code inside the curly braces is executed. Then the counter is incremented.

The condition is checked again, if **i** is less than ten it runs again. The next three pages show how this loop works in greater detail.

LOOP COUNTERS

A **for** loop uses a counter as a condition.
This instructs the code to run a specified number of times.
Here you can see the condition is made up of three statements:

INITIALIZATION

Create a variable and set it to **0**. This variable is commonly called **i**, and it acts as the counter.

```
var i = 0;
```

The variable is only created the first time the loop is run. (You may also see the variable called **index**, rather than just **i**.)

You will sometimes see this variable declared before the condition. The following is the same and it is mainly a preference of the coder.

```
var i;
for (i = 0; i < 10; i++) {
  // Code goes here
}
```

CONDITION

The loop should continue to run until the counter reaches a specified number.

```
i < 10;
```

The value of **i** was initially set to **0**, so in this case the loop will run 10 times before stopping.

The condition may also use a variable that holds a number. If a variable called **rounds** held the number of rounds in a test and the loop ran once for each round, the condition would be:

```
var rounds = 3;
i < (rounds);
```

UPDATE

Every time the loop has run the statements in the curly braces, it adds one to the counter.

```
i++
```

One is added to the counter using the increment (++) operator.

Another way of reading this is that it says, "Take the variable **i**, and add one using the **++** operator."

It is also possible for loops to count downwards using the decrement operator (--).

LOOPING

The first time the loop is run, the variable i (the counter) is assigned a value of zero.

Every time the loop is run, the condition is checked. Is the variable i less than 10?

Then the code inside the loop (the statements between the curly brackets) is run.

```
for (var i = 0; i < 10; i++) {
    document.write(i);
}
```

The variable **i** can be used inside the loop. Here it is used to write a number to the page.

When the statements have finished, the variable **i** is incremented by 1.

When the condition is no longer true, the loop ends. The script moves to the next line of code.

KEY LOOP CONCEPTS

Here are three points to consider when you are working with loops. Each is illustrated in examples on the following three pages.

KEYWORDS

You will commonly see these two keywords used with loops:

`break`

This keyword causes the termination of the loop and tells the interpreter to go onto the next statement of code outside of the loop. (You may also see it used in functions.)

`continue`

This keyword tells the interpreter to continue with the current iteration, and then check the condition again. (If it is `true`, the code runs again.)

LOOPS & ARRAYS

Loops are very helpful when dealing with arrays if you want to run the same code for each item in the array.

For example, you might want to write the value of each item stored in an array into the page.

You may not know how many items will be in an array when writing a script, but, when the code runs, it can check the total number of items in a loop. That figure can then be used in the counter to control how many times a set of statements is run.

Once the loop has run the right number of times, the loop stops.

PERFORMANCE ISSUES

It is important to remember that when a browser comes across JavaScript, it will stop doing anything else until it has processed that script.

If your loop is dealing with only a small number of items, this will not be an issue. If, however, your loop contains a lot of items, it can make the page slower to load.

If the condition never returns `false`, you get what is commonly referred to as an **infinite loop**. The code will not stop running until your browser runs out of memory (breaking your script).

Any variable you can define outside of the loop and that does not change *within* the loop should be defined outside of it. If it were declared inside the loop, it would be recalculated every time the loop ran, needlessly using resources.

USING FOR LOOPS

JAVASCRIPT

c04/js/for-loop.js

```
var scores = [24, 32, 17];        // Array of scores
var arrayLength = scores.length;// Items in array
var roundNumber = 0;              // Current round
var msg = '';                     // Message
var i;                            // Counter

// Loop through the items in the array
for (i = 0; i < arrayLength; i++) {

  // Arrays are zero based (so 0 is round 1)
  // Add 1 to the current round
  roundNumber = (i + 1);

  // Write the current round to message
  msg += 'Round ' + roundNumber + ': ';

  // Get the score from the scores array
  msg += scores[i] + '<br />';
}

document.getElementById('answer').innerHTML = msg;
```

RESULT

The counter and array both start from 0 (rather than 1). So, within the loop, to select the current item from the array, you use the counter variable `i` to specify the item from the array, e.g., `scores[i]`. But remember that it is a number lower then you might expect (e.g., first iteration is 0, second is 1).

A `for` loop is often used to loop through the items in an array.

In this example, the scores for each round of a test are stored in an array called `scores`.

The total number of items in the array is stored in a variable called `arrayLength`. This number is obtained using the `length` property of the array.

There are three more variables: `roundNumber` holds the round of the test; `msg` holds the message to display; `i` is the counter (declared outside the loop).

The loop starts with the `for` keyword, then contains the condition inside the parentheses. As long as the counter is less than the total number of items in the array, the contents of the curly braces will continue to run. Each time the loop runs, the round number is increased by 1.

Inside the curly braces are rules that write the round number and the score to the `msg` variable. The variables declared outside of the loop are used within the loop.

The `msg` variable is then written into the page. It contains HTML so the `innerHTML` property is used to do this. Remember, p228 will talk about security issues relating to this property.

USING WHILE LOOPS

Here is an example of a `while` loop. It writes out the 5 times table. Each time the loop is run, another calculation is written into the variable called `msg`.

This loop will continue to run for as long as the condition in the parentheses is true. That condition is a counter indicating that, as long as the variable `i` remains less than 10, the statements in the subsequent code block should run.

Inside the code block there are two statements:

The first statement uses the `+=` operator, which is used to add new content to the `msg` variable. Each time the loop runs, a new calculation and line break is added to the end of the message being stored in it. So `+=` works as a shorthand for writing:
`msg = msg + 'new msg'`
(See bottom of the next page for a breakdown of this statement.)

The second statement increments the counter variable by one. (This is done inside the loop rather than with the condition.)

When the loop has finished, the interpreter goes to the next line of code, which writes the `msg` variable to the page.

c04/js/while-loop.js

JAVASCRIPT

```
var i = 1;          // Set counter to 1
var msg = '';       // Message

// Store 5 times table in a variable
while (i < 10) {
  msg += i + ' x 5 = ' + (i * 5) + '<br />';
  i++;
}

document.getElementById('answer').innerHTML = msg;
```

RESULT

In this example, the condition specifies that the code should run nine times. A more typical use of a `while` loop would be when you *do not know* how many times you want the code to run. It should continue to run as long as a condition is met.

USING DO WHILE LOOPS

JAVASCRIPT c04/js/do-while-loop.js

```
var i = 1;          // Set counter to 1
var msg = '';       // Message

// Store 5 times table in a variable
do {
  msg += i + ' x 5 = ' + (i * 5) + '<br />';s
  i++;
} while (i < 1);
// Note how this is already 1 and it still runs

document.getElementById('answer').innerHTML = msg;
```

RESULT

The key difference between a `while` loop and a `do while` loop is that the statements in the code block come *before* the condition. This means that those statements are run once whether or not the condition is met.

If you take a look at the condition, it is checking that the value of the variable called `i` is less than 1, but that variable has already been set to a value of 1.

Therefore, in this example the result is that the 5 times table is written out once, even though the counter is not less than 1.

Some people like to write `while` on a separate line from the closing curly brace before it.

Breaking down the first statement in these examples:

```
 1   2  3        4            5          6
msg  +=  i  +  ' x 5 = '  +  (i * 5)  +  '<br />';
```

1. Take variable called `msg`
2. Add to the following to its value
3. The number in the counter
4. Write out the string **x 5 =**
5. The counter multiplied by **5**
6. Add a line break

DARTH
VADER
and son

iPad
01:40 pm
52%
javascriptbook.com
Bullseye! Tutoring
BULLSEYE!
TARGET PRACTICE FOR YOUR MIND
1 + 3 = 4
2 + 3 = 5
3 + 3 = 6
4 + 3 = 7
5 + 3 = 8
6 + 3 = 9
7 + 3 = 10
8 + 3 = 11
9 + 3 = 12
10 + 3 = 13

EXAMPLE
DECISIONS & LOOPS

In this example, the user can either be shown addition or multiplication of a given number. The script demonstrates the use of both conditional logic and loops.

The example starts with two variables:

1. `number` holds the number that the calculations will be performed with (in this case it is the number 3)
2. `operator` indicates whether it should be addition or multiplication (in this case it is performing addition)

An `if...else` statement is used to decide whether to perform addition or multiplication with the number. If the variable called `operator` has the value `addition`, the numbers will be added together; otherwise they will be multiplied.

Inside the conditional statement, a `while` loop is used to calculate the results. It will run 10 times because the condition is checking whether the value of the counter is less than 11.

EXAMPLE
DECISIONS & LOOPS

c04/example.html

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bullseye! Tutoring</title>
    <link rel="stylesheet" href="css/c04.css" />
  </head>
  <body>
    <section id="page2">
      <h1>Bullseye</h1>
      <img src="images/teacher.png" id="teacher2" alt="" />
      <section id="blackboard"></section>
    </section>
    <script src="js/example.js"></script>
  </body>
</html>
```

The HTML for this example is very slightly different than the other examples in this chapter because there is a blackboard which the table is written onto.

You can see the script is added to the page just before the closing </body> tag.

EXAMPLE
DECISIONS & LOOPS

JAVASCRIPT

c04/js/example.js

```
var table = 3;                      // Unit of table
var operator = 'addition';          // Type of calculation (defaults to addition)
var i = 1;                          // Set counter to 1
var msg = '';                       // Message

if (operator === 'addition') {      // If the operator variable says addition
  while (i < 11) {                  // While counter is less than 11
    msg += i + ' + ' + table + ' = ' + (i + table) + '<br />'; // Calculation
    i++;                            // Add 1 to the counter
  }
} else {                            // Otherwise
  while (i < 11) {                  // While counter is less than 11
    msg += i + ' x ' + table + ' = ' + (i * table) + '<br />'; // Calculation
    i++;                            // Add 1 to the counter
  }
}

// Write the message into the page
var el = document.getElementById('blackboard');
el.innerHTML = msg;
```

If you read the comments in the code, you can see how this example works. The script starts by declaring four variables and setting values for them.

Then, an `if` statement checks whether the value of the variable called `operator` is `addition`. If it is, it uses a `while` loop to perform the calculations and store the results in a variable called `msg`.

If you change the value of the `operator` variable to anything other than `addition`, the conditional statement will select the second set of statements. These also contain a `while` loop, but this time it will perform multiplication (rather than addition).

When one of the loops has finished running, the last two lines of the script select the element whose `id` attribute has a value of `blackboard`, and updates the the page with the content of the `msg` variable.

SUMMARY

DECISIONS & LOOPS

- Conditional statements allow your code to make decisions about what to do next.
- Comparison operators (`===`, `!==`, `==`, `!=`, `<`, `>`, `<=`, `=>`) are used to compare two operands.
- Logical operators allow you to combine more than one set of comparison operators.
- `if...else` statements allow you to run one set of code if a condition is true, and another if it is false.
- `switch` statements allow you to compare a value against possible outcomes (and also provides a default option if none match).
- Data types can be coerced from one type to another.
- All values evaluate to either truthy or falsy.
- There are three types of loop: `for`, `while`, and `do...while`. Each repeats a set of statements.

5

DOCUMENT OBJECT MODEL

The Document Object Model (DOM) specifies how browsers should create a model of an HTML page and how JavaScript can access and update the contents of a web page while it is in the browser window.

The DOM is neither part of HTML, nor part of JavaScript; it is a separate set of rules. It is implemented by all major browser makers, and covers two primary areas:

MAKING A MODEL OF THE HTML PAGE

When the browser loads a web page, it creates a model of the page in memory.

The DOM specifies the way in which the browser should structure this model using a **DOM tree**.

The DOM is called an object model because the model (the DOM tree) is made of objects.

Each object represents a different part of the page loaded in the browser window.

ACCESSING AND CHANGING THE HTML PAGE

The DOM also defines methods and properties to access and update each object in this model, which in turn updates what the user sees in the browser.

You will hear people call the DOM an **Application Programming Interface (API)**. User interfaces let humans interact with programs; APIs let programs (and scripts) talk to each other. The DOM states what your script can ask the browser about the current page, and how to tell the browser to update what is being shown to the user.

In each example of this chapter, the JavaScript will amend the HTML list shown here. Colors are used to convey the priority and status of each list item:

HOT	COOL
NORMAL	COMPLETE

THE DOM TREE IS A MODEL OF A WEB PAGE

As a browser loads a web page, it creates a model of that page. The model is called a DOM tree, and it is stored in the browsers' memory. It consists of four main types of nodes.

BODY OF HTML PAGE

```
<html>
  <body>
    <div id="page">
      <h1 id="header">List</h1>
      <h2>Buy groceries</h2>
      <ul>
        <li id="one" class="hot"><em>fresh</em> figs</li>
        <li id="two" class="hot">pine nuts</li>
        <li id="three" class="hot">honey</li>
        <li id="four">balsamic vinegar</li>
      </ul>
      <script src="js/list.js"></script>
    </div>
  </body>
</html>
```

THE DOCUMENT NODE

Above, you can see the HTML code for a shopping list, and on the right hand page is its **DOM tree**. Every element, attribute, and piece of text in the HTML is represented by its own **DOM node**. At the top of the tree a **document node** is added; it represents the entire page (and also corresponds to the `document` object, which you first met on p36).

When you access any element, attribute, or text node, you navigate to it via the `document` node. It is the starting point for all visits to the DOM tree.

ELEMENT NODES

HTML elements describe the structure of an HTML page. (The `<h1>` - `<h6>` elements describe what parts are headings; the `<p>` tags indicate where paragraphs of text start and finish; and so on.)

To access the DOM tree, you start by looking for elements. Once you find the element you want, *then* you can access its text and attribute nodes if you want to. This is why you start by learning methods that allow you to access element nodes, before learning to access and alter text or attributes.

Note: We will continue to use this list example throughout this chapter and the next two chapters so that you can see how different techniques allow you to access and update the web page (which is represented by this DOM tree).

Relationships between the `document` and all of the element nodes are described using the same terms as a family tree: parents, children, siblings, ancestors, and descendants. (Every node is a descendant of the `document` node.)

Each node is an object with methods and properties.
Scripts access and update this DOM tree (not the source HTML file).
Any changes made to the DOM tree are reflected in the browser.

DOM TREE

ATTRIBUTE NODES

The opening tags of HTML elements can carry attributes and these are represented by attribute nodes in the DOM tree.

Attribute nodes are not *children* of the element that carries them; they are *part of* that element. Once you access an element, there are specific JavaScript methods and properties to read or change that element's attributes. For example, it is common to change the values of `class` attributes to trigger new CSS rules that affect their presentation.

TEXT NODES

Once you have accessed an element node, you can then reach the text within that element. This is stored in its own text node.

Text nodes cannot have children. If an element contains text and another child element, the child element is *not* a child of the text node but rather a child of the *containing* element. (See the `<em>` element on the first `<li>` item.) This illustrates how the text node is always a new branch of the DOM tree, and no further branches come off of it.

WORKING WITH THE DOM TREE

Accessing and updating the DOM tree involves two steps:
1: Locate the node that represents the element you want to work with.
2: Use its text content, child elements, and attributes.

STEP 1: ACCESS THE ELEMENTS

Here is an overview of the methods and properties that access elements covered on p192 – p211. The first two columns are known as DOM queries. The last column is known as traversing the DOM.

SELECT AN INDIVIDUAL ELEMENT NODE

Here are three common ways to select an individual element:

`getElementById()`
Uses the value of an element's `id` attribute (which should be unique within the page).
See p195

`querySelector()`
Uses a CSS selector, and returns the first matching element.
See p202

You can also select individual elements by traversing from one element to another within the DOM tree (see third column).

SELECT MULTIPLE ELEMENTS (NODELISTS)

There are three common ways to select multiple elements.

`getElementsByClassName()`
Selects all elements that have a specific value for their `class` attribute.
See p200

`getElementsByTagName()`
Selects all elements that have the specified tag name.
See p201

`querySelectorAll()`
Uses a CSS selector to select all matching elements.
See p202

TRAVERSING BETWEEN ELEMENT NODES

You can move from one element node to a related element node.

`parentNode`
Selects the parent of the current element node (which will return just one element).
See p208

`previousSibling` / `nextSibling`
Selects the previous or next sibling from the DOM tree.
See p210

`firstChild` / `lastChild`
Select the first or last child of the current element.
See p211

Throughout the chapter you will see notes where DOM methods only work in certain browsers or are buggy. Inconsistent browser support for the DOM was a key reason why jQuery became so popular.

The terms **elements** and **element nodes** are used interchangeably but when people say the DOM is working with an element, it is actually working with a node that *represents* that element.

STEP 2: WORK WITH THOSE ELEMENTS

Here is an overview of methods and properties that work with the elements introduced on p186.

ACCESS / UPDATE TEXT NODES

The text inside any element is stored inside a text node. To access the text node above:

1. Select the `<li>` element
2. Use the `firstChild` property to get the text node
3. Use the text node's only property (`nodeValue`) to get the text from the element

`nodeValue`
This property lets you access or update contents of a text node.
See p214

The text node does not include text inside any child elements.

WORK WITH HTML CONTENT

One property allows access to child elements and text content:
`innerHTML`
See p220

Another just the text content:
`textContent`
See p216

Several methods let you create new nodes, add nodes to a tree, and remove nodes from a tree:
`createElement()`
`createTextNode()`
`appendChild()` / `removeChild()`
This is called DOM manipulation.
See p222

ACCESS OR UPDATE ATTRIBUTE VALUES

Here are some of the properties and methods you can use to work with attributes:
`class`Name / `id`
Lets you get or update the value of the `class` and `id` attributes.
See p232

`hasAttribute()`
`getAttribute()`
`setAttribute()`
`removeAttribute()`
The first checks if an attribute exists. The second gets its value. The third updates the value. The fourth removes an attribute.
See p232

CACHING DOM QUERIES

Methods that find elements in the DOM tree are called DOM queries. When you need to work with an element more than once, you should use a variable to store the result of this query.

When a script selects an element to access or update, the interpreter must find the element(s) in the DOM tree.

Below, the interpreter is told to look through the DOM tree for an element whose **id** attribute has a value of **one**.

Once it has found the node that represents the element(s), you can work with that node, its parent, or any children.

```
getElementById('one');
```

When people talk about storing elements in variables, they are really storing the location of the element(s) within the DOM tree in a variable. The properties and methods of that element node work on the variable.

If your script needs to use the the same element(s) more than once, you can store the location of the element(s) in a variable.

This saves the browser looking through the DOM tree to find the same element(s) again. It is known as **caching** the selection.

Programmers would say that the variable stores a **reference** to the object in the DOM tree. (It is storing the *location* of the node.)

```
var itemOne = getElementById('one');
```

`itemOne` does not store the **`<li>`** element, it stores a *reference* to where that node is in the DOM tree. To access the text content of this element, you might use the variable name: **`itemOne.textContent`**

ACCESSING ELEMENTS

DOM queries may return one element, or they may return a NodeList, which is a collection of nodes.

Sometimes you will just want to access one individual element (or a fragment of the page that is stored within that one element). Other times you may want to select a group of elements, for example, every `<h1>` element in the page or every `<li>` element within a particular list.

Here, the DOM tree shows the body of the page of the list example. We focus on accessing elements first so it only shows element nodes. The diagrams in the coming pages highlight which elements a DOM query would return. (Remember, element *nodes* are the DOM representation of an element.)

GROUPS OF ELEMENT NODES

If a method *can* return more than one node, it will always return a **NodeList**, which is a **collection** of nodes (even if it only finds one matching element). You then need to select the element you want from this list using an index number (which means the numbering starts at 0 like the items in an array).

For example, several elements can have the same tag name, so `getElementsByTagName()` will always return a `NodeList`.

FASTEST ROUTE

Finding the quickest way to access an element within your web page will make the page seem faster and/or more responsive. This usually means evaluating the minimum number of nodes on the way to the element you want to work with. For example, `getElementById()` will quickly return one element (because no two elements on the same page should have the same value for an `id` attribute), but it can only be used when the element you want to access has an `id` attribute.

METHODS THAT RETURN A SINGLE ELEMENT NODE:

`getElementById('id')`

Selects an individual element given the value of its `id` attribute. The HTML must have an `id` attribute in order for it to be selectable.

First supported: IE5.5, Opera 7, all versions of Chrome, Firefox, Safari.

`getElementById('one')`

`querySelector('css selector')`

Uses CSS selector syntax that would select one or more elements. This method returns only the first of the matching elements.

First supported: IE8, Firefox 3.5, Safari 4, Chrome 4, Opera 10

`querySelector('li.hot')`

METHODS THAT RETURN ONE OR MORE ELEMENTS (AS A NODELIST):

`getElementsByClassName('class')`

Selects one or more elements given the value of their `class` attribute. The HTML must have a `class` attribute for it to be selectable. This method is faster than `querySelectorAll()`.

First supported: IE9, Firefox 3, Safari 4, Chrome 4, Opera 10
(Several browsers had partial / buggy support in earlier versions)

`getElementsByClassName('hot')`

`getElementsByTagName('tagName')`

Selects all elements on the page with the specified tag name. This method is faster than `querySelectorAll()`.

First supported: IE6+, Firefox 3, Safari 4, Chrome, Opera 10
(Several browsers had partial / buggy support in earlier versions)

`getElementsByTagName('li')`

`querySelectorAll('css selector')`

Uses CSS selector syntax to select one or more elements and returns all of those that match.

First supported: IE8, Firefox 3.5, Safari 4, Chrome 4, Opera 10

`querySelectorAll('li.hot')`

METHODS THAT SELECT INDIVIDUAL ELEMENTS

`getElementById()` and **`querySelector()`** can both search an entire document and return individual elements. Both use a similar syntax.

`getElementById()` is the quickest and most efficient way to access an element because no two elements can share the same value for their **`id`** attribute. The syntax for this method is shown below, and an example of its use is on the page to the right.

`querySelector()` is a more recent addition to the DOM, so it is not supported in older browsers. But it is very flexible because its parameter is a CSS selector, which means it can be used to accurately target many more elements.

`document` refers to the **`document`** object. You always have to access individual elements via the **`document`** object.

The **`getElementById()`** method indicates that you want to find an element based upon the value of its **`id`** attribute.

OBJECT

METHOD

```
document.getElementById('one')
```

MEMBER OPERATOR

The dot notation indicates that the method (on the right) is being applied to the node on the left of the period.

PARAMETER

The method needs to know the value of the **`id`** attribute on the element you want. It is the parameter of the method.

This code will return the element node for the element whose **`id`** attribute has a value of **one**. You often see element nodes stored in a variable for use later in the script (as you saw on p190).

Here the method is used on the **`document`** object so it looks for that element anywhere within the page. DOM methods can also be used on element nodes within the page to find descendants of that node.

SELECTING ELEMENTS USING ID ATTRIBUTES

HTML

c05/get-element-by-id.html

```
<h1 id="header">List King</h1>
<h2>Buy groceries</h2>
<ul>
  <li id="one" class="hot"><em>fresh</em>
    figs</li>
  <li id="two" class="hot">pine nuts</li>
  <li id="three" class="hot">honey</li>
  <li id="four">balsamic vinegar</li>
</ul>
```

JAVASCRIPT

c05/js/get-element-by-id.js

```
// Select the element and store it in a variable.
var el = document.getElementById('one');

// Change the value of the class attribute.
el.className = 'cool';
```

RESULT

This result window shows the example after the script has updated the first list item. The original state, before the script ran, is shown on p185.

`getElementById()` allows you to select a single element node by specifying the value of its `id` attribute.

This method has one parameter: the value of the `id` attribute on the element you want to select. This value is placed inside quote marks because it is a string. The quotes can be single or double quotes, but they must match.

In the example on the left , the first line of JavaScript code finds the element whose `id` attribute has a value of one, and stores a reference to that node in a variable called `el`.

The code then uses a property called `className` (which you meet on p232) to update the value of the `class` attribute of the element stored in this variable. Its value is `cool`, and this triggers a new rule in the CSS that sets the background color of the element to aqua.

Note how the `className` property is used on the variable that stores the reference to the element.

Browser Support: This is one of the oldest and best supported methods for accessing elements.

NODELISTS: DOM QUERIES THAT RETURN MORE THAN ONE ELEMENT

When a DOM method *can* return more than one element, it returns a NodeList (even if it only finds one matching element).

A NodeList is a collection of element nodes. Each node is given an index number (a number that starts at zero, just like an array).

The order in which the element nodes are stored in a NodeList is the same order that they appeared in the HTML page.

When a DOM query returns a NodeList, you may want to:

- Select one element from the NodeList.
- Loop through each item in the NodeList and perform the same statements on each of the element nodes.

NodeLists look like arrays and are numbered like arrays, but they are not actually arrays; they are a type of object called a **collection**.

Like any other object, a NodeList has properties and methods, notably:

- The `length` property tells you how many items are in the NodeList.
- The `item()` method returns a specific node from the NodeList when you tell it the index number of the item that you want (in the parentheses). However, it is more common to use array syntax (with square brackets) to retrieve an item from a NodeList (as you will see on p199).

LIVE & STATIC NODELISTS

There are times when you will want to work with the same selection of elements several times, so the NodeList can be stored in a variable and re-used (rather than collecting the same elements again).

In a **live NodeList**, when your script updates the page, the NodeList is updated at the same time. The methods beginning `getElementsBy...` return live NodeLists. They are also typically faster to generate than static NodeLists.

In a **static NodeList** when your script updates the page, the NodeList is not updated to reflect the changes made by the script.

The new methods that begin `querySelector...` (which use CSS selector syntax) return static NodeLists. They reflect the document when the query was made. If the script changes the content of the page, the NodeList is not updated to reflect those changes.

Here you can see four different DOM queries that all return a NodeList. For each query, you can see the elements and their index numbers in the NodeList that is returned.

`getElementsByTagName('h1')`

Even though this query only returns one element, the method still returns a NodeList because of the potential for returning more than one element.

INDEX NUMBER & ELEMENT
0 `<h1>`

`getElementsByTagName('li')`

This method returns four elements, one for each of the `<li>` elements on the page. They appear in the same order as they do in the HTML page.

INDEX NUMBER & ELEMENT
0 `<li id="one" class="hot">`
1 `<li id="two" class="hot">`
2 `<li id="three" class="hot">`
3 `<li id="four">`

`getElementsByClassName('hot')`

This NodeList contains only three of the `<li>` elements because we are searching for elements by the value of their `class` attribute, not tag name.

INDEX NUMBER & ELEMENT
0 `<li id="one" class="hot">`
1 `<li id="two" class="hot">`
2 `<li id="three" class="hot">`

`querySelectorAll('li[id]')`

This method returns four elements, one for each of the `<li>` elements on the page that have an `id` attribute (regardless of the values of the `id` attributes).

INDEX NUMBER & ELEMENT
0 `<li id="one" class="hot">`
1 `<li id="two" class="hot">`
2 `<li id="three" class="hot">`
3 `<li id="four">`

SELECTING AN ELEMENT FROM A NODELIST

There are two ways to select an element from a NodeList: The **item()** method and array syntax. Both require the index number of the element you want.

THE item() METHOD

NodeLists have a method called **item()** which will return an individual node from the NodeList.

You specify the index number of the element you want as a parameter of the method (inside the parentheses).

Executing code when there are no elements to work with wastes resources. So programmers often check that there is at least one item in the NodeList before running any code. To do this, use the **length** property of the NodeList – it tells you how many items the NodeList contains.

Here you can see that an **if** statement is used. The condition for the **if** statement is whether the **length** property of the NodeList is greater than zero. If it is, then the statements inside the **if** statement are executed. If not, the code continues to run after the second curly brace.

```
var elements = document.getElementsByClassName('hot')
if (elements.length >= 1) {
    var firstItem = elements.item(0);
}
```

1

Select elements that have a **class** attribute whose value is **hot** and store the NodeList in a variable called **elements**.

2

Use the **length** property to check how many elements were found. If 1 or more are found, run the code in the **if** statement.

3

Store the first element from the NodeList in a variable called **firstItem**. (It says **0** because index numbers start at zero.)

Array syntax is preferred over the **`item()`** method because it is faster. Before selecting a node from a NodeList, check that it contains nodes. If you repeatedly use the NodeList, store it in a variable.

ARRAY SYNTAX

You can access individual nodes using a square bracket syntax similar to that used to access individual items from an array.

You specify the index number of the element you want inside square brackets that follow the NodeList.

As with all DOM queries, if you need to access the same NodeList several times, store the result of the DOM query in a variable.

In the examples on both of these pages, the NodeList is stored in a variable called **`elements`**.

If you create a variable to hold a NodeList (as shown below) but there are no matching elements, the variable will be an empty NodeList. When you check the **`length`** property of the variable, it will return the number **0** because it does not contain any elements.

```
var elements = document.getElementsByClassName('hot');
if (elements.length >= 1) {
    var firstItem = elements[0];
}
```

1

Create a NodeList containing elements that have a **`class`** attribute whose value is **`hot`**, and store it in the variable **`elements`**.

2

If that number is greater than or equal to one, run the code inside the **`if`** statement.

3

Get the first element from the NodeList (it says **0** because index numbers start at zero).

SELECTING ELEMENTS USING CLASS ATTRIBUTES

The `getElementsByClassName()` method allows you to select elements whose `class` attribute contains a specific value.

The method has one parameter: the `class` name which is given in quotes within the parentheses after the method name.

Because several elements can have the same value for their `class` attribute, this method always returns a NodeList.

c05/js/get-elements-by-class-name.js

JAVASCRIPT

```
var elements = document.getElementsByClassName('hot'); // Find hot items

if (elements.length > 2) {                             // If 3 or more are found

    var el = elements[2];            // Select the third one from the NodeList
    el.className = 'cool';           // Change the value of its class attribute

}
```

This example starts by looking for elements whose `class` attribute *contains* `hot`. (The value of a `class` attribute can contain several class names, each separated by a space.) The result of this DOM query is stored in a variable called `elements` because it is used more than once in the example.

An `if` statement checks if the query found more than two elements. If so, the third one is selected and stored in a variable called `el`. The `class` attribute of that element is then updated to say `class`. (In turn, this triggers a new CSS style, changing the presentation of that element.)

Browser Support: IE9, Firefox 3, Chrome 4, Opera 9.5, Safari 3.1

RESULT

SELECTING ELEMENTS BY TAG NAME

The `getElementsByTagName()` method allows you to select elements using their tag name.

The element name is specified as a parameter, so it is placed inside the parentheses and is contained by quote marks.

Note that you do not include the angled brackets that surround the tag name in the HTML (just the letters inside the brackets).

JAVASCRIPT

c05/js/get-elements-by-tag-name.js

```
var elements = document.getElementsByTagName('li');   // Find <li> elements

if (elements.length > 0) {                            // If 1 or more are found

    var el = elements[0];            // Select the first one using array syntax
    el.className = 'cool';           // Change the value of the class attribute

}
```

RESULT

This example looks for any `<li>` elements in the document. It stores the result in a variable called `elements` because the result is used more than once in this example.

An `if` statement checks if any `<li>` elements were found. As with any element that can return a `NodeList`, you check that there will be a suitable element before you try to work with it.

If matching elements were found, the first one is selected and its `class` attribute is updated. This changes the color of the list item to make it aqua.

Browser Support: Very good – it is safe to use in any scripts.

SELECTING ELEMENTS USING CSS SELECTORS

`querySelector()` returns the first element node that matches the CSS-style selector. `querySelectorAll()` returns a NodeList of all of the matches.

Both methods take a CSS selector as their only parameter. The CSS selector syntax offers more flexibility and accuracy when selecting an element than just specifying a class name or a tag name, and should also be familiar to front-end web developers who are used to targeting elements using CSS.

c05/js/query-selector.js

JAVASCRIPT

```
// querySelector() only returns the first match
var el = document.querySelector('li.hot');
el.className = 'cool';

// querySelectorAll returns a NodeList
// The second matching element (the third list item) is selected and changed
var els = document.querySelectorAll('li.hot');
els[1].className = 'cool';
```

These two methods were introduced by browser manufacturers because a lot of developers were including scripts like jQuery in their pages so that they could select elements using CSS selectors. (You meet jQuery in Chapter 7.)

If you look at the final line of code, array syntax is used to select the second item from the NodeList, even though that NodeList is stored in a variable.

RESULT

Browser Support: The drawback with these two methods is that they are only supported in more recent browsers.

IE8+ (released Mar 2009)
Firefox 3.5+ (released Jun 2009)
Chrome 1+ (released Sep 2008)
Opera 10+ (released Sep 2009)
Safari 3.2+ (released Nov 2008)

JavaScript code runs one line at a time, and statements affect the content of a page as the interpreter processes them.

If a DOM query runs when a page loads, the same query could return different elements if it is used again later in the page.

Below you can see how the example on the left-hand page (`query-selector.js`) changes the DOM tree as it runs.

1: WHEN THE PAGE FIRST LOADS

HTML c05/query-selector.html

```
<ul>
  <li id="one" class="hot">
    <em>fresh</em> figs</li>
  <li id="two" class="hot">pine nuts</li>
  <li id="three" class="hot">honey</li>
  <li id="four">balsamic vinegar</li>
</ul>
```

1. This is how the page starts. There are three `<li>` elements that have a `class` attribute whose value is `hot`. The `querySelector()` method finds the first one, and updates the value of its `class` attribute from `hot` to `cool`. This also updates the DOM tree stored in memory so – after this line has run – only the second and third `<li>` elements have a `class` attribute with a value of `hot`.

2: AFTER THE FIRST SET OF STATEMENTS

HTML c05/query-selector.html

```
<ul>
  <li id="one" class="cool">
    <em>fresh</em> figs</li>
  <li id="two" class="hot">pine nuts</li>
  <li id="three" class="hot">honey</li>
  <li id="four">balsamic vinegar</li>
</ul>
```

2. When the second selector runs, there are now only two `<li>` elements whose `class` attributes have a value of `hot` (see left), so it just selects these two. This time, array syntax is used to work with the second of the matching elements (which is the third list item). Again the value of its `class` attribute is changed from `hot` to `cool`.

3: AFTER THE SECOND SET OF STATEMENTS

HTML c05/query-selector.html

```
<ul>
  <li id="one" class="cool">
    <em>fresh</em> figs</li>
  <li id="two" class="hot">pine nuts</li>
  <li id="three" class="cool">honey</li>
  <li id="four">balsamic vinegar</li>
</ul>
```

3. When the second selector has done its job, the DOM tree now only holds one `<li>` element whose `class` attribute has a value of `hot`. Any further code looking for `<li>` elements whose `class` attribute has a value of `hot` would find only this one. However, if they were looking for `<li>` elements whose `class` attribute has a value of `cool`, they would find *two* matching element nodes.

REPEATING ACTIONS FOR AN ENTIRE NODELIST

When you have a NodeList, you can loop through each node in the collection and apply the same statements to each.

In this example, once a NodeList has been created, a **for** loop is used to go through each element in the NodeList.

All of the statements inside the **for** loop's curly braces are applied to each element in the NodeList one-by-one.

To indicate which item of the NodeList is currently being worked with, the counter **i** is used in the array-style syntax.

```
var hotItems = document.querySelectorAll('li.hot');
for (var i = 0; i < hotItems.length; i++) {
  hotItems[i].className = 'cool';
}
```

1

The variable **hotItems** contains a NodeList. It contains all list items whose **class** attribute has a value of **hot**. They are collected using the **querySelectorAll()** method.

2

The **length** property of the NodeList indicates how many elements are in the NodeList. The number of elements dictates how many times the loop should run.

3

Array syntax is used to indicate which item in the NodeList is currently being worked with: **hotItems[i]**
It uses the counter variable inside the square brackets.

LOOPING THROUGH A NODELIST

If you want to apply the same code to numerous elements, looping through a NodeList is a powerful technique.

It involves finding out how many items are in the NodeList, and then setting a counter to loop through them, one-by-one.

Each time the loop runs, the script checks that the counter is less than the total number of items in the NodeList.

JAVASCRIPT c05/js/node-list.js

```
var hotItems = document.querySelectorAll('li.hot'); // Store NodeList in array

if (hotItems.length > 0) {                          // If it contains items

    for (var i=0; i<hotItems.length; i++) {     // Loop through each item
        hotItems[i].className = 'cool';     // Change value of class attribute
    }

}
```

RESULT

In this example, the NodeList is generated using `querySelectorAll()`, and it is looking for any `<li>` elements that have a `class` attribute whose value is `hot`.

The NodeList is stored in a variable called `hotItems`, and the number of elements in the list is found using the `length` property.

For each of the elements in the NodeList, the value of the `class` attribute is changed to `cool`.

LOOPING THROUGH A NODELIST: PLAY-BY-PLAY

START

i = 0

i = 1

At the start of this example, there are three list items with a **`class`** attribute whose value is **`hot`** so the value of **`hotItems.length`** is **3**.

At first, the value of the counter is set to **0**, so the first item from the NodeList (which has an index of **0**) is targeted and the value of its **`class`** attribute is set to **`cool`**.

```
for (var i = 0; i < hotItems.length; i++) {
    hotItems[i].className = 'cool';
}
```

When the value of the counter is **1**, the second item from the NodeList (which has an index of **1**) is targeted and the value of its **`class`** attribute is set to **`cool`**.

When the value of the counter is **2**, the third item from the NodeList (which has an index of **2**) is targeted and the value of its **`class`** attribute is set to **`cool`**.

When the value of the counter is **3**, the condition no longer returns **`true`**, so the loops ends. The script then continues to the first line of code after the loop.

TRAVERSING THE DOM

When you have an element node, you can select another element in relation to it using these five properties. This is known as traversing the DOM.

parentNode

This property finds the element node for the containing (or parent) element in the HTML.

(1) If you started with the first <li> element, then its *parent node* would be the one representing the <ul> element.

previousSibling nextSibling

These properties find the previous or next sibling of a node if there are siblings.

If you started with the first <li> element, it would not have a *previous sibling*. However, its *next sibling* (2) would be the node representing the second <li>.

firstChild lastChild

These properties find the first or last child of the current element.

If you started with the <ul> element, the *first child* would be the node representing the first <li> element, and (3) the *last child* would be the last <li>.

These are properties of the current node (not methods to select an element); therefore, they do not end in parentheses.

If you use these properties and they do not have a previous/next sibling, or a first/last child, the result will be null.

These properties are read-only; they can only be used to select a new node, not to update a parent, sibling, or child.

WHITESPACE NODES

Traversing the DOM can be difficult because some browsers add a text node whenever they come across whitespace between elements.

Most browsers, except IE, treat whitespace between elements (such as spaces or carriage returns) as a text node, so the properties below return different elements in different browsers:

```
previousSibling
nextSibling
firstChild
lastChild
```

Below, you can see all of the whitespace nodes added to the DOM tree for the list example. Each one is represented by a green square. You could strip all the whitespace out of a page before serving it to the browser. This would also make the page smaller and faster to serve/load. However, it would also make the code much harder to read.

Another way around this problem is to avoid using these DOM properties altogether.

One of the most popular ways to address this kind of problem is to use a JavaScript library such as jQuery, which helps deal with such problems. These types of browser inconsistencies were a big factor in jQuery's popularity.

Internet Explorer (shown above) ignores whitespace and does not create extra text nodes.

Chrome, Firefox, Safari, and Opera create text nodes from whitespace (spaces and carriage returns).

PREVIOUS & NEXT SIBLING

You have just seen that these properties can return inconsistent results in different browsers. However, it is safe to use them when there is no whitespace between elements.

For this example, all spaces between the HTML elements have been removed. In order to demonstrate these properties, the second list item is selected using `getElementById()`.

From this element node, the `previousSibling` property will return the first `<li>` element, and the `nextSibling` property will return the third `<li>` element.

c05/sibling.html

HTML

```
<ul><li id="one" class="hot"><em>fresh</em> figs</li><li id="two"
class="hot">pine nuts</li><li id="three" class="hot">honey</li><li
id="four">balsamic vinegar</li></ul>
```

c05/js/sibling.js

JAVASCRIPT

```
// Select the starting point and find its siblings
var startItem = document.getElementById('two');
var prevItem = startItem.previousSibling;
var nextItem = startItem.nextSibling;

// Change the values of the siblings' class attributes
prevItem.className = 'complete';
nextItem.className = 'cool';
```

Note how references to sibling nodes are stored in new variables. This means properties such as `className` can be used on that node by adding the dot notation between the variable name and the property.

RESULT

FIRST & LAST CHILD

These properties also return inconsistent results if there is whitespace between elements. In this example, a slightly different solution is used in the HTML – the closing tags are put next to the opening tags of the next element, making it a little more readable. The example starts by using the `getElementsByTagName()` method to select the `<ul>` element from the page. From this element node, the `firstChild` property will return the first `<li>` element, and the `lastChild` property will return the last `<li>` element.

HTML

c05/child.html

```
<ul
  ><li id="one" class="hot"><em>fresh</em> figs</li
  ><li id="two" class="hot">pine nuts</li
  ><li id="three" class="hot">honey</li
  ><li id="four">balsamic vinegar</li
></ul>
```

JAVASCRIPT

c05/js/child.js

```
// Select the starting point and find its children
var startItem = document.getElementsByTagName('ul')[0];
var firstItem = startItem.firstChild;
var lastItem = startItem.lastChild;

// Change the values of the children's class attributes
firstItem.setAttribute('class', 'complete');
lastItem.setAttribute('class', 'cool');
```

RESULT

HOW TO GET/UPDATE ELEMENT CONTENT

So far this chapter has focused on finding elements in the DOM tree. The rest of this chapter shows how to access/update element content. Your choice of techniques depends upon what the element contains.

Take a look at the three examples of `<li>` elements on the right. Each one adds some more markup and, as a result, the fragment of the DOM tree for each list item is very different.

- The first (on this page) just contains text.
- The second and third (on the right-hand page) contain a mix of text and an `<em>` element.

You can see that by adding something as simple as an `<em>` element, the DOM tree's structure changes significantly. In turn, this affects how you might work with that list item. When an element contains a mix of text and other elements, you are more likely to work with the containing element rather than the individual nodes for each descendant.

```
<li id="one">figs</li>
```

Above, the `<li>` element has:

- One *child* node holding the word that you can see in the list item: `figs`
- An attribute node holding the `id` attribute.

To work with the content of elements you can:

- **Navigate to the text nodes**. This works best when the element contains only text, no other elements.
- **Work with the containing element**. This allows you to access its text nodes and child elements. It works better when an element has text nodes and child elements that are siblings.

TEXT NODES

Once you have navigated from an element to its text node, there is one property that you will commonly find yourself using:

PROPERTY	DESCRIPTION
`nodeValue`	Accesses text from node (p214)

```
<li id="one"><em>fresh</em> figs</li>
```

An element is added. It becomes the *first child*.

- The element node has its own *child* text node which contains the word `fresh`.
- The original text node is now a *sibling* of the node that represents the element.

```
<li id="one">six <em>fresh</em> figs</li>
```

When text is added before the element:

- The *first child* of the <li> element is a text node, which contains the word `six`.
- It has a *sibling* which is an element node for the element. In turn, that element node has a *child* text node containing the word `fresh`.
- Finally, there is a text node holding the word `figs`, which is a sibling of both the text node for the word "six" and the element node, .

CONTAINING ELEMENT

When you are working with an element node (rather than its text node), that element can contain markup. You have to choose whether you want to retrieve (get) or update (set) the markup as well as the text.

PROPERTY	DESCRIPTION	
`innerHTML`	Gets/sets text & markup	(p220)
`textContent`	Gets/sets text only	(p216)
`innerText`	Gets/sets text only	(p216)

When you use these properties to update the content of an element, the new content will overwrite the entire contents of the element (both text and markup).

For example, if you used any of these properties to update the content of the <body> element, it would update the entire web page.

ACCESS & UPDATE A TEXT NODE WITH NODEVALUE

When you select a text node, you can retrieve or amend the content of it using the `nodeValue` property.

```
<li id="one"><em>fresh</em> figs</li>
```

The code below shows how you access the second text node. It will return the result: `figs`

```
document.getElementById('one').firstChild.nextSibling.nodeValue;
```

① ② ③ ④

In order to use `nodeValue`, you must be on a text node, not the element that contains the text.

This example shows that navigating from the element node to a text node can be complicated.

If you do not know whether there will be element nodes alongside text nodes, it is easier to work with the containing element.

1. The `<li>` element node is selected using the `getElementById()` method.
2. The *first child* of `<li>` is the `<em>` element.
3. The text node is the *next sibling* of that `<em>` element.
4. You have the text node and can access its contents using `nodeValue`.

ACCESSING & CHANGING A TEXT NODE

To work with text in an element, first the element node is accessed and then its text node.

The text node has a property called `nodeValue` which returns the text in that text node.

You can also use the `nodeValue` property to update the content of a text node.

JAVASCRIPT

c05/js/node-value.js

```
var itemTwo = document.getElementById('two');        // Get second list item

var elText  = itemTwo.firstChild.nodeValue;          // Get its text content

elText = elText.replace('pine nuts', 'kale');        // Change pine nuts to kale

itemTwo.firstChild.nodeValue = elText;               // Update the list item
```

RESULT

This example takes the text content of the second list item and changes it from `pine nuts` to `kale`.

The first line collects the second list item. It is stored in a variable called `itemTwo`.

Next the text content of that element is stored in a variable called `elText`.

The third line of text replaces the words `'pine nuts'` with `'kale'` using the `String` object's `replace()` method.

The last line uses the `nodeValue` property to update the content of the text node with the updated value.

ACCESS & UPDATE TEXT WITH TEXTCONTENT (& INNERTEXT)

The `textContent` property allows you to collect or update just the text that is in the containing element (and its children).

textContent

To collect the text from the `<li>` elements in our example (and ignore any markup inside the element) you can use the `textContent` property on the containing `<li>` element. In this case it would return the value: `fresh figs`.

You can also use this property to update the content of the element; it replaces the entire content of it (including any markup).

```
<li id="one"><em>fresh</em> figs</li>
```

```
document.getElementById('one').textContent;
```

One issue with the `textContent` property is that Internet Explorer did not support it until IE9. (All other major browsers support it.)

innerText

You may also come across a property called `innerText`, but you should generally avoid it for three key reasons:

SUPPORT

Although most browser manufacturers adopted the property, Firefox does not because `innerText` is not part of any standard.

OBEYS CSS

It will not show any content that has been hidden by CSS. For example, if there were a CSS rule that hid the `<em>` elements, the `innerText` property would return only the word `figs`.

PERFORMANCE

Because the `innerText` property takes into account layout rules that specify whether the element is visible or not, it can be slower to retrieve the content than the `textContent` property.

ACCESSING TEXT ONLY

In order to demonstrate the difference between `textContent` and `innerText`, this example features a CSS rule to hide the contents of the `<em>` element.

The script starts off by getting the content of the first list item using both the `textContent` property and `innerText`. It then writes the values after the list.

Finally, the value of the first list item is then updated to say sourdough bread. This is done using the `textContent` property.

JAVASCRIPT c05/js/inner-text-and-text-content.js

```
var firstItem = document.getElementById('one');    // Find first list item
var showTextContent = firstItem.textContent;       // Get value of textContent
var showInnerText = firstItem.innerText;           // Get value of innerText

// Show the content of these two properties at the end of the list
var msg = '<p>textContent: ' + showTextContent + '</p>';
    msg += '<p>innerText: ' + showInnerText + '</p>';
var el = document.getElementById('scriptResults');
el.innerHTML = msg;

firstItem.textContent = 'sourdough bread';         // Update the first list item
```

RESULT

In most browsers:

- `textContent` collects the words `fresh figs`.
- `innerHTML` just shows `figs` (because `fresh` was hidden by the CSS).

But:

- In IE8 or earlier, the `textContent` property does not work.
- In Firefox, the `innerText` property will return `undefined` because the it was never implemented in Firefox.

ADDING OR REMOVING HTML CONTENT

There are two very different approaches to adding and removing content from a DOM tree: the **innerHTML** property and DOM manipulation.

THE innerHTML PROPERTY

Note: there are security risks associated with using **innerHTML** – these issues are described on p228.

APPROACH

innerHTML can be used on any element node. It is used both to retrieve and replace content. To update an element, new content is provided as a string. It can contain markup for descendant elements.

ADDING CONTENT

To add new content:

1. Store new content (including markup) as a string in a variable.
2. Select the element whose content you want to replace.
3. Set the element's **innerHTML** property to be the new string.

REMOVING CONTENT

To remove all content from an element, you set **innerHTML** to an empty string. To remove one element from a DOM fragment, e.g., one **<li>** from a **<ul>**, you need to provide the entire fragment minus that element.

EXAMPLE: CHANGING A LIST ITEM

1: Create variable holding markup

```
var item;
item = '<em>Fresh</em> figs';
```

You can have as much or as little markup in the variable as you want. It is a quick way to add a lot of markup to the DOM tree.

2: Select element whose content you want to update

3: Update content of selected element with new markup

DOM manipulation easily targets individual nodes in the DOM tree, whereas **`innerHTML`** is better suited to updating entire fragments.

DOM MANIPULATION METHODS

DOM manipulation can be safer than using **`innerHTML`**, but it requires more code and can be slower.

APPROACH

DOM manipulation refers to a set of DOM methods that allow you to create element and text nodes, and then attach them to the DOM tree or remove them from the DOM tree.

ADDING CONTENT

To add content, you use a DOM method to create new content one node at a time and store it in a variable. Then another DOM method is used to attach it to the right place in the DOM tree.

REMOVING CONTENT

You can remove an element (along with any contents and child elements it may contain) from the DOM tree using a single method.

EXAMPLE: ADDING A LIST ITEM

1: Create new text node

2: Create new element node

3: Add text node to element node

4: Select element you want to add the new fragment to

5: Append the new fragment to the selected element

ACCESS & UPDATE TEXT & MARKUP WITH INNERHTML

Using the `innerHTML` property, you can access and amend the contents of an element, including any child elements.

innerHTML

When getting HTML from an element, the `innerHTML` property will get the content of an element and return it as one long string, including any markup that the element contains.

When used to set new content for an element, it will take a string that can contain markup and process that string, adding any elements within it to the DOM tree.

When adding new content using `innerHTML`, be aware that one missing closing tag could throw out the design of the entire page.

Even worse, if `innerHTML` is used to add content that your users created to a page, they could add malicious content. See p228.

```
<li id="one"><em>fresh</em> figs</li>
```

GET CONTENT

The following line of code collects the content of the list item and adds it to a variable called `elContent`:

```
var elContent = document.getElementById('one').innerHTML;
```

The `elContent` variable would now hold the string:

```
'<em>fresh</em> figs'
```

SET CONTENT

The following line of code adds the content of the `elContent` variable (including any markup) to the first list item:

```
document.getElementById('one').innerHTML = elContent;
```

UPDATE TEXT & MARKUP

This example starts by storing the first list item in a variable called `firstItem`.

It then retrieves the content of this list item and stores it in a variable called `itemContent`.

Finally, the content of the list item is placed inside a link. Note how the quotes are escaped.

JAVASCRIPT

c05/js/inner-html.js

```
// Store the first list item in a variable
var firstItem = document.getElementById('one');

// Get the content of the first list item
var itemContent = firstItem.innerHTML;

// Update the content of the first list item so it is a link
firstItem.innerHTML = '<a href=\"http://example.org\">' + itemContent + '</a>';
```

RESULT

As the content of the string is added to the element using the `innerHTML` property, the browser will add any elements in the string to the DOM. In this example, an <a> element has been added to the page. (Any new elements will also be available to other scripts in the page.)

If you use attributes in your HTML code, escaping the quotation using the backslash character \ can make it clearer that those characters are not part of the script.

ADDING ELEMENTS USING DOM MANIPULATION

DOM manipulation offers another technique to add new content to a page (rather than `innerHTML`). It involves three steps:

1

CREATE THE ELEMENT

`createElement()`

You start by creating a new element node using the `createElement()` method. This element node is stored in a variable.

When the element node is created, it is not yet part of the DOM tree. It is not added to the DOM tree until step 3.

2

GIVE IT CONTENT

`createTextNode()`

`createTextNode()` creates a new text node. Again, the node is stored in a variable. It can be added to the element node using the `appendChild()` method.

This provides the content for the element, although you can skip this step if you want to attach an empty element to the DOM tree.

3

ADD IT TO THE DOM

`appendChild()`

Now that you have your element (optionally with some content in a text node), you can add it to the DOM tree using the `appendChild()` method.

The `appendChild()` method allows you to specify which element you want this node added to, as a child of it.

In the example at the end of the chapter, you will see another method that can be used to insert an element into the DOM tree. The `insertBefore()` method is used to add a new element before the selected DOM node.

DOM manipulation and `innerHTML` both have uses. You will see a discussion of when to choose each method on p226.

Note: You may see developers leave an empty element in their HTML pages in order to attach new content to that element, but this practice is best avoided unless absolutely necessary.

ADDING AN ELEMENT TO THE DOM TREE

`createElement()` creates an element that can be added to the DOM tree, in this case an empty `<li>` element for the list.

This new element is stored inside a variable called `newEl` until it is attached to the DOM tree later on.

`createTextNode()` allows you to create a new text node to attach to an element. It is stored in a variable called `newText`.

JAVASCRIPT c05/js/add-element.js

```
// Create a new element and store it in a variable.
var newEl = document.createElement('li');

// Create a text node and store it in a variable.
var newText = document.createTextNode('quinoa');

// Attach the new text node to the new element.
newEl.appendChild(newText);

// Find the position where the new element should be added.
var position = document.getElementsByTagName('ul')[0];

// Insert the new element into its position.
position.appendChild(newEl);
```

RESULT

The text node is added to the new element node using `appendChild()`.

The `getElementsByTagName()` method selects the position in the DOM tree to insert the new element (the first `<ul>` element in the page).

Finally, `appendChild()` is used again – this time to insert the new element and its content into the DOM tree.

REMOVING ELEMENTS VIA DOM MANIPULATION

DOM manipulation can be used to remove elements from the DOM tree.

1

STORE THE ELEMENT TO BE REMOVED IN A VARIABLE

You start by selecting the element that is going to be removed and store that element node in a variable.

You can use any of the methods you saw in the section on DOM queries to select the element.

2

STORE THE PARENT OF THAT ELEMENT IN A VARIABLE

Next, you find the parent element that contains the element you want to remove and store that element node in a variable.

The simplest way to get this element is to use the `parentNode` property of this element.

3

REMOVE THE ELEMENT FROM ITS CONTAINING ELEMENT

The `removeChild()` method is used on the containing element that you selected in step 2.

The `removeChild()` method takes one parameter: the reference to the element that you no longer want.

When you remove an element from the DOM, it will also remove any child elements.

The example on the right is quite simple, but this technique can significantly alter the DOM tree.

Removing elements from the DOM will affect the index number of siblings in a NodeList.

REMOVING AN ELEMENT FROM THE DOM TREE

This example uses the `removeChild()` method to remove the fourth item from the list (along with its contents).

The first variable, `removeEl`, stores the actual element you want to remove from the page (the fourth list item).

The second variable, `containerEl`, stores the `<ul>` element that *contains* the element you want to remove.

JAVASCRIPT

c05/js/remove-element.js

```
var removeEl = document.getElementsByTagName('li')[3]; // The element to remove

var containerEl = removeEl.parentNode;                 // Its containing element

containerEl.removeChild(removeEl);                     // Removing the element
```

RESULT

The `removeChild()` method is used on the variable that holds the container node.

It requires one parameter: the element you want to remove (which is stored in the second variable).

COMPARING TECHNIQUES: UPDATING HTML CONTENT

So far, you have seen three techniques for adding HTML to a web page. It's time to compare when you should use each one.

In any programming language, there are often several ways to achieve the same task. In fact, if you asked ten programmers to write the same script, you may well find ten different approaches.

Some programmers can be rather opinionated and believe that their way is always the "right" way to do things – when there are often several right ways. If you understand why people prefer some approaches over others, then you are in a strong position to decide whether it meets the needs of your project.

`document.write()`

The document object's `write()` method is a simple way to add content that was not in the original source code to the page, but its use is rarely advised.

ADVANTAGES

- It is a quick and easy way to show beginners how content can be added to a page.

DISADVANTAGES

- It only works when the page initially loads.
- If you use it after the page has loaded it can:
  1. Overwrite the whole page
  2. Not add the content to the page
  3. Create a new page
- It can cause problems with XHTML pages that are strictly validated.
- This method is very rarely used by programmers these days and is generally frowned upon.

You can choose different techniques depending on the task (and keep in mind how the site might be developed in the future).

element.innerHTML

The `innerHTML` property lets you get/update the entire content of any element (including markup) as a string.

ADVANTAGES

- You can use it to add a lot of new markup using less code than DOM manipulation methods.
- It can be faster than DOM manipulation when adding a lot of new elements to a web page.
- It is a simple way to remove all of the content from one element (by assigning it a blank string).

DISADVANTAGES

- It should not be used to add content that has come from a user (such as a username or blog comment), as it can pose a significant security risk which is discussed over the next four pages.
- It can be difficult to isolate single elements that you want to update within a larger DOM fragment.
- Event handlers may no longer work as intended.

DOM MANIPULATION

DOM manipulation refers to using a set of methods and properties to access, create, and update elements and text nodes.

ADVANTAGES

- It is suited to changing one element from a DOM fragment where there are many siblings.
- It does not affect event handlers.
- It easily allows a script to add elements incrementally (when you do not want to alter a lot of code at once).

DISADVANTAGES

- If you have to make a lot of changes to the content of a page, it is slower than `innerHTML`.
- You need to write more code to achieve the same thing compared with `innerHTML`.

CROSS-SITE SCRIPTING (XSS) ATTACKS

If you add HTML to a page using `innerHTML` (or several jQuery methods), you need to be aware of **Cross-Site Scripting Attacks** or **XSS**; otherwise, an attacker could gain access to your users' accounts.

This book has several warnings about security issues when you add HTML to a page using `innerHTML`. (There are also notes about it when using jQuery.)

The next four pages describe the issues you need to be aware of, and how to make your site secure against these kinds of attacks.

HOW XSS HAPPENS

XSS involves an attacker placing malicious code into a site. Websites often feature content created by many different people. For example:

- Users can create profiles or add comments
- Multiple authors may contribute articles
- Data can come from third-party sites such as Facebook, Twitter, news tickers, and other feeds
- Files such as images and video may be uploaded

Data you do not have complete control over is known as **untrusted data**; it must be handled with care.

WHAT CAN THESE ATTACKS DO?

XSS can give the attacker access to information in:

- The DOM (including form data)
- That website's cookies
- Session tokens: information that identifies you from other users when you log into a site

This could let the attacker access a user account and:

- Make purchases with that account
- Post defamatory content
- Spread their malicious code further / faster

EVEN SIMPLE CODE CAN CAUSE PROBLEMS:

Malicious code often mixes HTML and JavaScript (although URLs and CSS can be used to trigger XSS attacks). The two examples below demonstrate how fairly simple code could help an attacker access a user's account.

This first example stores cookie data in a variable, which could then be sent to a third-party server:

```
<script>var adr='http://example.com/xss.php?cookie=' + escape(document.cookie);</script>
```

This code shows how a missing image can be used with an HTML attribute to trigger malicious code:

```
<img src="http://nofile" onerror="adr='http://example.com/xss.php?'+escape(document.cookie)";>
```

Any HTML from untrusted sources opens your site to XSS attacks. But the threat is only from certain characters.

DEFENDING AGAINST CROSS-SITE SCRIPTING

VALIDATE INPUT GOING TO THE SERVER

1. Only let visitors input the kind of characters they need to when supplying information. This is known as **validation**. Do not allow untrusted users to submit HTML markup or JavaScript.

2. Double-check validation on the server before displaying user content/storing it in a database. This is important because users could bypass validation in the browser by turning JavaScript off.

3. The database may safely contain markup and script from trusted sources (e.g., your content management system). This is because it does not try to process the code; it just stores it.

REQUESTS PAGES FROM AND SENDS FORM DATA TO WEB SERVER

COLLECTS INFORMATION FROM BROWSER AND PASSES IT TO DATABASE

STORES INFORMATION CREATED BY WEBSITE ADMINS AND USERS

BROWSER

WEB SERVER

DATABASE

PROCESSES HTML, CSS, AND JAVASCRIPT FILES SENT FROM WEB SERVER

GENERATES PAGES USING DATA FROM DATABASE AND INSERTS IT INTO TEMPLATES

RETURNS CONTENT NEEDED TO CREATE WEB PAGES

ESCAPE DATA COMING FROM THE SERVER & DATABASE

6. Do not create DOM fragments containing HTML from untrusted sources. It should only be added as text once it has been escaped.

5. Make sure that you are only inserting content generated by users into certain parts of the template files (see p230).

4. As your data leaves the database, all potentially dangerous characters should be escaped (see p231).

So, you can safely use `innerHTML` to add markup to a page if you have written the code – but content from any untrusted sources should be escaped and added as text (not markup), using properties like `textContent`.

XSS: VALIDATION & TEMPLATES

Make sure that your users can only input characters they need to use and limit where this content will be shown on the page.

FILTER OR VALIDATE INPUT

The most basic defense is to prevent users from entering characters into form fields that they do not **need** to use when providing that kind of information.

For example, users' names and email addresses will not contain angled brackets, ampersands, or parentheses, so you can validate data to prevent characters like this being used.

This can be done in the browser, but must also be done on the server (in case the user has JavaScript turned off). You learn about validation in Chapter 13.

LIMIT WHERE USER CONTENT GOES

Malicious users will not *just* use `<script>` tags to try and create an XSS attack. As you saw on p228, malicious code can live in an event handler attribute without being wrapped in `<script>` tags. XSS can also be triggered by malicious code in CSS or URLs.

Browsers process HTML, CSS, and JavaScript in different ways (or execution contexts), and in each language different characters can cause problems. Therefore, you should only add content from untrusted sources as text (not markup), and place that text in elements that are visible in the viewport.

You may have seen that the comment sections on websites rarely allow you to enter a lot of markup (they sometimes allow a limited subset of HTML). This is to prevent people from entering malicious code such as `<script>` tags, or any other character with an event handling attribute.

Even the HTML editors used in many content management systems will limit the code that you are allowed to use within them, and will automatically try to correct any markup that looks malicious.

Never place any user's content in the following places without detailed experience of the issues involved (which are beyond the scope of this book):

Script tags:	`<script>not here</script>`
HTML comments:	`<!-- not here -->`
Tag names:	`<notHere href="/test" />`
Attributes:	`<div notHere="norHere" />`
CSS values:	`{color: not here}`

XSS: ESCAPING & CONTROLLING MARKUP

Any content generated by users that contain characters that are used in code should be escaped on the server. You must control any markup added to the page.

ESCAPING USER CONTENT

All data from untrusted sources should be escaped on the server before it is shown on the page. Most server-side languages offer helper functions that will strip-out or escape malicious code.

HTML

Escape these characters so that they are displayed as characters (not processed as code).

&	`&`	'	`'` (not `'`)
<	`<`	"	`"`
>	`>`	/	`/`
`	```		

JAVASCRIPT

Never include data from untrusted sources in JavaScript. It involves escaping all ASCII characters with a value less than 256 that are not alphanumeric characters (and can be a security risk).

URLS

If you have links containing user input (e.g., links to a user profile or search queries), use the JavaScript `encodeURIComponent()` method to encode the user input. It encodes the following characters:

, / ? : @ & = + $ #

ADDING USER CONTENT

When you add untrusted content to an HTML page, once it has been escaped on the server, it should still be added to the page as text. JavaScript and jQuery both offer tools for doing this:

JAVASCRIPT

DO use: `textContent` or `innerText` (see p216)
DO NOT use: `innerHTML` (see p220)

JQUERY

DO use: `.text()` (see p316)
DO NOT use: `.html()` (see p316)

You can still use the `innerHTML` property and jQuery `.html()` method to add HTML to the DOM, but you must make sure that:

- You control **all** of the markup being generated (do not allow user content that could contain markup).
- The user's content is escaped and added as text using the approaches noted above, rather than adding the user's content as HTML.

ATTRIBUTE NODES

Once you have an element node, you can use other properties and methods on that element node to access and change its attributes.

There are two steps to accessing and updating attributes.

First, select the element node that carries the attribute and follow it with a period symbol.

Then, use one of the methods or properties below to work with that element's attributes.

Finds the element node (works with any technique covered in this chapter)

DOM QUERY

Gets the value of the attribute that was given as a parameter of the method

METHOD

```
document.getElementById('one').getAttribute('class');
```

MEMBER OPERATOR

Indicates that the subsequent method will be used on the node specified to the left

METHOD	DESCRIPTION
`getAttribute()`	gets the value of an attribute
`hasAttribute()`	checks if element node has a specified attribute
`setAttribute()`	sets the value of an attribute
`removeAttribute()`	removes an attribute from an element node

PROPERTY	DESCRIPTION
`className`	gets or sets the value of the **`class`** attribute
`id`	gets or sets the value of the **`id`** attribute

You have seen that the DOM treats each HTML element as its own object in the DOM tree. The properties of the object correspond to the attributes that type of element can carry. On the left, you can see the **`className`** and **`id`** properties. (Others include **`accessKey`**, **`checked`**, **`href`**, **`lang`**, and **`title`**.)

CHECK FOR AN ATTRIBUTE AND GET ITS VALUES

Before you work with an attribute, it is good practice to check whether it exists. This will save resources if the attribute cannot be found.

The `hasAttribute()` method of any element node lets you check if an attribute exists. The attribute name is given as an argument in the parentheses.

Using `hasAttribute()` in an `if` statement like this means that the code inside the curly braces will run only if the attribute exists on the given element.

JAVASCRIPT

c05/js/get-attribute.js

```
var firstItem = document.getElementById('one');     // Get first list item

if (firstItem.hasAttribute('class')) {              // If it has class attribute
    var attr = firstItem.getAttribute('class');     // Get the attribute

    // Add the value of the attribute after the list
    var el = document.getElementById('scriptResults');
    el.innerHTML = '<p>The first item has a class name: ' + attr + '</p>';

}
```

RESULT

In this example, the DOM query `getElementById()` returns the element whose `id` attribute has a value of one.

The `hasAttribute()` method is used to check whether this element has a `class` attribute, and returns a Boolean. This is used with an `if` statement so that the code in the curly braces will run only if the `class` attribute does exist.

The `getAttribute()` method returns the value of the `class` attribute, which is then written to the page.

Browser Support: Both of these methods have good support in all major web browsers.

CREATING ATTRIBUTES & CHANGING THEIR VALUES

The `class`Name property allows you to change the value of the `class` attribute. If the attribute does not exist, it will be created and given the specified value.

You have seen this property used throughout the chapter to update the status of the list items. Below, you can see another way to achieve the task.

The `setAttribute()` method allows you to update the value of *any* attribute. It takes two parameters: the attribute name, and the value for the attribute.

c05/js/set-attribute.js

JAVASCRIPT

```
var firstItem = document.getElementById('one'); // Get the first item
firstItem.className = 'complete';               // Change its class attribute

var fourthItem = document.getElementsByTagName('li').item(3);// Get fourth item
el2.setAttribute('class', 'cool');              // Add an attribute to it
```

When there is a property (like the `class`Name or `id` properties), it is generally considered better to update the properties rather than use a method (because, behind the scenes, the method would just be setting the properties anyway).

When you update the value of an attribute (especially the `class` attribute) it can be used to trigger new CSS rules, and therefore change the appearance of the elements.

RESULT

Note: These techniques override the entire value of the `class` attribute. They do not add a new value to the existing value of the `class` attribute.

If you wanted to add a new value onto the existing value of the `class` attribute, you would need to read the content of the attribute first, then add the new text to that existing value of the attribute (or use the jQuery `.addClass()` method covered on p320).

REMOVING ATTRIBUTES

To remove an attribute from an element, first select the element, then call `removeAttribute()`. It has one parameter: the name of the attribute to remove.

Trying to remove an attribute that does not exist will not cause an error, but it is good practice to check for its existence before attempting to remove it.

In this example, the `getElementById()` method is used to retrieve the first item from this list, which has an `id` attribute with a value of one.

JAVASCRIPT c05/js/remove-attribute.js

```
var firstItem = document.getElementById('one'); // Get the first item
if (firstItem.hasAttribute('class')) {          // If it has a class attribute
    firstItem.removeAttribute('class');         // Remove its class attribute
}
```

RESULT

The script checks to see if the selected element has a `class` attribute and, if so, it is removed.

EXAMINING THE DOM IN CHROME

Modern browsers come with tools that help you inspect the page loaded in the browser and understand the structure of the DOM tree.

In the screenshot to the right, the `<li>` element is highlighted and the ***Properties*** panel (1) indicates that this is an:

- `li` element with an `id` attribute whose value is `one` and `class` whose value is `hot`
- an HTMLLIElement
- an HTMLElement
- an element
- a node
- an object

Each of these object names has an arrow next to it which you can use to expand that section. It will tell you what properties are available to that kind of node.

They are separated because some properties are specific to list item elements, others to element nodes, others to all nodes, and others to all objects, and the different properties are listed under the corresponding type of node. But they do remind you of which properties you can access through the DOM node for that element.

To get the developer tools in Chrome on a Mac, go to the ***View*** menu, select ***Developer*** and then ***Developer Tools***. On a PC, go to ***Tools*** (or ***More Tools***) and select ***Developer Tools***.

Or right-click on any element and select ***Inspect Element***.

Select ***Elements*** from the menu that runs across the top of this tool. The source of the page will be shown on the left and several other options to the right.

Any element that has child elements has an arrow next to it that lets you expand and collapse the item to show and hide its content.

The ***Properties*** panel (on the right) tells you the type of object the selected element is. (In some versions of Chrome this is shown as a tab.) When you highlight different elements in the main left-hand window, you can see the values in the ***Properties*** panel on the right reflect that element.

EXAMINING THE DOM IN FIREFOX

Firefox has similar built-in tools, but you can also download a DOM inspector tool that shows the text nodes.

If you search online for "DOM Inspector", you will find the tool designed for Firefox shown on the left. In the screen shot, you can see a similar tree view to the one shown in Chrome, but it also shows you where there are whitespace nodes (they are shown as #text). In the panel to the right, you can see the value in the nodes; whitespace nodes have no value in this panel.

Another FIrefox extension worth trying is called Firebug.

Firefox also has a 3D view of the DOM, where a box is drawn around each element, and you can change the angle of the page to show which parts of it stick out more than others. The further they protrude the further into child elements they appear.

This can give you an interesting (and quick) glimpse into the complexity of the markup used on a page and the depth to which elements are nested.

LISTKING
BUY GROCERIES 6
kale
fresh figs
pine nuts
honey
balsamic vinegar

EXAMPLE
DOCUMENT OBJECT MODEL

This example brings together a selection of the techniques you have seen throughout the chapter to update the contents of the list. It has three main aims:

1: Add a new item to the start and end of the list

Adding an item to the *start* of a list requires the use of a different method than adding an element to the *end* of the list.

2: Set a `class` attribute on all items

This involves looping through each of the `<li>` elements and updating the value of the `class` attribute to `cool`.

3: Add the number of list items to the heading

This involves four steps:

1. Reading the content of the heading
2. Counting the number of `<li>` elements in the page
3. Adding the number of items to the content of the heading
4. Updating the heading with this new content

EXAMPLE
DOCUMENT OBJECT MODEL

c05/js/example.js

JAVASCRIPT

```
// ADDING ITEMS TO START AND END OF LIST
var list = document.getElementsByTagName('ul')[0];  // Get the <ul> element

// ADD NEW ITEM TO END OF LIST
var newItemLast = document.createElement('li');     // Create element
var newTextLast = document.createTextNode('cream'); // Create text node
newItemLast.appendChild(newTextLast);               // Add text node to element
list.appendChild(newItemLast);                      // Add element end of list

// ADD NEW ITEM START OF LIST
var newItemFirst = document.createElement('li');    // Create element
var newTextFirst = document.createTextNode('kale'); // Create text node
newItemFirst.appendChild(newTextFirst);             // Add text node to element
list.insertBefore(newItemFirst, list.firstChild);   // Add element to list
```

This part of the example adds two new list items to the <ul> element: one to the end of the list and one to the start of it. The technique used here is DOM manipulation and there are four steps to creating a new element node and adding it to the DOM tree:

1. Create the element node
2. Create the text node
3. Add the text node to the element node
4. Add the element to the DOM tree

To achieve step four, you must first specify the *parent* that will contain the new node. In both cases, this is the <ul> element. The node for this element is stored in a variable called `list` because it is used many times.

The `appendChild()` method adds new nodes as a child of the parent element. It has one parameter: the new content to be added to the DOM tree. If the parent element already has child elements, it will be added after the last of these (and will therefore be the last child of the parent element).

parent.appendChild(*newItem*);

(You have seen this method used several times both to add new elements to the tree and to add text nodes to element nodes.)

To add the item to the *start* of the list, the `insertBefore()` method is used. This requires one extra piece of information: the element you want to add the new content before (the target element).

parent.insertBefore(*newItem*, *target*);

EXAMPLE
DOCUMENT OBJECT MODEL

JAVASCRIPT c05/js/example.js

```
var listItems = document.querySelectorAll('li');        // All <li> elements

// ADD A CLASS OF COOL TO ALL LIST ITEMS
var i;                                                  // Counter variable
for (i = 0; i < listItems.length; i++) {                // Loop through elements
 listItems[i].className = 'cool';                       // Change class to cool
}

// ADD NUMBER OF ITEMS IN THE LIST TO THE HEADING
var heading = document.querySelector('h2');             // h2 element
var headingText = heading.firstChild.nodeValue;         // h2 text
var totalItems = listItems.length;                      // No. of <li> elements
var newHeading =  headingText + '<span>' + totalItems + '</span>'; // Content
heading.textContent = newHeading;                       // Update h2
```

The next step of this example is to loop through all of the elements in the list and update the value of their `class` attributes, setting them to `cool`.

This is achieved by first collecting all of the list item elements and storing them in a variable called `listItems`. A `for` loop is then used to go through each of them in turn. In order to tell how many times the loop should run, you use the `length` property.

Finally, the code updates the heading to include the number of list items. It updates it using the `innerHTML` property as opposed to the DOM manipulation techniques used earlier in the script.

This demonstrates how you can add to the content of an existing element by reading its current value and adding to it. You could use a similar technique if you needed to add a value to an attribute – without overwriting its existing value.

To update the heading with the number of items in the list, you need two pieces of information:

1. **The original content of the heading** so that you can add the number of list items to it. It is collected using the `nodeValue` property (although `innerHTML` or `textContent` would do the same).
2. **The number of list items**, which can be found using the `length` property on the `listItems` variable.

With this information ready, there are two steps to updating the content of the <h2> element:

1. **Creating the new heading** and storing it in a variable – the new heading will be made up of the original heading content, followed by the number of items in the list.
2. **Updating the heading**, which is done by updating the content of the heading element using the `innerText` property of that node.

SUMMARY

DOCUMENT OBJECT MODEL

- The browser represents the page using a DOM tree.
- DOM trees have four types of nodes: document nodes, element nodes, attribute nodes, and text nodes.
- You can select element nodes by their `id` or `class` attributes, by tag name, or using CSS selector syntax.
- Whenever a DOM query can return more than one node, it will always return a `NodeList`.
- From an element node, you can access and update its content using properties such as `textContent` and `innerHTML` or using DOM manipulation techniques.
- An element node can contain multiple text nodes and child elements that are siblings of each other.
- In older browsers, implementation of the DOM is inconsistent (and is a popular reason for using jQuery).
- Browsers offer tools for viewing the DOM tree.

6

EVENTS

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

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

INTERACTIONS CREATE EVENTS

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

EVENTS TRIGGER CODE

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

CODE RESPONDS TO USERS

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

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

DIFFERENT EVENT TYPES

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

UI EVENTS

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

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

KEYBOARD EVENTS

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

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

MOUSE EVENTS

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

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

TERMINOLOGY

EVENTS FIRE OR ARE RAISED

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

EVENTS TRIGGER SCRIPTS

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

FOCUS EVENTS

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

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

FORM EVENTS

Occur when a user interacts with a form element

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

MUTATION EVENTS*

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

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

HOW EVENTS TRIGGER JAVASCRIPT CODE

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

1

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

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

2

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

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

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

3

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

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

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

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

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

1

SELECT ELEMENT

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

2

SPECIFY EVENT

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

3

CALL CODE

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

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

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

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

THREE WAYS TO BIND AN EVENT TO AN ELEMENT

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

HTML EVENT HANDLERS

See p251

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

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

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

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

TRADITIONAL DOM EVENT HANDLERS

See p252

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

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

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

DOM LEVEL 2 EVENT LISTENERS

See p254

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

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

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

HTML EVENT HANDLER ATTRIBUTES (DO NOT USE)

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

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

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

HTML

c06/event-attributes.html

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

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

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

JAVASCRIPT

c06/js/event-attributes.js

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

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

For example:

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

TRADITIONAL DOM EVENT HANDLERS

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

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

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

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

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

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

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

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

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

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

The code starts by defining the named function.

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

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

USING DOM EVENT HANDLERS

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

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

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

JAVASCRIPT

c06/js/event-handler.js

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

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

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

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

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

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

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

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

EVENT LISTENERS

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

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

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

METHOD

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

ELEMENT
DOM element node to target

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

CODE
Name of function to call

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

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

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

The code starts by defining the named function.

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

The event name is enclosed in quotation marks.

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

USING EVENT LISTENERS

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

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

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

JAVASCRIPT c06/js/event-listener.js

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

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

The `addEventListener()` method takes three properties:

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

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

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

BROWSER SUPPORT

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

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

EVENT NAMES

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

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

USING PARAMETERS WITH EVENT HANDLERS & LISTENERS

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

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

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

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

Event name

Start of anonymous function

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

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

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

End of statement

End of `addEventListener()` method

Event flow Boolean (see p260)

End of anonymous function

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

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

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

USING PARAMETERS WITH EVENT LISTENERS

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

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

JAVASCRIPT

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

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

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

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

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

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

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

SUPPORTING OLDER VERSIONS OF IE

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

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

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

If the browser supports `addEventListener()`:

Run the code inside these curly braces

If it doesn't, do something else:

Run the code inside these curly braces

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

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

FALLBACK FOR USING EVENT LISTENERS IN IE8

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

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

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

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

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

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

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

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

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

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

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

EVENT FLOW

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

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

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

EVENT BUBBLING

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

EVENT CAPTURING

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

WHY FLOW MATTERS

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

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

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

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

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

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

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

THE EVENT OBJECT

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

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

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

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

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

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

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

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

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

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

EVENT LISTENER WITH NO PARAMETERS

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

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

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

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

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

EVENT LISTENER WITH PARAMETERS

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

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

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

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

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

THE EVENT OBJECT IN IE5-8

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

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

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

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

GETTING PROPERTIES

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

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

A FUNCTION TO GET THE TARGET OF AN EVENT

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

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

USING EVENT LISTENERS WITH THE EVENT OBJECT

Here is the example that has been used throughout the chapter so far with some modifications:

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

This function is now far more flexible than the previous code you have seen in this chapter because:

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

JAVASCRIPT

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

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

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

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

EVENT DELEGATION

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

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

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

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

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

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

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

ADDITIONAL BENEFITS OF EVENT DELEGATION

WORKS WITH NEW ELEMENTS

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

SOLVES LIMITATIONS WITH `this` KEYWORD

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

SIMPLIFIES YOUR CODE

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

CHANGING DEFAULT BEHAVIOR

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

preventDefault()

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

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

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

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

stopPropagation()

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

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

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

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

USING BOTH METHODS

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

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

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

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

USING EVENT DELEGATION

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

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

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

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

HTML

c06/event-delegation.html

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

JAVASCRIPT

c06/js/event-delegation.js

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

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

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

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

WHICH ELEMENT DID AN EVENT OCCUR ON?

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

THE `this` KEYWORD

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

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

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

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

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

USING PARAMETERS

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

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

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

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

DIFFERENT TYPES OF EVENTS

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

Events are defined in:

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

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

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

W3C DOM EVENTS

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

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

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

HTML5 EVENTS

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

```
submit
input
change
```

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

```
readystatechange
DOMContentLoaded
hashchange
```

BOM EVENTS

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

```
touchstart
touchend
touchmove
orientationchange
```

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

USER INTERFACE EVENTS

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

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

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

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

LOAD

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

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

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

JAVASCRIPT c06/js/load.js

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

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

RESULT

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

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

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

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

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

FOCUS & BLUR EVENTS

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

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

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

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

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

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

FOCUS & BLUR

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

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

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

JAVASCRIPT c06/js/focus-blur.js

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

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

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

RESULT

MOUSE EVENTS

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

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

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

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

WHEN TO USE CSS

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

WHY SEPARATE MOUSEDOWN & UP?

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

CLICK

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

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

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

JAVASCRIPT c06/js/click.js

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

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

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

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

RESULT

ACCESSIBILITY

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

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

WHERE EVENTS OCCUR

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

SCREEN

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

PAGE

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

CLIENT

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

DETERMINING POSITION

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

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

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

JAVASCRIPT c06/js/position.js

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

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

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

RESULT

KEYBOARD EVENTS

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

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

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

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

WHICH KEY WAS PRESSED?

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

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

WHICH KEY WAS PRESSED

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

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

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

JAVASCRIPT

c06/js/keypress.js

```
var el;                                                    // Declare variables

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

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

RESULT

FORM EVENTS

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

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

FOCUS AND BLUR

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

VALIDATION

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

USING FORM EVENTS

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

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

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

JAVASCRIPT

c06/js/form.js

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

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

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

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

MUTATION EVENTS & OBSERVERS

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

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

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

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

PROBLEMS WITH MUTATION EVENTS

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

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

NEW MUTATION OBSERVERS

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

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

USING MUTATION EVENTS

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

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

JAVASCRIPT

c06/js/mutation.js

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

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

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

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

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

RESULT

HTML5 EVENTS

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

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

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

USING HTML5 EVENTS

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

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

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

JAVASCRIPT

06/js/html5-events.js

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

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

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

RESULT

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

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

SAMSUNG
09:48
LISTKING
AUDIO NOTE
Enter note name:

EXAMPLE
EVENTS

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

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

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

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

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

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

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

EXAMPLE
EVENTS

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

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

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

c06/js/example.js

JAVASCRIPT

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

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

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

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

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

EXAMPLE
EVENTS

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

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

JAVASCRIPT

c06/js/example.js

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

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

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

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

SUMMARY

EVENTS

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

7

JQUERY

jQuery offers a simple way to achieve a variety of common JavaScript tasks quickly and consistently, across all major browsers and without any fallback code needed.

SELECT ELEMENTS

It is simpler to access elements using jQuery's CSS-style selectors than it is using DOM queries. The selectors are also more powerful and flexible.

PERFORM TASKS

jQuery's methods let you update the DOM tree, animate elements into and out of view, and loop through a set of elements, all in one line of code.

HANDLE EVENTS

jQuery includes methods that allow you to attach event listeners to selected elements without having to write any fallback code to support older browsers.

This chapter assumes that you have read the book up to this point or are familiar with the basics of JavaScript. As you will see, jQuery is powerful when combined with traditional JavaScript techniques, but you need to understand JavaScript to make full use of jQuery.

LISTKING
BUY GROCERIES
fresh figs
pine nuts
honey
balsamic vinegar

WHAT IS JQUERY?

jQuery is a JavaScript file that you include in your web pages. It lets you find elements using CSS-style selectors and then do something with the elements using jQuery methods.

1: FIND ELEMENTS USING CSS-STYLE SELECTORS

A function called **`jQuery()`** lets you find one or more elements in the page. It creates an object called **`jQuery`** which holds references to those elements. **`$()`** is often used as a shorthand to save typing **`jQuery()`**, as shown here.

FUNCTION (CREATES JQUERY OBJECT)

```
$('li.hot')
```

SELECTOR

The **`jQuery()`** function has one parameter: a CSS-style **selector**. This selector finds all of the **`<li>`** elements with a **`class`** of **`hot`**.

SIMILARITIES TO DOM

- jQuery selectors perform a similar task to traditional DOM queries, but the syntax is much simpler.
- You can store the **`jQuery`** object in a variable, just as you can with DOM nodes.
- You can use jQuery methods and properties (like DOM methods and properties) to manipulate the DOM nodes that you select.

The jQuery object has many methods that you can use to work with the elements you select. The methods represent tasks that you commonly need to perform with elements.

2: DO SOMETHING WITH THE ELEMENTS USING JQUERY METHODS

Here a **jQuery** object is created by the **jQuery()** function. The object and the elements it contains is referred to as a **matched set** or a **jQuery selection**.

You can then use the methods of the **jQuery** object to update the elements that it contains. Here, the method adds a new value to the **class** attribute.

JQUERY OBJECT

METHOD

```
$('li.hot').addClass('complete');
```

MEMBER OPERATOR

PARAMETER(S)

The member operator indicates that the method on the right should be used to update the elements in the **jQuery** object on the left.

Each method has parameter(s) that provide details about how to update the elements. This parameter specifies a value to add to the **class** attribute.

KEY DIFFERENCES FROM DOM

- It's cross-browser, and there's no need to write fallback code.
- Selecting elements is simpler (because it uses CSS-style syntax) and is more accurate.
- Event handling is simpler as it uses one method that works in all major browsers.
- Methods affect all the selected elements without the need to loop through each one (see p310).
- Additional methods are provided for popular required tasks such as animation (see p332).
- Once you have made a selection, you can apply multiple methods to it.

A BASIC JQUERY EXAMPLE

The examples in this chapter revisit the list application used in the previous two chapters, and they will use jQuery to update the content of the page.

1. In order to use jQuery, the first thing you need to do is include the jQuery script in your page. You can see that it is included before the closing `</body>` tag.

2. Once jQuery has been added to the page, a second JavaScript file is included that uses jQuery selectors and methods to update the content of the HTML page.

c07/basic-example.html

HTML

```
  <body>
    <div id="page"
      <h1 id="header">List</h1>
      <h2>Buy groceries</h2>
      <ul>
        <li id="one" class="hot"><em>fresh</em> figs</li>
        <li id="two" class="hot">pine nuts</li>
        <li id="three" class="hot">honey</li>
        <li id="four">balsamic vinegar</li>
      </ul>
    </div>
①   <script src="js/jquery-1.11.0.js"></script>
②   <script src="js/basic-example.js"></script>
  </body>
```

WHERE TO GET JQUERY AND WHICH VERSION TO USE

Above, jQuery is included before the closing `</body>` tag just like other scripts. (Another way to include the script is shown on p355.) A copy of jQuery is included with the code for this book, or you can download it from `http://jquery.org`. The version number of jQuery should be kept in the file name. Here, it is `jquery-1.11.0.js`, but by the time you read this book, there may be a newer version. The examples should still work with newer versions.

You often see websites use a version of the jQuery file with the file extension `.min.js`. It means unnecessary spaces and carriage returns have been stripped from the file. e.g., `jquery-1.11.0.js` becomes `jquery-1.11.0.min.js`.

It is done using a process called **minification** (hence `min` is used in the file name). The result is a much smaller file which makes it faster to download. But minified files are much harder to read.

If you want to look at the jQuery file, you can open it with a text editor – it is just text like JavaScript, albeit very complicated JavaScript.

Most people who use jQuery do not try to understand how the jQuery JavaScript file achieves what it does. As long as you know how to select elements and how to use its methods and properties, you can reap the benefits of using jQuery without looking under the hood.

Here, the JavaScript file uses the `$()` shortcut for the `jQuery()` function. It selects elements and creates three `jQuery` objects that hold references to the elements.

The methods of the `jQuery` object fade the list items in, and remove them when they are clicked on. Don't worry if you don't understand the code yet.

First, you will learn how to *select* elements using jQuery selectors, and then how to *update* those elements using the methods and properties of the `jQuery` object.

JAVASCRIPT

c07/js/basic-example.js

```
① $(':header').addClass('headline');
② $('li:lt(3)').hide().fadeIn(1500);
③ $('li').on('click', function() {
      $(this).remove();
  });
```

1. The first line selects all of the `<h1>` – `<h6>` headings, and adds a value of `headline` to their `class` attributes.

2. The second line selects the first three list items and does two things:

- The elements are hidden (in order to allow the next step).
- The elements fade into view.

3. The last three lines of the script set an event listener on each of the `<li>` elements. When a user clicks on one, it triggers an anonymous function to remove that element from the page.

RESULT

Here is a reminder of the colors used to convey the priority and status of each list item:

HOT | COOL
NORMAL | COMPLETE

WHY USE JQUERY?

jQuery doesn't do anything you cannot achieve with pure JavaScript. It is just a JavaScript file but estimates show it has been used on over a quarter of the sites on the web, because it makes coding simpler.

1: SIMPLE SELECTORS

As you saw in Chapter 5, which introduced the DOM, it is not always easy to select the elements that you want to. For example:

- Older browsers do not support the latest methods for selecting elements.
- IE does not treat whitespace between elements as text nodes, while other browsers do.

Such issues make it hard to select the right elements on a page across all major browsers.

Rather than learn a new way to select elements, jQuery uses a language that is already familiar to front-end web developers: CSS selectors. They:

- Are much faster at selecting elements
- Can be a lot more accurate about which elements to select
- Often require a lot less code than older DOM methods
- Are already used by most front-end developers

jQuery even adds some extra CSS-style selectors which offer additional functionality.

Since jQuery was created, modern browsers have implemented the `querySelector()` and `querySelectorAll()` methods to let developers select elements using CSS syntax. However, these methods are not supported in older browsers.

2: COMMON TASKS IN LESS CODE

There are some tasks that front-end developers need to do regularly, such as loop through the elements that have been selected.

jQuery has methods that offer web developers simpler ways to perform common tasks, such as:

- Loop through elements
- Add / remove elements from the DOM tree
- Handle events
- Fade elements into / out of view
- Handle Ajax requests

jQuery simplifies each of these tasks, and allows you to write less code to achieve them.

jQuery also offers chaining of methods (a technique which you will meet on p311). Once you have selected some elements, this allows you to apply multiple methods to the same selection.

jQuery's motto is "Write less, do more," because it allows you to achieve the same goals but in fewer lines of code than you would need to write with plain JavaScript.

3: CROSS-BROWSER COMPATIBILITY

jQuery automatically handles the inconsistent ways in which browsers select elements and handle events, so you do not need to write cross-browser fallback code (such as that shown in the previous two chapters).

To do this, jQuery uses **feature detection** to find the best way to achieve a task. It involves the use of many conditional statements: if the browser supports the ideal way to achieve a task, it uses that approach; otherwise, it tests to see if it supports the next best option to achieve the same task.

This was the technique used in the last chapter to determine whether or not the browser supported event listeners. If event listeners were not supported, an alternative approach was offered (aimed at users of Internet Explorer 8 and older versions of IE).

Here, a conditional statement checks if the browser supports `querySelector()`. If it does, that method is used. If it doesn't, it checks to see if the next best option is supported and uses that instead.

JQUERY 1.9.X+ OR 2.0.X+

As jQuery developed, it built up a lot of code to support IE6, 7, and 8; which made the script bigger and more complicated. As version 2.0 of jQuery was approaching, the development team decided to create a version that would drop support for older browsers in order to create a smaller, faster script.

The jQuery team was, however, aware that many people on the web still used these older browsers, and that developers therefore needed to support them. For this reason, they now maintain two parallel versions of jQuery:

jQuery 1.9+: Encompasses the same features as 2.0.x but still offers support for IE6, 7, and 8

jQuery 2.0+: Drops support for older browsers to make the script smaller and faster to use

The functionality of both versions is not expected to diverge significantly in the short term.

The jQuery file name should contain the version number in it (e.g., `jquery-1.11.0.js` or `jquery-1.11.0.min.js`). If you don't do this, a user's browser might try to use a cached version of the file that is either older or newer – which could prevent other scripts from working correctly.

FINDING ELEMENTS

Using jQuery, you usually select elements using CSS-style selectors. It also offers some extra selectors, noted below with a 'jQ'.

Examples of using these selectors are demonstrated throughout the chapter. The syntax will be familiar to those who have used selectors in CSS.

BASIC SELECTORS

`*`	All elements
`element`	All elements with that element name
`#id`	Elements whose **`id`** attribute has the value specified
`.class`	Elements whose **`class`** attribute has the value specified
`selector1, selector2`	Elements that match more than one selector (see also the **`.add()`** method, which is more efficient when combining selections)

HIERARCHY

`ancestor descendant`	An element that is a descendant of another element (e.g., **`li a`**)
`parent > child`	An element that is a direct child of another element (you can use * in the place of the child to select all child elements of the specified parent)
`previous + next`	Adjacent sibling selector only selects elements that are immediately followed by the previous element
`previous ~ siblings`	Sibling selector will select any elements that are a sibling of the previous element

BASIC FILTERS

`:not(`*`selector`***`)`**		All elements except the one in the selector (e.g., **`div:not('#summary')`**)
`:first`	jQ	The first element from the selection
`:last`	jQ	The last element from the selection
`:even`	jQ	Elements with an even index number in the selection
`:odd`	jQ	Elements with an odd index number in the selection
`:eq(`*`index`***`)`**	jQ	Elements with an index number equal to the one in the parameter
`:gt(`*`index`***`)`**	jQ	Elements with an index number greater than the parameter
`:lt(`*`index`***`)`**	jQ	Elements with an index number less than the parameter
`:header`	jQ	All **`<h1>`** – **`<h6>`** elements
`:animated`	jQ	Elements that are currently being animated
`:focus`		The element that currently has focus

CONTENT FILTERS

`:contains('text')`		Elements that contain the specified text as a parameter
`:empty`		All elements that have no children
`:parent`	jQ	All elements that have a child node (can be text or element)
`:has(selector)`	jQ	Elements that contain at least one element that matches the selector (e.g., `div:has(p)` matches all `div` elements that contain a `<p>` element)

VISIBILITY FILTERS

`:hidden`	jQ	All elements that are hidden
`:visible`	jQ	All elements that consume space in the layout of the page Not selected if: `display: none`; `height` / `width`: `0`; ancestor is hidden Selected if: `visibility: hidden`; `opacity: 0` because they would take up space in layout

CHILD FILTERS

`:nth-child(expr)`	The value here is not zero-based e.g. `ul li:nth-child(2)`
`:first-child`	First child from the current selection
`:last-child`	Last child from the current selection
`:only-child`	When there is only one child of the element (`div p:only-child`)

ATTRIBUTE FILTERS

`[attribute]`		Elements that carry the specified attribute (with any value)
`[attribute='value']`		Elements that carry the specified attribute with the specified value
`[attribute!='value']`	jQ	Elements that carry the specified attribute but not the specified value
`[attribute^='value']`		The value of the attribute begins with this value
`[attribute='value']`		The value of the attribute ends with this value
`[attribute*='value']`		The value should appear somewhere in the attribute value
`[attribute|='value']`		Equal to given string, or starting with string and followed by a hyphen
`[attribute~='value']`		The value should be one of the values in a space separated list
`[attribute] [attribute2]`		Elements that match all of the selectors

FORM

`:input`	jQ	All input elements
`:text`	jQ	All text inputs
`:password`	jQ	All password inputs
`:radio`	jQ	All radio buttons
`:checkbox`	jQ	All checkboxes
`:submit`	jQ	All submit buttons
`:image`	jQ	All `<img>` elements
`:reset`	jQ	All reset buttons
`:button`	jQ	All `<button>` elements
`:file`	jQ	All file inputs
`:selected`	jQ	All selected items from drop-down lists
`:enabled`		All enabled form elements (the default for all form elements)
`:disabled`		All disabled form elements (using the CSS `disabled` property)
`:checked`		All checked radio buttons or checkboxes

DOING THINGS WITH YOUR SELECTION

Once you have seen the basics of how jQuery works, most of this chapter is dedicated to demonstrating these methods.

These two pages both offer an overview to the jQuery methods and will also help you find the methods you are looking for once you have read the chapter.

You often see jQuery method names written starting with a period (.) before the name. This convention is used in this book to help you easily identify those methods as being jQuery methods rather than built-in JavaScript methods, or methods of custom objects.

When you make a selection, the **`jQuery`** object that is created has a property called **`length`**, which will return the number of elements in the object.

If the jQuery selection did not find any matching elements, you will not get an error by calling any of these methods – they just won't do or return anything.

There are also methods that are specifically designed to work with Ajax (which lets you refresh part of the page rather than an entire page) shown in Chapter 8.

CONTENT FILTERS

Get or change content of elements, attributes, text nodes

GET/CHANGE CONTENT

`.html()`	p316
`.text()`	p316
`.replaceWith()`	p316
`.remove()`	p316

ELEMENTS

`.before()`	p318
`.after()`	p318
`.prepend()`	p318
`.append()`	p318
`.remove()`	p346
`.clone()`	p346
`.unwrap()`	p346
`.detach()`	p346
`.empty()`	p346
`.add()`	p338

ATTRIBUTES

`.attr()`	p320
`.removeAttr()`	p320
`.addClass()`	p320
`.removeClass()`	p320
`.css()`	p322

FORM VALUES

`.val()`	p343
`.isNumeric()`	p343

FINDING ELEMENTS

Find and select elements to work with & traverse the DOM

GENERAL

`.find()`	p336
`.closest()`	p336
`.parent()`	p336
`.parents()`	p336
`.children()`	p336
`.siblings()`	p336
`.next()`	p336
`.nextAll()`	p336
`.prev()`	p336
`.prevAll()`	p336

FILTER/TEST

`.filter()`	p338
`.not()`	p338
`.has()`	p338
`.is()`	p338
`:contains()`	p338

ORDER IN SELECTION

`.eq()`	p340
`.lt()`	p340
`.gt()`	p340

Once you have selected the elements you want to work with (and they are in a `jQuery` object), the jQuery methods listed on these two pages perform tasks on those elements.

DIMENSION/POSITION

Get or update the dimensions or position of a box

DIMENSION

`.height()`	p348
`.width()`	p348
`.innerHeight()`	p348
`.innerWidth()`	p348
`.outerHeight()`	p348
`.outerWidth()`	p348
`$(document).height()`	p350
`$(document).width()`	p350
`$(window).height()`	p350
`$(window).width()`	p350

POSITION

`.offset()`	p351
`.position()`	p351
`.scrollLeft()`	p350
`.scrollTop()`	p350

EFFECTS & ANIMATION

Add effects and animation to parts of the page

BASIC

`.show()`	p332
`.hide()`	p332
`.toggle()`	p332

FADING

`.fadeIn()`	p332
`.fadeOut()`	p332
`.fadeTo()`	p332
`.fadeToggle()`	p332

SLIDING

`.slideDown()`	p332
`.slideUp()`	p332
`.slideToggle()`	p332

CUSTOM

`.delay()`	p332
`.stop()`	p332
`.animate()`	p332

EVENTS

Create event listeners for each element in the selection

DOCUMENT/FILE

`.ready()`	p312
`.load()`	p313

USER INTERACTION

`.on()`	p326

There used to be methods for individual types of event, so you may see methods such as `.click()`, `.hover()`, `.submit()`. However, these have been dropped in favour of the `.on()` method to handle events.

A MATCHED SET / JQUERY SELECTION

When you select one or more elements, a `jQuery` object is returned. It is also known as a **matched set** or a **jquery selection**.

SINGLE ELEMENT

If a selector returns one element, the **jQuery** object contains a reference to just one element node.

```
$('ul')
```

This selector picks the **<ul>** element from the page. So the **jQuery** object contains a reference to just one node (the only **<ul>** element in the page):

Each element is given an index number. Here there is just one element in the object.

INDEX	ELEMENT NODE
0	ul

MULTIPLE ELEMENTS

If a selector returns several elements, the **jQuery** object contains references to each element.

```
$('li')
```

This selector picks all the **<li>** elements. Here, the **jQuery** object has references for each of the nodes that was selected (each **<li>** element):

The resulting jQuery object contains four list items. Remember that index numbers start at zero.

INDEX	ELEMENT NODE
0	li#one.hot
1	li#two.hot
2	li#three.hot
3	li#four

JQUERY METHODS THAT GET AND SET DATA

Some jQuery methods both retrieve information from, and update the contents of, elements. But they do not always apply to all elements.

GET INFORMATION

If a jQuery selection holds more than one element, and a method is used to get information from the selected elements, it will **retrieve information from only the first element** in the matched set.

In the list example we have been using, the following selector chooses the four **<li>** elements from a list.

```
$('li')
```

When you use the **.html()** method (which will be introduced on p316) to get information from an element, it will return the content of the first element in the matched set.

```
var content = $('li').html();
```

This will retrieve the content of the first list item, and store it in the variable called **content**.

To get a different element, you can use methods to traverse (p336) or filter (p338) the selection, or write a more specific selector (p302).

To get the content of all of the elements, see the **.each()** method (p324).

SET INFORMATION

If a jQuery selection holds more than one element, and a method is used to update information on the page, it will **update all of the elements** in the matched set, not just the first one.

When you use the **.html()** method (which you meet on p316) to update the element, it will replace the contents of each element in the matched set. Here, it updates the content of each item in the list.

```
$('li').html('Updated');
```

This will update the content of all of the list items in the matched set with the word **Updated**.

To update just *one* element, you can use methods to traverse (p336) or filter (p338) the selection, or write a more specific selector (p302).

JQUERY OBJECTS STORE REFERENCES TO ELEMENTS

When you create a selection with jQuery, it stores a reference to the corresponding nodes in the DOM tree. It does not create copies of them.

As you have seen, when HTML pages load, the browser creates a model of the page in memory. Imagine your browser's memory is a set of tiles:

- < > Nodes in the DOM take up a tile
- var Variables take up a tile
- $ Complex JavaScript objects may take several tiles because they hold more data

In reality, the items in the browser's memory are not spread out as they are in this diagram, but the diagram helps explain the concept.

When you create a jQuery selection, the jQuery object holds **references** to the elements in the DOM – it does not create a copy of them.

When programmers say that a variable or object is storing a reference to something, what it is doing is storing the *location* a piece of information in the browser's memory. Here, the jQuery object would know that the list items are stored in A4, B4, and C4. Again, this is purely for illustration purposes; the browser's memory is not quite as simple as a checkerboard with these locations.

The jQuery object is an array-like object because it stores a list of the elements in the same order that they appear in the HTML document (unlike other objects where the order of the properties is not usually preserved).

CACHING JQUERY SELECTIONS IN VARIABLES

A jQuery object stores references to elements. Caching a jQuery object stores a reference to it in a variable.

To create a jQuery object takes time, processing resources, and memory. The interpreter must:

1. Find the matching nodes in the DOM tree
2. Create the jQuery object
3. Store references to the nodes in the jQuery object

So, if the code needs to use the same selection more than once, it is better to use that same jQuery object again rather than repeat the above process. To do this, you store a reference to the jQuery object in a variable.

Below, a jQuery object is created. It stores the locations of the `<li>` elements in the DOM tree.

```
$('li');
```

A reference to this object is in turn stored in a variable called `$listItems`. Note that when a variable contains a jQuery object, it is often given a name beginning with the `$` symbol (to help differentiate it from other variables in your script).

```
$listItems = $('li');
```

Caching jQuery selections is similar to the idea of storing a reference to a DOM node once you have made a DOM query (as you saw in Chapter 5).

LOOPING

In plain JavaScript, if you wanted to do the same thing to several elements, you would need to write code to loop through all of the elements you selected.

With jQuery, when a selector returns multiple elements, you can update all of them using the one method. There is no need to use a loop.

In this code, the same value is added to the `class` attribute for all of the elements that are found using the selector. It doesn't matter if there are one or many.

c07/js/looping.js

JAVASCRIPT

```
$('li em').addClass('seasonal');
$('li.hot').addClass('favorite');
```

In this example, the first selector applies only to one element and the `class` attribute's new value triggers a CSS rule that adds a calendar icon to the left of it.

The second selector applies to three elements. The new value added to the `class` attribute for each of these elements triggers a CSS rule that adds a heart icon on the right-hand side.

The ability to update all of the elements in the jQuery selection is known as **implicit iteration**.

When you want to get information from a series of elements, you can use the `.each()` method (which you meet on p324) rather than writing a loop.

RESULT

CHAINING

If you want to use more than one jQuery method on the same selection of elements, you can list several methods at a time using dot notation to separate each one, as shown below.

In this one statement, three methods act on the same selection of elements:
`hide()` hides the elements
`delay()` creates a pause
`fadeIn()` fades in the elements

The process of placing several methods in the same selector is referred to as **chaining**. As you can see, it results in code that is far more compact.

JAVASCRIPT

c07/js/chaining.js

```
$('li[id!="one"]').hide().delay(500).fadeIn(1400);
```

RESULT

To make your code easier to read, you can place each new method on a new line:

```
$('li[id!="one"]')
  .hide()
  .delay(500)
  .fadeIn(1400);
```

Each line starts with the dot notation, and the semicolon at the end of the statement indicates that you have finished working with this selection.

Most methods used to **update** the jQuery selection can be chained. However the methods that **retrieve** information from the DOM (or about the browser) cannot be chained.

It is worth noting that if one method in the chain does not work, the rest will not run either.

CHECKING A PAGE IS READY TO WORK WITH

jQuery's `.ready()` method checks that the page is ready for your code to work with.

`$(document)` creates a jQuery object representing the page.

When the page is ready, the function inside the parentheses of the **`.ready()`** method is run.

JQUERY OBJECT

READY EVENT METHOD

```
$(document).ready(function() {
  // Your script goes here
});
```

As with plain JavaScript, if the browser has not yet constructed the DOM tree, jQuery will not be able to select elements from it.

If you place a script at the end of the page (just before the closing **</body>** tag), the elements will be loaded into the DOM tree.

If you wrap your jQuery code in the method above, it will still work when used elsewhere on the page or even in another file.

A shorthand for this is shown on the right-hand page. It is more commonly used than this longer version.

THE load EVENT

jQuery had a **.load()** method. It fired on the **load** event, but has been replaced by the **.on()**. As you saw on p272, the **load** event fires after the page and all of its resources (images, CSS, and scripts) have loaded.

You should use this when your script relies on assets to have loaded, e.g., if it needs to know the dimensions of an image.

It works in all browsers, and also provides function-level scope for the variables it contains.

VS

THE .ready() METHOD

jQuery's **.ready()** method checks if the browser supports the **DOMContentLoaded** event, because it fires as soon as the DOM has loaded (it does not wait for other assets to finish loading) and can make the page appear as if it is loading faster.

If **DOMContentLoaded** is supported, jQuery creates an event listener that responds to that event. But the event is only supported in modern browsers. In older browsers, jQuery will wait for the **load** event to fire.

VS

PLACING SCRIPTS BEFORE THE CLOSING </body> TAG

When you place your script at the end of the page (before the closing **</body>** tag), the HTML will have loaded into the DOM before the script runs.

You will, however, still see people using the **.ready()** method because scripts that use it will still work if someone moves the script tag elsewhere in the HTML page. (This is particularly common when that script is being made available for other people to use.)

SHORTCUT FOR READY EVENT METHOD ON DOCUMENT OBJECT

```
$(function() {
  // Your script goes here
});
```

Above, you can see the shorthand that is commonly used instead of **$(document).ready()**

A positive side-effect of writing jQuery code inside this method is that it creates function-level scope for its variables.

This function-level scope prevents naming collisions with other scripts that might use the same variable names.

Any statements inside the method automatically run when the page has loaded. This is the version that will be used in the examples in the rest of the chapter.

GETTING ELEMENT CONTENT

The `.html()` and `.text()` methods both retrieve and update the content of elements. This page will focus on how to retrieve element content. To learn how to update element content, see p316.

.html()

When this method is used to retrieve information from a jQuery selection, it retrieves only the HTML inside the *first* element in the matched set, along with any of its descendants.

For example, `$('ul').html();` will return this:

```
<li id="one"><em>fresh</em> figs</li>
<li id="two">pine nuts</li>
<li id="three">honey</li>
<li id="four">balsamic vinegar</li>
```

Whereas `$('li').html();` will return this:

```
<em>fresh</em> figs
```

Note how this returns only the content of the first `<li>` element.

If you want to retrieve the value of every element, you can use the `.each()` method (see p324).

.text()

When this method is used to retrieve the text from a jQuery selection, it returns the content from every element in the jQuery selection, along with the text from any descendants.

For example, `$('ul').text();` will return this:

```
fresh figs
pine nuts
honey
balsamic vinegar
```

Whereas `$('li').text();` will return this:

```
fresh figspine nutshoneybalsamic vinegar
```

Note how this returns the text content of all `<li>` elements (including spaces between words), but there are no spaces between the individual list items.

To get the content from `<input>` or `<textarea>` elements, use the `.val()` method shown on p343.

GETTING AT CONTENT

On this page you can see variations on how the `.html()` and `.text()` methods are used on the same list (depending on whether `<ul>` or `<li>` elements are used in the selector).

Please note: The `.append()` method (covered on p318) lets you add content to the page.

JAVASCRIPT c07/js/get-html-fragment.js

```
var $listHTML = $('ul').html();
$('ul').append($listHTML);
```

The selector returns the `<ul>` element. The `.html()` method gets all the HTML inside it (the four `<li>` elements). This is then appended to the end of the selection, in this case after the existing `<li>` elements.

JAVASCRIPT c07/js/get-text-fragment.js

```
var $listText = $('ul').text();
$('ul').append('<p>' + $listText  + '</p>');
```

The selector returns the `<ul>` element. The `.text()` method gets the text from all of the `<ul>` element's children. This is then appended to the end of the selection, in this case after the existing `<ul>` element.

JAVASCRIPT c07/js/get-html-node.js

```
var $listItemHTML = $('li').html();
$('li').append('<i>' + $listItemHTML + '</i>');
```

The selector returns the four `<li>` elements, but the `.html()` method returns only the contents of the first one. This is then appended to the end of the selection, in this case after each existing `<li>` element.

JAVASCRIPT c07/js/get-text-node.js

```
var $listItemText = $('li').text();
$('li').append('<i>' + $listItemText + '</i>');
```

The selector returns the four `<li>` elements. The `.text()` method gets the text from these. This is then appended to each of the `<li>` elements in the selection.

UPDATING ELEMENTS

Here are four methods that update the content of all elements in a jQuery selection.

When the `.html()` and `.text()` methods are used as setters (to update content) they will replace the content of each element in the matched set (along with any content and child elements).

The `.replaceWith()` and `.remove()` methods replace and remove the elements they match (as well as their content and any child elements).

The `.html()`, `.text()`, and `.replaceWith()` methods can take a string as a parameter. The string can:

- Be stored in a variable
- Contain markup

When you add markup to the DOM, be sure to escape all untrusted content properly on the server. Both the `.html()` and `.replaceWith()` methods carry the same security risks as using the DOM's `innerHTML` property. See p228 - p231 on XSS.

.html()

This method gives every element in the matched set the same new content. The new content may include HTML.

.text()

This method gives every element in the matched set the same new text content. Any markup would be shown as text.

.replaceWith()

This method replaces every element in a matched set with new content. It also returns the replaced elements.

.remove()

This method removes all of the elements in the matched set.

USING A FUNCTION TO UPDATE CONTENT

If you want to use *and* amend the content of the current selection, these methods can take a function as a parameter. The function can be used to create new content. Here the text from each element is placed inside tags.

```
$('li.hot').html(function() {
    return '<em>' + $(this).text() + '</em>';
});
```

1. `return` indicates that content should be returned by the function.
2. tags are placed around the text content of the list item.
3. `this` refers to the current list item. `$(this)` places that element in a new jQuery object so that you can use jQuery methods on it.

CHANGING CONTENT

In this example, you can see three methods that allow you to update the content of the page.

When updating the content of an element, you can use a string, a variable, or a function.

JAVASCRIPT

c07/js/changing-content.js

```
$(function() {
  $('li:contains("pine")').text('almonds');
  $('li.hot').html(function() {
    return '<em>' + $(this).text() + '</em>';
  });
  $('li#one').remove();
});
```

1. This line selects any list items that contain the word `pine`. It then changes the text of the matching element to `almonds` using the `.text()` method.

2. These lines select all list items whose `class` attribute contains the word `hot`, and uses the `.html()` method to update the content of each of them.

The `.html()` method uses a function to place the content of each element inside an `<em>` element. (See the bottom of the left-hand page for a closer look at the syntax.)

3. This line selects the `<li>` element that has an `id` attribute whose value is `one`, then uses the `remove()` method to remove it. (This does not require a parameter.)

When specifying new content, carefully choose when to use single quotes and when to use double quotes. If you append a new element that has attributes, use single quotes to surround the content. Then use double quotes for the attribute values themselves.

RESULT

INSERTING ELEMENTS

Inserting new elements involves two steps:
1: Create the new elements in a jQuery object
2: Use a method to insert the content into the page

You can create new jQuery objects to hold text and markup that you then add to the DOM tree using one of the methods listed in step 2 on the right.

If you create a selection that returns multiple elements, these methods will add the same content to each of the elements in the matched set.

When adding content to the DOM, make sure you have escaped all untrusted content properly on the server. (See p228 - p231 on XSS.)

1: CREATING NEW ELEMENTS IN A JQUERY OBJECT

The following statement creates a variable called `$newFragment` and stores a jQuery object in it. The jQuery object is set to contain an empty `<li>` element: `var $newFragment = $('<li>');`

The following statement creates a variable called `$newItem` and stores a jQuery object in it. This jQuery object in turn contains an `<li>` element with a `class` attribute and some text:

```
var $newItem = $('<li class="new">item</li>');
```

2: ADDING THE NEW ELEMENTS TO THE PAGE

Once you have a variable holding the new content, you can use the following methods to add the content to the DOM tree:

.before()

This method inserts content before the selected element(s).

.after()

This method inserts content after the selected element(s).

.prepend()

This method inserts content inside the selected element(s), after the opening tag.

.append()

This method inserts content inside the selected element(s), before the closing tag.

There are also `.prependTo()` and `.appendTo()` methods. They work the other way around from `.prepend()` and `.append()`. So:

`a.prepend(b)` adds b to a
`a.prependTo(b)` adds a to b

`a.append(b)` adds b to a
`a.appendTo(b)` adds a to b

ADDING NEW CONTENT

In this example, you can see three jQuery selections are made. Each selection uses a different method to amend the content of the list.

The first adds a new notice before the list, the second adds a + symbol before the hot items, and the third adds a new element to the end of the list.

JAVASCRIPT

c07/js/adding-new-content.js

```
$(function() {
① $('ul').before('<p class="notice">Just updated</p>');
② $('li.hot').prepend('+ ');
③ var $newListItem = $('<li><em>gluten-free</em> soy sauce</li>');
  $('li:last').after($newListItem);
});
```

1. The `<ul>` element is selected, and the `.before()` method is used to insert a new paragraph before the list.

2. Selects all `<li>` elements whose `class` attribute contains a value of `hot` and uses the `.prepend()` method to add a plus symbol (+) before the text.

3. A new `<li>` element is created and stored in a variable. Then the last `<li>` element is selected, and the new element is added using the `.after()` method.

RESULT

GETTING AND SETTING ATTRIBUTE VALUES

You can create attributes, or access and update their contents, using the following four methods.

You can work with any attribute on any element using the `attr()` and `removeAttr()` methods.

If you use the `attr()` method to update an attribute that does not exist, it will create the attribute and give it the specified value.

The value of the `class` attribute can hold more than one class name (each separated by a space). The `addClass()` and `removeClass()` methods are very powerful because they let you add or remove an *individual* class name within the value of the `class` attribute (and they do not affect any other class names).

.attr()

This method can get or set a specified attribute and its value. To get the value of an attribute, you specify the name of the attribute in the parentheses.

```
$('li#one').attr('id');
```

To update the value of an attribute, you specify both the attribute name and its new value.

```
$('li#one').attr('id','hot');
```

.removeAttr()

This method removes a specified attribute (and its value). You just specify the name of the attribute that you want to remove from the element in the parentheses.

```
$('li#one').removeAttr('id');
```

.addClass()

This method adds a new value to the existing value of the `class` attribute. It does not overwrite existing values.

.removeClass()

This method removes a value from the `class` attribute, leaving any other class names within that attribute intact.

These two methods are another good example of how jQuery adds helpful functionality commonly needed by web developers.

WORKING WITH ATTRIBUTES

The statements in this example use jQuery methods to change the `class` and `id` attributes of the specified HTML elements.

When the values of these attributes change, new CSS rules are applied to the elements, changing how they look.

Using events to trigger changes to attribute values that apply new CSS rules is a popular way to make a web page interactive.

JAVASCRIPT

c07/js/attributes.js

```
  $(function() {
①   $('li#three').removeClass('hot');
②   $('li.hot').addClass('favorite');
③   $('ul').attr('id', 'group');
  });
```

1. The first statement finds the third list item (it has an `id` attribute with a value of `three`) and removes `hot` from the `class` attribute on that element. This is important to note because it affects the next statement.

2. The second statement selects all `<li>` elements whose `class` attribute has a value of `hot`. It adds a new class name called `favorite`. Because step 1 updated the third list item, this statement affects only the first two.

3. The third statement selects the `<ul>` element and adds an `id` attribute, giving it a value of `group` (which triggers a CSS rule that will add a margin and border to the `<ul>` element).

RESULT

GETTING & SETTING CSS PROPERTIES

The `.css()` method lets you retrieve and set the values of CSS properties.

To **get** the value of a CSS property, you indicate which property you want to retrieve in parentheses. If the matched set contains more than one element, it will return the value from the first element.

To **set** the values of a CSS property, you specify the property name as the first argument in the parentheses, then a comma, followed by its value as the second argument. This will update every element in the matched set. You can also specify multiple properties in the same method using object literal notation.

Note: In the method used to set an individual property, the property name and its value are separated by a comma (because all parameters in a method are separated by a comma).

In the object literal notation, properties and their values are separated by a colon.

HOW TO GET A CSS PROPERTY

This will store the background color of the first list item in a variable called `backgroundColor`. The color will be returned as an RGB value.

```
var backgroundColor = $('li').css('background-color');
```

HOW TO SET A CSS PROPERTY

This will set the background color of all list items. Note how the CSS property and its value are separated using a comma instead of a colon.

```
$('li').css('background-color', '#272727');
```

When dealing with dimensions that are specified in pixels, you can increase and decrease the values using the += and -= operators.

```
$('li').css('padding-left', '+=20');
```

SETTING MULTIPLE PROPERTIES

You can set multiple properties using **object literal notation**:

- Properties and values are placed in curly braces
- A colon is used to separate property names from their values
- A comma separates each pair (but there is not one after the last pair)

This sets the background color and typeface for all list items.

```
$('li').css({
  'background-color': '#272727',
  'font-family': 'Courier'
});
```

CHANGING CSS RULES

This example demonstrates how the `.css()` method can be used to select and update the CSS properties of elements.

The script checks what the background color of the first list item is when the page loads and then writes it after the list.

Next, it updates several CSS properties in all list items using the same `.css()` method with object literal notation.

JAVASCRIPT c07/js/css.js

```
$(function() {
  var backgroundColor = $('li').css('background-color');
  $('ul').append('<p>Color was: ' + backgroundColor + '</p>');
  $('li').css({
    'background-color': '#c5a996',
    'border': '1px solid #fff',
    'color': '#000',
    'font-family': 'Georgia',
    'padding-left': '+=75'
  });
});
```

1. The `backgroundColor` variable is created. The jQuery selection contains all `<li>` elements, and the `.css()` method returns the value of the `background-color` property of the first list item.

2. The background color of the first list item is written into the page using the `.append()` method (which you met on p318). Here, it is used to add content after the `<ul>` element.

3. The selector picks all `<li>` elements, and then the `.css()` method updates several properties at the same time:

- The background color is changed to brown
- A white border is added
- The color of the text is changed to black
- The typeface is changed to Georgia
- Extra padding is added on the left

RESULT

Note: It is better to change the value of a `class` attribute (to trigger new CSS rules in the style sheet) rather than to change CSS properties from within the JavaScript file itself.

WORKING WITH EACH ELEMENT IN A SELECTION

jQuery allows you to recreate the functionality of a loop on a selection of elements, using the `.each()` method.

You have already seen several jQuery methods that update all of the elements in a matched set without the need for a loop.

There are, however, times when you will want to loop through each of the elements in the selection. Often this will be to:

- Get information from each element in the matched set.
- Perform a *series* of actions on each of the elements.

The `.each()` method is provided for this purpose. The parameter of the `.each()` method is a function. This could be an anonymous function (as shown here) or a named function.

.each()

Allows you to perform one or more statements on each of the items in the selection of elements that is returned by a selector – rather like a loop in JavaScript.

It takes one parameter: a function containing the statements you want to run on each element.

```
①  ②
$('li').each(function() {
  var ids = this.id;
  $(this).append(' <em class="order">' + ids + '</em>');  ③
});
```

1. The jQuery selection contains all of the `<li>` elements.
2. `.each()` applies the same code to each element in the selection.
3. An anonymous function is run for each of the items in the list.

Since `this` refers to the current node, if you want to access a property of that node, e.g., that element's `id` or `class` attributes, it is better to use plain JavaScript to access those attributes:
`ids = this.id;`

this or $(this)

As the `.each()` method goes through the elements in a selection, you can access the current element using the `this` keyword.

You also often see `$(this)`, which uses the `this` keyword to create a new jQuery selection containing the current element. It allows you to use jQuery methods on the current element.

It is more efficient than writing `ids = $(this).attr('id');` because this would involve the interpreter creating a new jQuery object, and then using a method to access info that is available as a property.

USING .EACH()

This example creates a jQuery object containing all of the list items from the page.

The `.each()` method is then used to loop through the list items and run an anonymous function for each of them.

The anonymous function takes the value from the `id` attribute on the `<li>` element and adds it to the text in the list item.

JAVASCRIPT c07/js/each.js

```
$(function() {
  ① $('li').each(function() {
  ②   var ids = this.id;
  ③   $(this).append(' <span class="order">' + ids + '</span>');
  });
});
```

1. The selector creates a jQuery object containing all `<li>` elements. The `.each()` method calls an anonymous function for each of the list items in the matched set.

2. The `this` keyword refers to the current element node in the loop. It is used to access the value of the current element's `id` attribute, which is stored in a variable called `ids`.

3. `$(this)` is used to create a jQuery object that contains the current element in the loop.

Having the element in a jQuery object enables you to use jQuery methods on that element. In this case the `.append()` method is used to add a new `<span>` element to the current list item.

The content of that element is the value of its `id` attribute, which was obtained in step 2.

RESULT

EVENT METHODS

The `.on()` method is used to handle all events. Behind the scenes, jQuery handles all of the cross-browser issues you saw in the last chapter.

Using the `.on()` method is no different than using any other jQuery method; you:

- Use a selector to create a jQuery selection.
- Use `.on()` to indicate which event you want to respond to. It adds an event listener to each element in the selection.

`.on()` was introduced in v 1.7 of jQuery. Prior to that, jQuery used separate methods for each event, e.g., `.click()` and `.focus()`. You may come across them in older code, but you should only use the `.on()` method now.

```
①  ②  ③
$('li').on('click', function() {
  $(this).addClass('complete');  ④
});
```

1. The jQuery selection contains all of the `<li>` elements.
2. The `.on()` method is used to handle events. It needs two parameters:
3. The first parameter is the event you want to respond to. Here it is the `click` event.
4. The second parameter is the code you want to run when that event occurs on any element in the matched set. This could be a named function or an anonymous function. Above, it is an anonymous function that adds a value of `complete` to the `class` attribute.

You will see more advanced options for this method on p330.

JQUERY EVENTS

Some of the most popular events that `.on()` deals with are listed below. jQuery also added some extras to make life easier, such as `ready`, which fires when the page is ready to be worked with. These are noted with a pink asterisk: *

UI	`focus`, `blur`, `change`
KEYBOARD	`input`, `keydown`, `keyup`, `keypress`
MOUSE	`click`, `dblclick`, `mouseup`, `mousedown`, `mouseover`, `mousemove`, `mouseout`, `hover`*
FORM	`submit`, `select`, `change`
DOCUMENT	`ready`*, `load`, `unload`*
BROWSER	`error`, `resize`, `scroll`

EVENTS

In this example, when the mouse moves over a list item, the content of its `id` attribute is written into the list item.

The same happens if the user clicks on a list item (because `mouseover` does not work on touchscreen devices).

The `mouseout` event also removes this extra information from the page to prevent the added content building up.

JAVASCRIPT c07/js/events.js

```
$(function() {
  var ids = '';
① var $listItems = $('li');

  $listItems.on('mouseover click', function() {
    ids = this.id;
②   $listItems.children('span').remove();
    $(this).append(' <span class="priority">' + ids + '</span>');
  });

  $listItems.on('mouseout', function() {
③   $(this).children('span').remove();
  });

});
```

1. The selector finds all list items on the page. The resulting jQuery object is used more than once, so it is stored in a variable called `$listItems`.

2. The `.on()` method creates an event listener, which waits for when the user moves a mouse over a list item or clicks on it. It triggers an anonymous function.

Note how the two events are specified in the same set of quote marks, with a space between them.

The anonymous function:

- Gets the value of the `id` attribute on that element.
- Removes `<span>` elements from all of the list items.
- Adds the value of the `id` attribute to the list item in a new `<span>` element.

3. The `.mouseout()` method triggers the removal of any child `<span>` elements to prevent build-up of added values.

RESULT

THE EVENT OBJECT

Every event handling function receives an **event** object.
It has methods and properties related to the event that occurred.

Just like the JavaScript **event** object, the jQuery **event** object has properties and methods that tell you more about the event that took place.

If you look at the function that is called when the event occurs, the **event** object is named in the parentheses. Like any other parameter, this name is then used within the function to refer to the **event** object.

The example on the right uses the letter e as shorthand for the **event** object. However, as noted in the previous chapter, you should be aware that this shorthand is also often used for the **error** object.

```
                                  ①
$('li').on('click' function(e) {
  eventType = e.type;
});          ② ③
```

1. Give the **event** object a parameter name.
2. Use that name in the function to reference the **event** object.
3. Access the properties and methods of the object using the familiar dot notation (the member operator).

PROPERTY	DESCRIPTION
`type`	Type of event, (e.g., click, mouseover)
`which`	Button or key that was pressed
`data`	An object literal containing extra information passed to the function when the event fires (See right-hand page for an example)
`target`	DOM element that initiated the event
`pageX`	Mouse position from left edge of viewport
`pageY`	Mouse position from top of viewport
`timeStamp`	Number of milliseconds from Jan 1st, 1970, to when the event was triggered (this is known as Unix Time). Does not work in Firefox.

METHOD	DESCRIPTION
`.preventDefault()`	Prevents the default (e.g., submitting a form)
`.stopPropagation()`	Stops the event bubbling up to ancestors

EVENT OBJECT

In this example, when users click on a list item, the date that the event happened on is written next to that item, along with the type of event that triggered it.

To achieve this, two properties of the event object will be used: timeStamp states when the event occurred; type states the kind of event that triggered it.

To prevent the list from becoming cluttered with multiple date entries, whenever a list item is clicked, any elements will be removed from the list.

JAVASCRIPT

c07/js/event-object.js

```
$(function() {

  $('li').on('click' function(e) {
①   $('li span').remove();
②   var date = new Date();
②   date.setTime(e.timeStamp);
③   var clicked = date.toDateString();
④   $(this).append('<span class="date">' + clicked + ' ' + e.type + '</span>');
  });

});
```

1. Any elements that already exist inside the <li> elements are removed.

2. A new Date object is created, and its time is set to the time at which the event was clicked.

3. The time the event was clicked is then converted into a date that can be read.

4. The date that the list item was clicked is written into the list item (along with the type of event that was used).

Note that the timeStamp property does not display in Firefox.

RESULT

ADDITIONAL PARAMETERS FOR EVENT HANDLERS

The `.on()` method has two optional properties that let you:
Filter the initial jQuery selection to respond to a subset of the elements;
Pass extra information into the event handler using object literal notation.

Here you can see two additional properties that can be used with the `.on()` method.

When square brackets are used inside a method, they signify that the parameter is optional.

Leaving out a parameter written in square brackets will not stop the method working.

1. This is the event(s) that you want to respond to. If you want to respond to more than one event, you can provide a space-separated list of event names, e.g., `'focus click'` will work on both focus *and* click.

2. If you just want to respond to the event happening on a *subset* of the elements in the initial jQuery selection, you can provide a second selector that will filter its descendants.

3. You can pass extra information to the function that is called when the event is triggered. This information is passed along with the **event** object (e).

```
.on(events[, selector][, data], function(e));
```

① ② ③ ④ ⑤

4. This is the function that should be run when the specified events occur on one of the elements in the matched set.

5. The function is automatically passed the **event** object as a parameter, as you saw on the previous two pages. (Remember, if you use it you must give it a name in the parentheses.)

Older jQuery scripts may use the `.delegate()` method for delegation. However, since jQuery 1.7 `.on()` is the preferred approach to delegation.

DELEGATING EVENTS

In this example, the event handler will run when users click or mouseover items in the list, *except* for the last list item.

It writes out the content of the element the user interacted with, a status message (using the `data` property), and the event type.

The information passed in the `data` property here uses object literal notation (so it could handle multiple properties).

JAVASCRIPT c07/js/event-delegation.js

```
$(function() {
  var listItem, itemStatus, eventType;

  $('ul').on(
    'click mouseover',
    ':not(#four)',
    {status: 'important'},
    function(e) {
      listItem = 'Item: ' + e.target.textContent + '<br />';
      itemStatus = 'Status: ' + e.data.status + '<br />';
      eventType = 'Event: ' + e.type;
      $('#notes').html(listItem + itemStatus + eventType);
    }
  );

});
```

RESULT

There is an extra element in the HTML for this example to hold the data that appears under the list.

1. The event handler is triggered by `click` and `mouseover` events.

2. The `selector` parameter filters *out* the element whose `id` attribute has a value of `four`.

3. Additional data that will be used by the event handler is passed in as an object literal.

4. The event handler uses the `event` object to display the content of the element the user interacts with, the information from the data that was passed into the function, and the event type, under the list in a white box.

EFFECTS

When you start using jQuery, the effects methods can enhance your web page with transitions and movement.

Here you can see some of the jQuery effects that show or hide elements and their content. You can animate them fading in and out, or slide them up and down.

When an element that was previously hidden is shown, faded in, or slides into view, the other elements on the page may move to make space for it.

When an element is hidden, has been faded out, or has slid out of view, the other elements on the page can move into the space these elements took up.

Methods with `toggle` in their name will look at the current state of the element (whether it is visible or hidden) and will switch to the opposite state.

Increasingly it is possible to create animations using CSS3. They are often faster than their jQuery counterparts, but they only work in recent browsers.

BASIC EFFECTS

METHOD	DESCRIPTION
`.show()`	Displays selected elements
`.hide()`	Hides selected elements
`.toggle()`	Toggles between showing and hiding selected elements

FADING EFFECTS

METHOD	DESCRIPTION
`.fadeIn()`	Fades in selected elements making them opaque
`.fadeOut()`	Fades out selected elements making them transparent
`.fadeTo()`	Changes opacity of selected elements
`.fadeToggle()`	Hides or shows selected elements by changing their opacity (the opposite of their current state)

SLIDING EFFECTS

METHOD	DESCRIPTION
`.slideUp()`	Shows selected elements with a sliding motion
`.slideDown()`	Hides selected elements with a sliding motion
`.slideToggle()`	Hides or shows selected elements with a sliding motion (in the opposite direction to its current state)

CUSTOM EFFECTS

METHOD	DESCRIPTION
`.delay()`	Delays execution of subsequent items in queue
`.stop()`	Stops an animation if it is currently running
`.animate()`	Creates custom animations (see p334)

BASIC EFFECTS

In this example, it appears as if list items are faded into view when the page loads. Each item is faded out when it is clicked on. In fact, the items are loaded normally along with the rest of the page, but then immediately hidden using JavaScript. Once hidden, only then are they faded into view. This is so they will still be visible in browsers that do not have JavaScript enabled.

JAVASCRIPT c07/js/effects.js

```
$(function() {
  $('h2').hide().slideDown();
  var $li = $('li');
  $li.hide().each(function(index) {
    $(this).delay(700 * index).fadeIn(700);
  });
  $li.on('click', function() {
    $(this).fadeOut(700);
  });
});
```

1. In the first statement, the selector picks the <h2> element and hides it so that it can be animated in. The chosen effect to show the heading is the `.slideDown()` method. Note how the methods are chained; there is no need to make a new selection for each of the tasks.

2. The second part causes the list of items to appear one by one. Again, before they can be faded in, they must be hidden. Then the `.each()` method is used to loop through each of the <li> elements in turn. You can see that this triggers an anonymous function. Inside the anonymous function, the `index` property acts as a counter indicating which <li> element is the current one.

The `.delay()` method creates a pause before the list item is shown. The delay is set, multiplying the index number by 700 ms (otherwise all of the list items would appear at the same time). Then it is faded in using the `fadeIn()` method.

3. The final part creates an event listener that waits for the user to click on a list item. When they do, it will fade that item out to remove it from the list (the fade will take 700 milliseconds).

RESULT