WebKit DOM Programming Topics

User Experience

2010-01-20

Contents

Figures, Tables, and Listings

Introduction to WebKit DOM Programming Topics

> **Note:** This document was previous titled *Safari JavaScript Programming Topics*.

> **Important:** This is a preliminary document. Although it has been reviewed for technical accuracy, it is not final. Apple is supplying this information to help you adopt the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be vetted against final documentation. For information about updates to this and other developer documentation, you can check the ADC Reference Library Revision List. To receive notification of documentation updates, you can sign up for a free Apple Developer Connection Online membership and receive the bi-weekly ADC News e-mail newsletter. See http://developer.apple.com/products/for more details about ADC membership.)

JavaScript is a powerful interpreted scripting language designed for embedding into web-based applications. You can use the JavaScript Document Object Model (DOM) in Safari and the WebKit framework to help provide dynamic content to your users, whether you are designing web content, Dashboard widgets, or Cocoa applications.

Who Should Read This Document?

This document is designed for a number of different audiences:

- If you are a web content developer—developing web sites and embedded JavaScript applications—you should read about Safari's JavaScript support and how scripts operate within WebKit-based applications.

- If you are a Cocoa and WebKit developer, you should read about how to integrate JavaScript into your WebKit views and how to enhance your user experience in doing so.

- If you are a Dashboard developer, you should read about integrating JavaScript into your widgets to provide a better user experience and more advanced features to your users.

Organization of This Document

The topic contains the following articles:

- "About JavaScript and the DOM" (page 9) describes the JavaScript language and its implementation in Safari and WebKit.

- "Using the Document Object Model From JavaScript" (page 11) discusses the Document Object Model and how to use it from JavaScript.

- "Using the Canvas" (page 15) describes the canvas object, a critical part of Dashboard but also useful within other WebKit applications.

- "Using the Pasteboard From JavaScript" (page 19) describes how to implement copy and paste functionality within embedded JavaScript applications.

- "Using Drag and Drop From JavaScript" (page 21) describes how to use Mac OS X drag and drop functionality from within Safari and WebKit windows.

- "Using XMLHttpRequest Objects" (page 29) describes how to request data from network resources using the `XMLHttpRequest` object.

- "Calling Objective-C Methods From JavaScript" (page 45) describes how to use Objective-C in the JavaScript scripting environment, either within a WebKit object or by using a custom browser plug-in.

See Also

- *Safari HTML Reference* provides descriptions of HTML tags, attributes, and other markup.

- *Safari CSS Reference* provides descriptions of CSS properties and constants.

- *Safari Web Content Guide* provides information about designing web content for iPhone.

- *Dashboard Programming Topics* provides information on the technologies available to you when creating a Dashboard widget. Additional Dashboard documents and sample code can be found in the Reference Library > Apple Applications > Dashboard.

- *Apple JavaScript Coding Guidelines* provides general tips about the JavaScript programming language.

- The Reference Library > Apple Applications > Safari section of the ADC Reference Library provides useful information on WebKit, the technology that provides Apple's JavaScript runtime.

About JavaScript and the DOM

JavaScript is a platform-independent, object-oriented scripting language designed for the web, originally created by Netscape Communications (though the name is a trademark of Sun Microsystems). It was designed to add interactivity to web sites and has since grown into a fundamental tool for web content developers. JavaScript programs—called *scripts*—are usually embedded in HTML. Features like dynamic typing and event handling, and its interface with a web page's Document Object Model (DOM), all make JavaScript a very useful extension to HTML.

About JavaScript

JavaScript is not a compiled language; rather, it is interpreted during the parsing of an HTML page by a web client—it is not interpreted on the server side. Despite their similar names, JavaScript has no functional equivalence to the Java language; however, technologies like LiveConnect create interoperability between the two.

Scripts can be placed anywhere within an HTML file, but most commonly are placed in the `<head>` section, where the page's title and stylesheet definitions usually reside:

```
<head>
    <title>My Page</title>
    <!-- a script in another file -->
    <script language="JavaScript" TYPE="text/javascript"
        src="myscript.js"></script>

    <!-- a script inline -->
    <script language="JavaScript" TYPE="text/javascript"><!--

        function myFunction() {
            ...
        }
    // -->
    </script>
</head>
```

The script's content is enclosed in an HTML comment by convention. The `<script>` tag is notably the only tag in HTML whose contents are not supposed to be treated as content to display. Putting comments around these inline scripts ensures that older browsers (and non-browser tools) that do not understand the `<script>` tag do not mistake that content for ordinary text.

> ⚠️ **Warning:** In XHTML, data within the `<script>` tag is treated as parsed character data (`PCDATA`) instead of raw character data (`CDATA`). This means that special characters are parsed like ordinary HTML, and thus must be properly entity encoded. For example, instead of using `&`, you must use `&`.
>
> Because entity encoding makes scripts difficult to read, as a general rule, you should generally use external script files when working in XHTML. (As an added bonus, using separate script files reduces network bandwidth and improves page load performance because the scripts are cached independently.)

Apple's WebKit framework, and the Safari web browser based on it, both support the latest versions of JavaScript. Since the support is built into the framework, you can use all the features of JavaScript within anything that uses WebKit, including Safari, Dashboard, and any WebKit-based OS X application.

About the Document Object Model (DOM)

The Document Object Model (DOM) is a standardized software interface that allows code written in JavaScript and other languages to interact with the contents of an HTML document. The Document Object Model consists of a series of classes that represent HTML elements, events, and so on, each of which contains methods that operate on those elements or events.

With the Document Object Model, you can manipulate the contents of an HTML document in any number of ways, including adding, removing, and changing content, reading and altering the contents of a form, changing CSS styles (to hide or show content, for example), and so on.

By taking advantage of the Document Object Model, you can create much more dynamic websites that adapt as the user takes actions, such as showing certain form fields depending on selections in other fields, organizing your content based on what pages the viewer has recently visited, adding dynamic navigation features such as pull-down menus, and so on.

Using the Document Object Model From JavaScript

The JavaScript Document Object Model implements the Document Object Model (DOM) specification, developed by the World Wide Web Consortium. This specification provides a platform and language-neutral interface that allows programs and scripts to dynamically access and change the content, structure and style of a document —usually HTML or XML—by providing a structured set of objects that correspond to the document's elements.

The Level 2 DOM specification in particular added the ability to create object trees from style sheets, and manipulate the style data of document elements.

The Document Object Model is already implemented in a wide variety of languages, most notably JavaScript. Each declaration in the JavaScript DOM was created using the Interface Definition Language (IDL) files from the W3C.

By taking advantage of the DOM, you can:

■ Rearrange sections of a document without altering their contents

■ Create and delete existing elements as discrete objects

■ Search and traverse the document tree and the subtrees of any element on the tree

■ Completely isolate the document's structure from its contents

Accessing a Document's Structure with the DOM

The primary function of the Document Object Model is to view, access, and change the structure of an HTML document separate from the content contained within it. You can access certain HTML elements based on their `id` identifier, or allocate arrays of elements by their tag or CSS class type. All transformations are done according to the most recent HTML specification. More importantly, they happen dynamically—any transformation will happen without reloading the page.

The DOM tree is completely comprised of JavaScript objects. Some of the fundamental and most commonly-used ones are listed in Table 1.

Table 1   Commonly used JavaScript DOM types

DOM object	Definition
`document`	Returns the document object for the page. Represents the root node of the DOM tree, and actions on it will affect the entirety of the page.
`element`	Represents an instance of most structures and sub-structures in the DOM tree. For example, a text block can be an element, and so can the entire `body` section of an HTML document.

DOM object	Definition
nodeList	A `nodeList` is equivalent to an array, but it is an array specific to storing elements. You can access items in a nodeList through common syntax like `myList[n]`, where `n` is an integer.

There are a number of JavaScript methods specified by the DOM which allow you to access its structure. Some of the fundamental and most commonly-used ones are listed in Table 2.

Table 2   Commonly used JavaScript DOM methods

DOM method	Parent class(es)	Definition
`element getElementById(`*id*`)`	document, element	Returns the element uniquely identified by its `id` identifier.
`nodeList getElementsByTagName(`*name*`)`	document, element	Returns a `nodeList` of any elements that have a given tag (such as `p` or `div`), specified by *name*.
`element createElement(`*type*`)`	element	Creates an element with the type specified by *type* (such as `p` or `div`)
`void appendChild(`*node*`)`	element, node	Appends the node specified by *node* onto the receiving node or element.
`string style`	element	Returns the style rules associated with an element, in string form. You can also use this to set the style rules, by calling something like `div.style.margin = "10px";`
`string innerHTML`	element	Returns the HTML that contains the current element and all the content within it. You can also use this to set the innerHTML of an element, by using something like `element.innerHTML = "<p>test</p>";`
`void setAttribute(`*name*`, `*value*`)`	element	Adds (or changes) an attribute of the receiving element, such as its `id` or `align` attribute.
`string getAttribute(`*name*`)`	element	Returns the value of the element attribute specified by *name*.

The entire DOM reference can be found in "Other Resources" (page 14).

Using the Document Object Model

This section introduces some code examples to familiarize you with the Document Object Model.

To work with DOM code examples, you need to create a sample HTML file. The code below represents the HTML file you will use in the following examples:

```
<html>
<head>
    <title>My Sample HTML file</title>
    <script language="javascript" type="text/javascript"><!--
        // Insert JavaScript
        // commands here.

        // Do not delete this comment. -->
    </script>
</head>
<body>
 <div id="allMyParas" style="border-top: 1px #000 solid;">
    <p id="firstParagraph">
        This is my first paragraph.
    </p>
    <p id="secondParagraph">
        This is my second paragraph.
    </p>
 </div>
</body>
</html>
```

Now that you have an HTML document to work with, you're ready to add some DOM transformations into the script area. This first example appends a paragraph to the DOM tree, following the firstParagraph and secondParagraph elements:

```
var parasDiv = document.getElementById("allMyParas");
var thirdPara = document.createElement("p");
thirdPara.setAttribute("id", "thirdParagraph");
parasDiv.appendChild(thirdPara);
```

Of course, the paragraph has no text in it right now. Using the DOM, you can even add text to that new paragraph:

```
thirdPara.innerHTML = "This is my third paragraph.";
```

You may also want to add a bottom margin to the enclosing div element, where there is currently only a top margin. You could do that with setAttribute, but you would have to copy the existing style attribute with getAttribute, append to it, and then send it back using setAttribute. Instead, use the style block:

```
parasDiv.style.borderBottom = "1px #000 solid";
```

Finally, maybe you want to change the style on all the paragraph elements within the enclosing div element. You can use the nodeList-generating getElementsByTagName method to get an array of the paragraph elements, and then cycle through them. In this example, we'll add a gray background to all the paragraphs:

```
var parasInDiv = parasDiv.getElementsByTagName("p");
for (var i = 0; i < parasInDiv.length; i++) {
    parasInDiv[i].style.backgroundColor = "lightgrey";
}
```

When you get done, the HTML effectively looks like this:

```
<html>
<head>
    <title>My Sample HTML file</title>
</head>
<body>
```

```
<div id="allMyParas" style="border-top: 1px #000 solid; border-bottom: 1px #000
solid;">
    <p id="firstParagraph" style="backgroundColor: lightgrey">
        This is my first paragraph.
    </p>
    <p id="secondParagraph" style="backgroundColor: lightgrey">
        This is my second paragraph.
    </p>
    <p id="thirdParagraph" style="backgroundColor: lightgrey">
        This is my third paragraph.
    </p>
</div>
</body>
</html>
```

> **Note:** For illustration purposes, in the above block of HTML, the styles shown include the changes made to the CSS style for the elements. In actuality, the text content of the each element's style attribute remains unchanged, though the effect is equivalent.

The combination of the JavaScript Document Object Model with WebKit-based applications or Dashboard widgets is powerful. By tying these scripts to mouse events or button clicks, for example, you can create very dynamic and fluid content on your web page or within your WebKit-based applications, including Dashboard widgets.

Other Resources

The following resources will help you use the JavaScript Document Object Model:

- *WebKit DOM Reference* is Apple's reference for the JavaScript functions and properties supported by Safari and WebKit.

- Mozilla Gecko DOM Reference is one of the most comprehensive references for the JavaScript DOM.

Using the Canvas

Safari, Dashboard, and WebKit-based applications support the JavaScript *canvas* object. The canvas allows you to easily draw arbitrary content within your HTML content.

Introduction to the Canvas

A canvas is an HTML tag that defines a custom drawing region within your web content. You can then access the canvas as a JavaScript object and draw upon it using features similar to Mac OS X's Quartz drawing system. The World Clock Dashboard widget (available on all Apple machines running Mac OS X version 10.4 or later) shows a good example , though using a canvas is by no means exclusive to Dashboard.

There are two steps to using a canvas in your web page: defining a content area, and drawing to the canvas object in the script section of your HTML.

Defining the Canvas

To use a canvas in your web page you first set up the drawing region. The World Clock Dashboard widget designates this region with the following code:

```
<canvas id="myCanvas" width='172' height='172'></canvas>
```

In this context, the attributes of `<canvas>` worth noting are `id`, `width`, and `height`.

The `id` attribute is a custom identifier used to target a particular canvas object when drawing. The `width` and `height` attributes specify the size of the canvas region.

Within the World Clock widget, this area is defined to be the canvas:

Figure 1   The World Clock canvas region

Now that the canvas region has been defined, it is ready to be filled.

Drawing on a Canvas

Once you have defined the canvas area, you can write code to draw your content. Before you can do this, you need to obtain the canvas and its drawing context. The context handles the actual rendering of your content. The World Clock widget does this in its `drawHands()` function:

```
function drawHands (hoursAngle, minutesAngle, secondsAngle)
{
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
```

This function draws the hour, minute, and second hands on the face of the World Clock. As parameters, it takes the angles at which the three hands should be rotated as passed in by its caller.

You first query the JavaScript environment for the previously defined canvas, using its unique identifier: the `id` attribute in the `<canvas>` tag.

Once your script has acquired the canvas, you need to obtain its context. Using the `getContext("2d")` method, assign the canvas' draw context it to the `context` variable. From this point on, you call all operations intended for the canvas on `context`.

The first operation you perform empties the canvas. As the `drawHands()` function is called every second, it is important to empty it each time, so that the previously drawn configuration doesn't simply draw on top of the new configuration. The entire region, as defined by standard coordinates in the `<canvas>` tag, is cleared:

```
context.clearRect(0, 0, 172, 172);
```

Next, you save the state of the original context space so that you can restore later. In the original context, the origin (the 0,0 coordinate) of the canvas is in the top left corner. Upon completion of the upcoming drawing code, you want to return to this context. Use the context's save method to do so:

```
context.save();
```

Since you want the hands of the clock to rotate around the center of the clock, translate the origin of the context space to the center of the canvas:

```
context.translate(172/2, 172/2);
```

Then draw the hour hand on the face of the clock. You copy the current context (with the origin at the center of the clock face), so that it can restored later. Then, you rotate the entire context, so that the y-axis aligns itself with the angle that the hour hand should point towards. Next, you draw the hour hand image (created in the code as a JavaScript `Image` object). The method `drawImage()` has five parameters: the image to be drawn, the x and y coordinate for the bottom left hand corner of the image, and the width and height of the image. Remember that while you draw the image as going straight up within the graphics context, you rotated the context to be at the correct angle for the hour hand:

```
context.save();
context.rotate(hoursAngle);
context.drawImage(hourhand, -4, -28, 9, 25);
context.restore();
```

Once you draw the hand, you restore the last saved context. This means that the context that you saved four lines prior, with its origin at the center of the canvas but not yet rotated, will be the active context again.

Use a similar procedure to draw the minute hand on the face of the clock. The differences this time are in the angle you rotate the context to and the size of the minute hand. Note that you save and rotate the context again, and then you restore it to its previous state, so that you can draw the next element independent of the rotation needed for the minute hand:

```
context.save();
context.rotate (minutesAngle);
context.drawImage (minhand, -8, -44, 18, 53);
context.restore();
```

Finally, draw the second hand. Note that this time, the context should not be saved and restored. Since this is the last time anything will be drawn in this particular context (with the origin at the center of the canvas), it is not necessary for you to save and restore again:

```
context.rotate (secondsAngle);
context.drawImage (sechand, -4, -52, 8, 57);
context.restore();
}
```

Now that the clock face has been drawn, you should restore the context to its original state, as saved before any drawing occurred. This prepares the canvas for any future drawing that will occur, and gives you a consistent origin (the top-left corner of the canvas) to work from.

Remember, all of these techniques can be applied to a canvas object within any WebKit-based application. For more information on the canvas see `Canvas`.

Accessing and Manipulating Canvas Data

Safari 4.0 and later support direct manipulation of the pixels of a canvas. You can obtain the raw pixel data of a canvas with the `getImageData()` function and create a new buffer for manipulated pixels with the `createImageData()` function:

```
var canvas = document.getElementById("myCanvas");
var context = canvas.getContext("2d");
var currentPixels = context.getImageData(0, 0, canvas.width, canvas.height);
var newPixels = context.createImageData(canvas.width, canvas.height);
```

The `getImageData()` and `createImageData()` functions both return an `ImageData` object that contains pixel information in its `data` property. The `data` property is an array that contains four values between `0` and `255` for every pixel in the canvas. These values correspond to the red, green, blue, and alpha components of the pixel. The following example loops through the `data` array of the `currentPixels` variable. For each pixel, it subtracts each color component's value from `255` and assigns the result to the corresponding cell in the `data` array of the `newPixels` variable, thus creating an inverted copy of the image. The alpha value of every pixel is copied over without any manipulation. Finally, the `newPixels` variable is passed to the `putImageData()` function, which applies the new pixel values to the canvas.

```
for (var y = 0; y < newPixels.height; y += 1) {
    for (var x = 0; x < newPixels.width; x += 1) {
        for (var c = 0; c < 3; c += 1) {
            var i = (y*newPixels.width + x)*4 + c;
            newPixels.data[i] = 255 - currentPixels.data[i];
        }
        var alphaIndex = (y*newPixels.width + x)*4 + 3;
```

```
        newPixels.data[alphaIndex] = currentPixels.data[alphaIndex];
    }
}
context.putImageData(newPixels, 0, 0);
```

You can also obtain a data URL representation of the image of a canvas with the `toDataURL()` function. This function returns a PNG encoding of the image when no parameters are passed. You can obtain other image formats by passing in a string indicating the desired format, such as `image/jpg` or `image/gif`.

Using the Pasteboard From JavaScript

Safari, Dashboard, and WebKit-based applications include support to let you handle cut, copy, and paste operations of your HTML content.

Introduction to JavaScript Pasteboard Operations

Support for pasteboard operations is implemented in JavaScript and may be applied to any element of your HTML page. To handle these operations, you provide functions to handle any of six JavaScript events:

- `onbeforecut`
- `oncut`
- `onbeforecopy`
- `oncopy`
- `onbeforepaste`
- `onpaste`

You can provide handlers for the `oncut`, `oncopy`, and `onpaste` events if you want to define custom behavior for the corresponding operations. You can also provide handlers for the `onbeforecut`, `onbeforecopy`, and `onbeforepaste` events if you want to manipulate the target data before it is actually cut, copied, or pasted.

If your `oncut`, `oncopy`, and `onpaste` handlers do the actual work of cutting, copying, or pasting the data, your handler must call the `preventDefault` method of the event object. This method takes no parameters and notifies WebKit that your handler takes care of moving the data to or from the pasteboard. If you do not call this method, WebKit takes responsibility for moving the data. You do not need to call `preventDefault` if you simply want to be notified when the events occur.

Adding Pasteboard Handlers to Elements

You can add handlers for pasteboard events to any element in a web page. When a pasteboard operation begins, WebKit looks for the appropriate handler on the element that is the focus of the operation. If that element does not define a handler, WebKit walks up the list of parent elements until it finds one that does. (If no element defines a handler, WebKit applies the default behavior.) To demonstrate this process, suppose you have the following basic HTML in a web page:

```
<body oncut="MyBodyCutFunction()"
      oncopy="MyBodyCopyFunction()"
      onpaste="MyBodyPasteFunction()">
    <span onpaste="MySpanPasteFunction()">Cut, copy, or paste here.</span>
</body>
```

If a user initiates a cut or copy operation on the text in the `span` tag, WebKit calls `MyBodyCutFunction` or `MyBodyCopyFunction` to handle the event. However, if the user tries to paste text into the span tag, WebKit calls the `MySpanPasteFunction` to handle the event. The `MyBodyPasteFunction` function would be called only if the paste operation occurred outside of the `span` tag.

Manipulating Pasteboard Data

When an event occurs, your handler uses the `clipboardData` object attached to the event to get and set the clipboard data. This object defines the `clearData`, `getData`, and `setData` methods to allow you to clear, get, and set the clipboard data.

> **Note:** For security purposes, the `getData` method can be called only from within the `onpaste` event handler.

WebKit's pasteboard implementation supports data types beyond those that are typically found in HTML documents. When you call either `getData` or `setData`, you specify the MIME type of the target data. For types it recognizes, including standard types found in HTML documents, WebKit maps the type to a known pasteboard type. However, you can also specify MIME types that correspond to any custom data formats your application understands. For most pasteboard operations, you will probably want to work with simple data types, such as plain text or a list of URIs.

WebKit also supports the ability to post the same data to the pasteboard in multiple formats. To add another format, you simply call `setData` once for each format, specifying the format's MIME type and a string of data that conforms to that type.

To get a list of types currently available on the pasteboard, you can use the `types` property of the `clipboardData` object. This property contains an array of strings with the MIME types of the available data.

Using Drag and Drop From JavaScript

Safari, Dashboard, and WebKit-based applications include support for customizing the behavior of drag and drop operations within your HTML pages.

> **Note:** This technology is supported only on desktop versions of Safari. For iPhone OS, use DOM Touch, described in Handling Events (part of *Safari Web Content Guide*) and *Safari DOM Additions Reference*.

Introduction to JavaScript Drag and Drop

Support for Drag and Drop operations is implemented in JavaScript and may be applied to individual elements of your HTML page. For drag operations, an element can handle the following JavaScript events:

- `ondragstart`
- `ondrag`
- `ondragend`

The `ondragstart` event initiates the drag operation. You can provide a handler for this event to initiate or cancel drag operations selectively. To cancel a drag operation, call the `preventDefault` method of the event object. To handle an event, assign a value to the `effectAllowed` property and put the data for the drag in the `dataTransfer` object, which you can get from the event object. See "Changing Drag Effects" (page 23) for information on the `effectAllowed` property. See "Manipulating Dragged Data" (page 23) for information on handling the drag data.

Once a drag is under way, the `ondrag` event is fired continuously at the element to give it a chance to perform any tasks it wants to while the drag is in progress. Upon completion of the operation, the element receives the `ondragend` event. If the drag was successful, the `ondrop` handler for the drop target element is also called (before the `ondragend` handler is called).

While a drag is in progress, events are sent to elements that are potential drop targets for the contents being dragged. Those elements can handle the following events:

- `ondragenter`
- `ondragover`
- `ondragleave`
- `ondrop`

The `ondragenter` and `ondragleave` events let the element know when the user's mouse enters or leaves the boundaries of the element. You can use these events to change the cursor or provide feedback as to whether a drop can occur on an element. The `ondragover` event is sent continuously while the mouse is over the element to give it a chance to perform any needed tasks. If the user releases the mouse button, the element receives an `ondrop` event, which gives it a chance to incorporate the dropped content.

If you implement handlers for the `ondragenter` and `ondragover` events, you should call the `preventDefault` method of the event object. This method takes no parameters and notifies WebKit that your handler will act as the receiver of any incoming data. If you do not call this method, WebKit receives the data and incorporates it for you. You do not need to call `preventDefault` if you simply want to be notified when the events occur.

> **Note:** You must, at minimum, implement `ondragover` and call the `preventDefault` method on the event object. If you do not do this, you will not receive any of these four events.

Adding Handlers to Elements

You can add handlers for drag and drop events to any element in a web page. When a drag or drop operation occurs, WebKit looks for the appropriate handler on the element that is the focus of the operation. If that element does not define a handler, WebKit walks up the list of parent elements until it finds one that does. If no element defines a handler, WebKit applies the default behavior. To demonstrate this process, suppose you have the following basic HTML in a web page:

```
<body ondragstart="BodyDragHandler()"
      ondragend="BodyDragEndHandler()">
    <span ondragstart="SpanDragHandler()">Drag this text.</span>
</body>
```

If a user initiates a drag operation on the text in the `span` tag, WebKit calls `SpanDragHandler` to handle the event. When the drag operation finishes, WebKit calls the `BodyDragEndHandler` to handle the event.

Making an Element Draggable

WebKit provides automatic support to let users drag common items, such as images, links, and selected text. You can extend this support to include specific elements on an HTML page. For example, you could mark a particular `div` or `span` tag as draggable.

To mark an arbitrary element as draggable, add the `-webkit-user-drag` attribute (previously `-khtml-user-drag`) to the style definition of the element. Because it is a cascading style sheet (CSS) attribute, you can include it as part of a style definition or as an inline style attribute on the element tag. The values for this attribute are listed in Table 1.

Table 1   Values for -webkit-user-drag attribute

Value	Description
none	Do not allow this element to be dragged.

Value	Description
`element`	Allow this element to be dragged.
`auto`	Use the default logic for determining whether the element should be dragged. (Images, links, and text selections can be dragged but all others cannot.) This is the default value.

The following example shows how you might use this attribute in a `span` tag to permit the dragging of the entire tag. When the user clicks on the `span` text, WebKit identifies the span as being draggable and initiates the drag operation.

```
<span style="color:rgb(22,255,22); -webkit-user-drag:element;">draggable
text</span>
```

Manipulating Dragged Data

When an event occurs, your handler uses the `dataTransfer` object attached to the event to get and set the clipboard data. This object defines the `clearData`, `getData`, and `setData` methods to allow you to clear, get, and set the data on the dragging pasteboard.

> **Note:** For security purposes, the `getData` method can be called only from within the `ondrop` event handler.

Unlike many other browsers, the WebKit drag-and-drop implementation supports data types beyond those that are found in HTML documents. When you call either `getData` or `setData`, you specify the MIME type of the target data. For types it recognizes, WebKit maps the type to a known pasteboard type. However, you can also specify MIME types that correspond to any custom data formats your application understands. For most drag-and-drop operations, you will probably want to work with simple data types, such as plain text or a list of URIs.

Like applications, WebKit supports the ability to post the same data to the pasteboard in multiple formats. To add another format, you simply call the `setData` method with a different MIME type and a string of data that conforms to that type.

To get a list of types currently available on the pasteboard, you can use the `types` property of the `dataTransfer` object. This property contains an array of strings with the MIME types of the available data.

Changing Drag Effects

When dragging content from one place to another, it might not always make sense to move that content permanently to the destination. You might want to copy the data or create a link between the source and destination documents instead. To handle these situations, you can use the `effectAllowed` and `dropEffect` properties of the `dataTransfer` object to specify how you want data to be handled.

The `effectAllowed` property tells WebKit what types of operation the source element supports. You would typically set this property in your `ondragstart` event handler. The value for this property is a string, whose value can be one of those listed in Table 2.

Table 2   Options for dragging and dropping an element

Value	Description
none	No drag operations are allowed on the element.
copy	The contents of the element should be copied to the destination only.
link	The contents of the element should be shared with the drop destination using a link back to the original.
move	The element should be moved to the destination only.
copyLink	The element can be copied or linked.
copyMove	The element can be copied or moved. This is the default value.
linkMove	The element can be linked or moved.
all	The element can be copied, moved, or linked.

The `dropEffect` property specifies the single operation supported by the drop target (`copy`, `move`, `link`, or `none`). When an element receives an `ondragenter` event, you should set the value of this property to one of those values, preferably one that is also listed in the `effectAllowed` property. If you do not specify a value for this property, WebKit chooses one based on the available operations (as specified in `effectAllowed`). Copy operations have priority over move operations, which have priority over link operations.

When these properties are set by the source and target elements, WebKit displays feedback to the user about what type of operation will occur if the dragged element is dropped. For example, if the dragged element supports all operations but the drop target only supports copy operations, WebKit displays feedback indicating a copy operation would occur.

Changing the Appearance of Dragged Elements

During a drag operation, WebKit provides feedback to the user by displaying an image of the dragged content under the mouse. The default image used by WebKit is a snapshot of the element being dragged, but you can change this image to suit your needs.

Changing the Snapshot With CSS

The simplest way to change the drag-image appearance is to use cascading style sheet entries for draggable elements. WebKit defines the `-webkit-drag` (formerly `-khtml-drag`) pseudoclass, which you can use to modify the style definitions for a particular class during a drag operation. To use this pseudoclass, create a new empty style-sheet class entry with the name of the class you want to modify, followed by a a colon and the string `-webkit-drag`. In the style definition of this new class, change or add attributes to specify the differences in appearance between the original element and the element while it is being dragged.

The following example shows the style-sheet definition for an element. During normal display, the appearance of the element is determined by the style-sheet definition of the `divSrc4` class. When the element is dragged, WebKit changes the background color to match the color specified in the `divSrc4:-webkit-drag` pseudoclass.

```
#divSrc4 {
    display:inline-block;
    margin:6;
    position:relative;
    top:20px;
    width:100px;
    height:50px;
    background-color:rgb(202,232,255);
}

#divSrc4:-webkit-drag {
    background-color:rgb(255,255,154)
}
```

Specifying a Custom Drag Image

Another way to change the drag image for an element is to specify a custom image. When a drag operation begins, you can use the `setDragImage` method of the `dataTransfer` object. This method has the following definition:

```
function setDragImage(image, x, y)
```

The `image` parameter can contain either a JavaScript Image object or another element. If you specify an Image object, WebKit uses that image as the drag image for the element. If you specify an element, WebKit takes a snapshot of the element you specify (including its child elements) and uses that snapshot as the drag image instead.

The `x` and `y` parameters of `setDragImage` specify the point of the image that should be placed directly under the mouse. This value is typically the location of the mouse click that initiated the drag, with respect to the upper-left corner of the element being manipulated.

Unfortunately, obtaining this information in a cross-browser fashion is easier said than done. There is no standard way to determine the position of the mouse relative to the document because different browsers implement the standard event values in subtly incompatible ways.

For the purposes of Safari and WebKit, `clientX` and `clientY` are document relative, as are `pageX` and `pageY` (which are thus always equal to `clientX` and `clientY`). For other browsers, Evolt.org has an article that describes how to obtain the mouse position in a cross-browser fashion, including sample code, at http://evolt.org/article/Mission_Impossible_mouse_position/17/23335/index.html.

Obtaining the position of the element under the mouse is somewhat easier. QuirksMode has a page (with code samples) on the subject at http://www.quirksmode.org/js/findpos.html.

Cross-Browser Compatibility

The drag-and-drop functionality built into WebKit and Safari works similarly to support in Microsoft Internet Explorer. The functionality built into FireFox and other Gecko-based browsers is very different, however.

In currently released versions of FireFox, this form of drag and drop is not generally supported from ordinary webpages (signed XUL applications notwithstanding) because you cannot register a drop target without loading an XPCOM component. Thus, with the exception of dropping things onto text areas (which are already drop targets), drag and drop must be emulated on this browser using mouse event handlers such as `onmouseup`.

As a result of differences in browser support, most web developers who need drag-and-drop support use libraries that mask browser differences. Some of these include the Dojo Toolkit, DOM-Drag, ToolMan, Rico, and others.

Complete Example

No description of drag and drop would be complete without a working example. Save this into an HTML file and open it in Safari. You should see a very simple set of boxes containing words. If you drag each word box into the blue "target" box, the box will disappear and the word will appear in its correct place to form the phrase "This is a test".

```
<html>
<head>

<script language="javascript" type="text/javascript"><!--
    var dragitem = undefined;

    function setdragitem(item, evt) {
        dragitem=item;
        // alert('item: '+item);
        // item is an HTML DIV element.
        // evt is an event.

        // If the item should not be draggable, enable this next line.
        // evt.preventDefault();

        return true;
    }
    function cleardragitem() {
        dragitem=undefined;
        // alert('item: '+item);
    }
    function dodrag() {
        // alert('item: '+dragitem);
    }

    // This is required---used to tell WebKit that the drag should
    // be allowed.
    function handledragenter(elt, evt) {
        evt.preventDefault();
        return true;
```

```
        }
    function handledragover(elt, evt) {
        evt.preventDefault();
        return true;
    }


    function handledragleave(elt, evt) {

    }

    function handledrop(elt, evt) {
        // alert('drop');
        dragitem.style.display="none";
        var newid=dragitem.id + '_dest';
        var dest = document.getElementById(newid);
        dest.innerHTML = dragitem.innerHTML;
    }


// --></script>

<style type="text/css"><!--

    .wordbox { border: 1px solid black; text-align: center; width: 50px; float:
 left; -webkit-user-drag: element; -webkit-user-select: none; }
    .spacer { clear: both; }
    .target { margin-top: 30px; padding: 30px; width: 70px; border: 1px solid
black; background: #c0c0ff; margin-bottom: 30px; -webkit-user-drop: element; }
    .word   { margin: 30px; min-height: 30px; border-bottom: 1px solid black;
width: 50px;  float: left; }

--></style>

</head>
<body>

<p>Drop words onto target area to put them in their places.</p>

<div class='wordbox' id='this' ondragstart='setdragitem(this, event);'
ondrag='dodrag();' ondragend='cleardragitem();'>This</div>
<div class='wordbox' id='is' ondragstart='setdragitem(this, event);'
ondrag='dodrag();' ondragend='cleardragitem();'>is</div>
<div class='wordbox' id='a' ondragstart='setdragitem(this, event);'
ondrag='dodrag();' ondragend='cleardragitem();'>a</div>
<div class='wordbox' id='test' ondragstart='setdragitem(this, event);'
ondrag='dodrag();' ondragend='cleardragitem();'>test</div>

<div class='spacer'></div>
<div class='target' ondragenter='handledragenter(this, event);'
ondragover='handledragover(this, event);' ondragleave='handledragleave(this,
event);' ondrop='handledrop(this, event);'>TARGET</div>

<div class='words'>
<div class='word' id='this_dest'></div>
<div class='word' id='is_dest'></div>
<div class='word' id='a_dest'></div>
<div class='word' id='test_dest'></div>
```

```
</div>


</body>
</html>
```

Using XMLHttpRequest Objects

Safari, Dashboard, and WebKit-based applications support the JavaScript `XMLHttpRequest` object. `XMLHttpRequest` (often abbreviated as XHR) allows you to easily fetch content from a server and use it within your webpage or widget without requiring a page reload.

For example, you can use XHR to update a store search page so that when the user enters a Zip code, your site can display stores within a reasonable distance. Clicking on a store could then fill a particular content div with details on that store.

Of course, you could do any of these things without XHR (by requiring a full page load or by including information about every store in the country in a single page), but XHR provides a way to do this more efficiently with less bandwidth.

Introduction to XMLHttpRequest

`XMLHttpRequest` is a JavaScript object provided by WebKit that fetches data via HTTP for use within your JavaScript code. It's tuned for retrieving XML data but can be used to perform any HTTP request. XML data is made available in a DOM object that lets you use standard DOM operations, as discussed in "Using the Document Object Model From JavaScript" (page 11), to extract data from the request response.

Typically, you define an `XMLHttpRequest` object's options and provide an `onload` or `onreadystatechange` handler, then send the request. When the request is complete, you work with either the request's response text or its response XML, as discussed in "XMLHttpRequest Responses" (page 30).

Defining an XMLHttpRequest Object

To create a new instance of the `XMLHttpRequest` object, call the object's constructor with the `new` keyword and save the result in a variable, like this:

```
var myRequest = new XMLHttpRequest();
```

> **Note:** If you are writing a webpage, you should be aware that most versions of Microsoft Internet Explorer on Windows do not support creating an `XMLHttpRequest` object in this way. Jibbering.com has cross-browser sample code at http://jibbering.com/2002/4/httprequest.html if you need to support Internet Explorer prior to version 7.

After you have created a new `XMLHttpRequest` object, call `open` to initialize the request:

```
myRequest.open("GET", "http://www.apple.com/");
```

The `open` method requires two arguments: the HTTP method and the URI of the data to fetch. It also can take three more arguments: an asynchronous flag, a username, and a password. By default, `XMLHttpRequest` executes asynchronously.

After you open the request, use `setRequestHeader` to provide any optional HTTP headers for the request, like this:

```
myRequest.setRequestHeader("Cache-Control", "no-cache");
```

> **Note:** This particular header asks web caches between the browser and the server to not serve the request from a cache. Not all caches respect these flags, however, and some browsers do not consistently respect it, either.
>
> This can be problematic if, for example, you send a request in an `onChange` handler on a form field. If that request can be cached, any request that changes the field back to a previous value won't ever reach the server, resulting in the UI not matching the actual values stored on the server.
>
> Thus, if it is absolutely essential that a request not be served from a cache, you should err on the side of caution by adding a timestamp or other nonrecurring value to the end of each URL. For example: `http://mysite.mydomain.top/file.html?junktimevalue=1187999959`.

To handle the different states of a request, set a handler function for the `onreadystatechange` event:

```
myRequest.onreadystatechange = myReadyStateChangeHandlerFunction;
```

If the only state you're concerned about is the `loaded` state (state 4), try using the `onload` event instead:

```
myRequest.onload = myOnLoadHandlerFunction;
```

If you are sending a cross-site request (see "Using XMLHttpRequest for Cross-Site Requests" (page 31) for more information) and want cookies or authentication information to be sent to the remote server, you need to enable credentials:

```
myRequest.withCredentials = "true";
```

When the request is ready, use the `send` method to send it:

```
myRequest.send();
```

If your request is sending content, like a string or DOM object, pass it in as the argument to the `send` method.

XMLHttpRequest Responses

Once your send your request, you can abort it using the `abort` method:

```
myRequest.abort();
```

If you provided an `onreadystatechange` handler, you can query your request to find its current state using the readyState property:

```
var myRequestState = myRequest.readyState;
```

A `readyState` value of 4 means that content has loaded. This is similar to providing an `onload` handler, which is called when a request's `readyState` equals 4.

When a request is finished loading, you can query its HTTP status using the `status` and `statusText` properties:

```
var myRequestStatus = myRequest.status;
var myRequestStatusText = myRequest.statusText;
```

Also, you can fetch the request's HTTP response headers using the `getResponseHeader` method:

```
var aResponseHeader = myRequest.getResponseHeader("Content-Type");
```

To obtain a list of all of the response headers for a request, use `getAllResponseHeaders`:

```
var allResponseHeaders = myRequest.getAllResponseHeaders();
```

To obtain the request's response XML as a DOM object, use the responseXML property:

```
var myResponseXML = myRequest.responseXML;
```

This object responds to standard DOM methods, like `getElementsByTagName`. If the response is not in a valid XML format, use the `responseText` property to access the raw text response:

```
var myResponseText = myRequest.responseText;
```

Security Considerations

Within Safari, the `XMLHttpRequest` object can only make requests to `http` and `https` URIs in the same domain as the webpage.

As an exception, however, beginning with Safari 4.0, a website can allow content served by other websites to request data from specific `http` and `https` URLs on their servers.

> **Note:** To enable this behavior, the site must send specific response headers conforming to the Cross-Origin Resource Sharing specification, defined by the Web Applications Working Group. It is not possible to make HTTP requests to arbitrary sites without the cooperation of those sites.

Using XMLHttpRequest for Cross-Site Requests

For the most part, the `XMLHttpRequest` object works the same way when used in a cross-site fashion. There are a few exceptions, however.

- cookies—Because the JavaScript code is not in the same domain, it is not possible to set or read cookies for that domain.

  > **Note:** Although the cookie fields of the XMLHttpRequest object are ignored for cross-site requests, cookies can still be sent and received if you specifically set the `withCredentials` attribute on the request object to `"true"`. See "Defining an XMLHttpRequest Object" (page 29) for more information.

- authentication tokens—As with cookies, it is not possible to add or read authentication information for the remote domain.

> **Note:** Although authentication information cannot be provided by the script, authenticated HTTP requests are allowed; the user is asked to enter login credential information upon initial connection just as though the user had connected to the site manually.
>
> In order for a request to be sent using any login credentials, however, you must explicitly set the `withCredentials` attribute on the request object to `"true"`. See "Defining an XMLHttpRequest Object" (page 29) for more information.

- `Referer` header—Because it can contain confidential path information, the `Referer` header is not sent to the remote site. You can, however, check the `Origin` header to determine the domain name of the referring page.

- other headers—Most HTTP headers are explicitly scrubbed both when sending and receiving. The only headers that are sent are:

  - ❏ `Accept`—a comma-separated list of Internet Media Type values (as defined in RFC 1590) that are acceptable as a response for the request. These may be tagged with quality values to indicate a preference for one type over another.

    For example, Accept: `text/html, text/plain;q=.5` indicates that the caller prefers HTML data with an implied quality of `1`, but will also accept plain text data with an explicit quality of `0.5`

    The second field may be replaced with an asterisk (*) wildcard character (`Accept: text/*`, for example). The first field may also be a wildcard if and only if the second field contains a wildcard (`Accept: */*`).

  - ❏ `Accept-Language`—a comma-separated list of natural languages preferred by the user, optionally tagged with quality values for each language. The language tag values are defined by RFC 1766.

    For example, `Accept-Language: de, en-gb;q=.5, es` indicates the user prefers German or Spanish with implied quality of `1`, but will tolerate British English with an explicit quality of `0.5`.

  - ❏ `Content-Language`—the natural language of the content of the request (`en-US`, for example), as defined by RFC 1766.

  - ❏ `Content-Type`—the Internet Media Type for the data in the request (`Content-Type: text/html`, for example), as defined in RFC 1590.

  - ❏ `Origin`—sent by the browser to indicate the domain from which the request was sent. This field contains the non-path portion of the URL (`http://example.com`, for example).

    > **Note:** The `Origin` header is only sent when requesting the specified URI. If the cross-origin request returns a redirect to a different URI, when fetching the subsequent location or locations, the Origin header is set to `null` and the Referer header field is set to the previous location in the redirect chain.

The only headers your script can expect to receive from the server (after the browser filters them) are:

- ❏ `Host`—provides the host name and port number of the server sending a response.

- ❏ `Cache-Control`—provides caching policy information.

- ❏ `Content-Language`—a comma-separated list of natural language of the content (`en-US`, for example), as defined by RFC 1766.

- ❏ `Content-Type`—the Internet Media Type for the data in the response (`Content-Type: text/html`, for example), as defined in RFC 1590.

❑ `Expires`—the date and time at which any cache of this data should be discarded. The date format is specified by RFC 1123.

❑ `Last-Modified`—the most recent date and time when the document's contents were modified. The date format is specified by RFC 1123.

❑ `Pragma`—used for optional implementation-specific directives that may apply to intermediate web caches, the browser, and so on.

All other headers may be scrubbed.

> ⚠️ **Warning:** Data requested from a remote site should be treated as untrusted. For example, you should not execute JavaScript code retrieved from a remote site without thoroughly checking it for validity.

Access Control Request Headers

Before performing the actual request, the browser first sends an `OPTIONS` query to the server to ask if it is allowed to send the actual request. The following headers are sent by the browser as part of this query and may be used by scripts on the remote web server when deciding whether or not to allow the request:

`Origin`

Contains the non-path portion of the resource making the request (for example, `http://example.com`).

`Access-Control-Request-Method`

Contains the method that the actual request will use (`GET` or `POST`, for example).

`Access-Control-Request-Headers`

Indicates which HTTP headers will be sent in the actual request.

Access Control Response Headers

Before performing the actual request, the browser first sends an `OPTIONS` query to the server to ask if it is allowed to send the actual request. The following headers may be sent by the remote web server (or scripts running on that server) in response to this request to control cross-site access policies:

`Access-Control-Allow-Origin`

Indicates that a particular origin can access the resource. The value may be either a specific origin (`http://example.com`, for example) or an asterisk (`*`) wildcard, which allows any domain to access the resource.

`Access-Control-Max-Age`

Indicates how long a browser should continue to cache the site's acceptance of requests from other domains. If your site only allows requests from other domains after some specific handshake by a user, for example, you might set this to a short value.

`Access-Control-Allow-Credentials`

Indicates that the request may be made with credentials. The only allowable value is `true`. To disallow credentials, omit this header.

`Access-Control-Allow-Methods`

Provides a request method that the script may use for future requests (`GET` or `POST`, for example). The valid methods are defined in RFC 2616. This field may be specified more than once to allow multiple methods.

`Access-Control-Allow-Headers`

Provides a header field name that the client has permission to use in the actual request.

Cross-Document Messaging

As a general rule, scripts loaded by web content served from one origin (host and domain) cannot access web content served by a different origin. This is an important security feature that prevents a multitude of different security attack vectors. However, it also makes it difficult for scripts to interact with one another across these boundaries.

To make communication between documents from different origins easier, the HTML 5 specification adds cross-document messaging. This feature is supported in Safari 4.0 and later.

Posting a Message to a Window

To post a message, you must first obtain the `Window` object of the document you want to message. In effect, this means that you can post messages only to:

- other frames or inline frames within your document window (or their descendants if all intermediate frames or inline frames were served from the same origin).

    ```
    var iFrameObj = document.getElementById('myId');
    var windowObj = iFrameObj.contentWindow;
    ```

- windows that your document explicitly opened through JavaScript calls.

    ```
    var windowObj = window.open(...);
    ```

- the window that contains your document window, the window that contains that window, and so on up to the root window.

    ```
    var windowObj = window.parent;
    ```

- the window that opened your document.

    ```
    var windowObj = window.opener;
    ```

Once you have obtained the `Window` object for the target document, you can send it a message with the following code:

```
windowObj.postMessage('test message', 'http://example.com');
```

The first parameter is an arbitrary message.

The second parameter is the target origin value. An origin value is just a URL with the path part removed. For example, the origin of a local file is `file://`. By specifying a target origin, you are saying that that your message should only be delivered if the target window's current contents came from that origin.

Although you may specify an asterisk (*) wildcard for the target origin (to allow the message to be sent regardless of where the contents of the target window came from), you should do so only if you are certain that it would not be harmful if your message were received by content originating from a different website.

Receiving a Message Posted to a Window

To receive messages, you must add an event listener for the `message` event type to your document's `window` object. To do this, use the following code:

```
function messageReceive(evt) {
    if (evt.origin == 'http://example.com') {
        // The message came from an origin that
        // your code trusts.  Work with the message.
        alert('Received data '+evt.data);

        // Send a message back to the source:
        evt.source.postMessage('response', evt.origin);
    } else {
        alert('unexpected message from origin '+evt.origin);
    }
}
window.addEventListener('message', messageReceive, false);
```

The `message` event you receive has three properties of interest:

- `data`—the message contents.

- `origin`—the domain from which the message was sent (`http://example.com` in this case).

- `source`—the window from which the message was sent.

A Service Discovery Example: Message Boxes

This section contains two code listings: `index.html` and `msg_contents.html` that, when combined, implement basic service discovery and data relaying on top of the cross-document messaging architecture.

To use this code, you must first do the following things:

- Install Safari 4.0 or later, or install a recent WebKit nightly.

- Save the contents of the two code listings in separate files.

- In the file `msg_contents.html`, modify the variable `allowed_origins` to contain a list of origins from which you plan to serve the `index.html` or `msg_contents.html` file.

  For example, if you intend to place this file at `http://www.example.org/message_test/msg_contents.html`, you should make sure that `'http://www.example.org'` (enclosed in quotes) is a key in the `allowed_origins` object.

  If you only have one machine, turn on web sharing, then use `http://localhost` as one origin and `file://` as the other.

- In the file `msg_contents.html`, optionally change the variable `root_origin` to the actual expected origin of the top level HTML page.

- Place a copy of the modified `msg_contents.html` file on the desired server or servers.

- In the file `index.html`, replace the `allowed_origins` declaration with the one from your `msg_contents.html` file. Then, update the `boxes` object to provide the URLs for the `msg_contents.html` files you just put on your servers.

  If you only have one machine, use a `file://` URL pointing to the path of the `msg_contents.html` file. For example, if you placed the file in `/Library/WebServer/Documents/message_test/msg_contents.html`, the local URL would be `file:///Library/WebServer/Documents/message_test/msg_contents.html`.

- Place this modified `index.html` file on one of the servers and navigate to the URL, or open it as a local file on disk.

Once you have completed these steps, you should see several boxes, one per entry in the `boxes` object. Each of these boxes should contain a small form with a text input box, a series of checkboxes (one for each of the outer boxes), and a submit button.

If you type something into the text box, check one of the checkboxes, and click submit, the text should appear at the bottom of the window whose name corresponds with the checkbox.

Listing 1   Cross-document messaging example: index.html

```
<html><head>
<script language='javascript' type='text/javascript'><!--
/*global alert, navigator, document, window */


var window_list = [];
var origin_list = [];

var boxes = {
    local: "http://host1.domain1.top/messages2/msg_contents.html",
    remote: "http://host2.domain2.top/messages2/msg_contents.html",
    third: "http://host3.domain3.top/messages2/msg_contents.html"
};

var allowed_origins = {
    'http://host1.domain1.top': 1,
    'http://host2.domain2.top': 1,
    'http://host3.domain3.top': 1
};

function smartsplit(string, pattern, count)
{
    // alert('string '+string);
    // alert('pattern "'+pattern+'"');
    // alert('count '+count);

    var lastpos = count - 1;

    var arr = string.split(/ /);
    // alert('AC: '+arr.length+" "+string);
    if (arr.length > lastpos) {
        var temparr = [];
```

```
        for (var i=lastpos; i<arr.length; i++) {
            temparr[i-lastpos] = arr[i];
            arr[i] = undefined;
        }
        arr[lastpos] = temparr.join(pattern);
    }
    return arr;
}

function listWindows()
{
    var retstring = "";
    for (var i in origin_list) {
        if (origin_list.hasOwnProperty(i)) {
        // alert('UUID: '+i+' Origin: '+origin_list[i]);
        retstring += i+' '+origin_list[i]+'\n';
        }
    }
    return retstring;
}


function messageReceive(evt) {
    var windowlist;

    if (evt.origin === null || evt.origin in allowed_origins) {
    var arr = smartsplit(evt.data, " ", 2);

    if (arr[0] == 'sendto') {
        // usage: sendto UUID remote_origin message
        arr = smartsplit(evt.data, " ", 4);
        var remote_window = window_list[arr[1]];
        remote_window.postMessage('sendto_output '+arr[3], arr[2]);
    } else if (arr[0] == 'register') {
        // usage register UUID
        var name = arr[1];

        if (window_list[name]) {
            // name conflict.
            var add=1;
            while (window_list[name+'_'+add]) {
                add++;
            }
            name = name+'_'+add;
            evt.source.postMessage("register_newid "+name, evt.origin);
        }

        window_list[name] = evt.source;
        origin_list[name] = evt.origin;
        windowlist = listWindows();
        for (var windowid in window_list) {
            if (window_list.hasOwnProperty(windowid)) {
            // alert('windowid: '+windowid);
            window_list[windowid].postMessage("list_output "+windowlist,
origin_list[windowid]);
            }
        }
    } else if (arr[0] == 'list') {
```

```
        // usage list
        windowlist = listWindows();
        evt.source.postMessage("list_output "+windowlist, evt.origin);
    } else {
        alert('unknown command '+arr[0]);
    }
    } else {
        alert('unexpected message from origin '+evt.origin);
    }
}

function dosetup()
{
    if (navigator.userAgent.match(/Safari/)) {
        var version = parseFloat(navigator.userAgent.replace(/.*AppleWebKit\//,
 "").replace(/[^0-9.].*$/, ""));

        if (version < 528) {
            alert('WebKit version '+version+' does not support this
application.');
        }

    }
    // for (var i=0; i<nchannels; i++) {
        // alert('setup channel '+i);
        // channel[i] = new MessageChannel();
    // }
    window.addEventListener('message', messageReceive, false);

    var boxstr = "";
    for (var i in boxes) {
        if (boxes.hasOwnProperty(i)) {
        boxstr += "<iframe height='600' id='"+i+"'
src='"+boxes[i]+"'></iframe>\n";
        }
    }
    var boxlistdiv = document.getElementById('boxlist');
    boxlistdiv.innerHTML = boxstr;
}

dosetup();

--></script>
</head>
<body onload='dosetup();'>
<div id='boxlist'>
</div>
</body>
</html>
```

Listing 2   Cross-document messaging example: msg_contents.html

```
<html><head>
<script language='javascript' type='text/javascript'><!--
/*global alert, document, window, navigator */

var allowed_origins = {
    'http://stavromula-beta.apple.com': 1,
```

```
        'http://holst.apple.com': 1
};

var received_root_origin = '';

var root_origin = '*';

function smartsplit(string, pattern, count)
{
        // alert('string '+string);
        // alert('pattern "'+pattern+'"');
        // alert('count '+count);

        var lastpos = count - 1;

        var arr = string.split(/ /);
        // alert('AC: '+arr.length+" "+string);
        if (arr.length > lastpos) {
                var temparr = [];
                for (var i=lastpos; i<arr.length; i++) {
                        temparr[i-lastpos] = arr[i];
            arr[i] = undefined;
                }
                arr[lastpos] = temparr.join(pattern);          }
        return arr;
}

function mkcheckbox(inpstr)
{
    var str = "";

    var arr = inpstr.split("\n");
    for (var entid in arr) {
        if (arr.hasOwnProperty(entid)) {
        var ent = arr[entid];
        if (ent !== "") {
            // alert('ent: '+ent);
            var bits = smartsplit(ent, " ", 2);
            str += "<input type=checkbox name='"+ent+"'>"+bits[0]+"</input>\n";
        }
        }
    }
    return str;
}

function messageReceive(evt) {
    if (evt.origin === null || evt.origin in allowed_origins) {

        // The message came from an origin that
        // your code trusts.  Work with the message.

        // alert('Received data: '+evt.data);
    // var tmp = 'Test this, please';
    // var x = smartsplit(tmp, " ", 2);
    // alert('x[0] = '+x[0]);
    // alert('x[1] = '+x[1]);
    // alert('x[2] = '+x[2]);
```

```
    var arr = smartsplit(evt.data, " ", 2);
    if (arr[0] == 'list_output') {
        if (evt.origin == root_origin || root_origin == '*') {
        var div2 = document.getElementById('temp2');
        div2.innerHTML = mkcheckbox(arr[1]);
        received_root_origin = evt.origin;
        // alert('arr[1] = '+arr[1]);
        } else {
      alert('received list_output message from unexpected origin: '+evt.origin);
        }
    } else if (arr[0] == 'sendto_output') {
            // alert('Received data: '+evt.data);
        var div3 = document.getElementById('temp3');
        div3.innerHTML += arr[1]+'<br />\n';
    } else if (arr[0] == 'register_newid') {
        var mydiv = document.getElementById('temp');
        mydiv.innerHTML = arr[1]+" box";
    }

        // Send a message back to the source:
        // evt.source.postMessage('response', evt.origin);
    } else {
        alert('unexpected message from origin '+evt.origin);
    }
}

function setup_listener()
{
    window.addEventListener('message', messageReceive, false);
}

function setup()
{
    var mydiv = document.getElementById('temp');

    // mydiv.innerHTML = 'Test';
    // mydiv.innerHTML = ' '+bigdoc;

    // If we can access the document object, we are locally loaded and should
    // talk to the remotely-loaded window.  Otherwise, the reverse
    // is true.

    var myid = '';
    if (window.parent.document) {
    myid = 'local';
    } else {
    myid = 'remote';
    }

    mydiv.innerHTML = myid+" box";
    // window.addEventListener('message', messageReceive, false);
    setup_listener();

    var topwindow = window;
    while (topwindow.parent && (topwindow.parent != topwindow)) { topwindow =
topwindow.parent; }

    topwindow.postMessage('register '+myid, root_origin);
```

```
    // window.parent.postMessage('list', '*');

}

function sendmsg()
{
    var message = document.getElementById('sendtext').value;
    var mydiv2 = document.getElementById('temp2');
    var checkboxes = mydiv2.children;

    // alert('checkboxes: '+checkboxes);
    var topwindow = window;
    while (topwindow.parent && (topwindow.parent != topwindow)) { topwindow =
topwindow.parent; }

    for (var boxid in checkboxes) {
        if (checkboxes.hasOwnProperty(boxid)) {
        var checkbox = checkboxes[boxid];

        // alert('checkbox: '+checkbox);
        if (checkbox.tagName == "INPUT") {
            // alert('input');
            if (checkbox.checked) {
                // alert('checked');
                var arr = smartsplit(checkbox.name, " ", 2);
                var uuid = arr[0];
                var origin = arr[1];
                // alert('send to: '+uuid+' origin '+origin);

                // alert('topwindow is '+topwindow);
                topwindow.postMessage('sendto '+uuid+' '+origin+' '+message,
received_root_origin);
            }
        }
        }
    }


    return false;
}


--></script>
</head><body onload='setup();'>
<div id='temp'></div>
<form onsubmit='return false;'>
<input type='text' id='sendtext'></input>
<input type='submit' value='submit' onclick='sendmsg();'></input>
<div id='temp2'></div>
</form>
<div id='temp3'></div>

</body></html>
```

You can create numerous interesting extensions on top of this sort of design. For example, you might add a command message that asks the window at the other end what commands it supports, then communicate with it if you share a common set of commands. The possibilities are limitless.

Security Considerations

There are several key things you should be aware of when using cross-document messaging:

- Obtaining `Window` objects for other windows is not always easy. Scripts running in a window, frame, or `iframe` element served from one origin cannot access the DOM tree of documents served from a different origin, and thus cannot get access to the `Window` objects of other `iframe` elements within such a window.

  Among other things, this means that two `iframe` elements from different domains cannot directly obtain each other's `Window` objects because one or the other is (by definition) from a different origin than the document containing the `iframe` elements. In this situation, there are two possible solutions.

  The easiest solution is to have both windows discover each other using the parent window as a communications hub. This solution is also the most general solution because it works even if neither window can see the element containing the other window.

  Alternatively, the window with access to the parent window's DOM tree could discover the other one and initiate communication. By doing so, the second window receives the first window's `Window` object by way of the `source` field in the event object.

  > **Note:** The `postMessage` function does not allow you to pass an object to other windows. This means that if you need to communicate between windows that cannot obtain one another's `Window` objects, you must handle all communication using relay code at the top level of the window hierarchy.

- Sending data to other windows can be dangerous, particularly if that information contains login information or other sensitive data. You should almost always take advantage of the target origin field when sending messages to avoid interception by content served by other sites. You should only use the wildcard asterisk (*) target origin value if you are absolutely sure that the data is harmless.

- Receiving data from other windows can also be dangerous. You should generally check the origin field when receiving messages to make sure that the data was sent from a website that you trust to some degree.

  Where possible, you should also check any received data for validity before using it. In particular, you should generally avoid executing JavaScript code received from another window (with the possible exception of JSON objects after careful validity checking).

As with any software, for maximum reliability and security, you should write your output code carefully to minimize the risk of causing problems for other code, and you should write your code under the assumption that other code is maliciously trying to attack your code, and thus you should perform type, bounds, and other sanity checks accordingly.

Calling Objective-C Methods From JavaScript

The web scripting capabilities of WebKit permit you to access Objective-C properties and call Objective-C methods from the JavaScript scripting environment.

An important but not necessarily obvious fact about this bridge is that it does *not* allow *any* JavaScript script to access Objective-C. You cannot access Objective-C properties and methods from a web browser unless a custom plug-in has been installed. The bridge is intended for people using custom plug-ins and JavaScript environments enclosed within WebKit objects (for example, a WebView).

How to Use Objective-C in JavaScript

The WebScripting informal protocol, defined in `WebScriptObject.h`, defines methods that you can implement in your Objective-C classes to expose their interfaces to a scripting environment such as JavaScript. Methods and properties can both be exposed. To make a method valid for export, you must assure that its return type and all its arguments are Objective-C objects or basic data types like `int` and `float`. Structures and non object pointers will not be passed to JavaScript.

Method argument and return types are converted to appropriate types for the scripting environment. For example:

- JavaScript numbers are converted to NSNumber objects or basic data types like `int` and `float`.
- JavaScript strings are converted to NSString objects.
- Other JavaScript objects are wrapped as WebScriptObject instances.

Instances of all other classes are wrapped before being passed to the script, and unwrapped as they return to Objective-C.

As an exception, JavaScript arrays cannot be cleanly mapped to `NSArray` objects because they are a hybrid between a numerically-indexed array and an associative array. To avoid loss of data during the mapping, you must instead use the `webScriptValueAtIndex:` and `setWebScriptValueAtIndex:value:` methods.

A Sample Objective-C Class

Let's look at a sample class. In this case, we will create an Objective-C address book class and expose it to JavaScript. Let's start with the class definition:

```
@interface BasicAddressBook: NSObject {
}
+ (BasicAddressBook *)addressBook;
- (NSString *)nameAtIndex:(int)index;
@end
```

Now we'll write the code to publish a `BasicAddressBook` instance to JavaScript:

```
BasicAddressBook *littleBlackBook = [BasicAddressBook addressBook];

id win = [webView windowScriptObject];
[win setValue:littleBlackBook forKey:@"AddressBook"];
```

Once you expose these methods to JavaScript (described at the end of this section), you should be able to access your basic address book from the JavaScript environment and perform actions on it using standard JavaScript functions.

Now, let's make an example showing how you can use the `BasicAddressBook` class instance in JavaScript. In this case, we'll print the name of a person at a certain index in our address book:

```
function printNameAtIndex(index) {
    var myaddressbook = window.AddressBook;
    var name = myaddressbook.nameAtIndex_(index);
    document.write(name);
}
```

You may have noticed one oddity in the previous code example. There is an underscore after the JavaScript call to the Objective-C `nameAtIndex` method. In JavaScript, it is called `nameAtIndex_`. This is an example of the default method renaming scheme in action.

Unless you implement `webScriptNameForSelector` to return a custom name, the default construction scheme is used. It is your responsibility to ensure that the returned name is unique to the script invoking this method. If your implementation of `webScriptNameForSelector` returns `nil` or you do not implement it, the default name for the selector will be constructed as follows:

■   Any colon (":") in the Objective-C selector is replaced by an underscore ("_").

■   Any underscore in the Objective-C selector is prefixed with a dollar sign ("$").

■   Any dollar sign in the Objective-C selector is prefixed with another dollar sign.

The following table shows example results of the default method name constructor:

Objective-C selector	Default script name for selector
setFlag:	setFlag_
setFlag:forKey:withAttributes:	setFlag_forKey_withAttributes_
propertiesForExample_Object:	propertiesForExample$_Object_
set_$:forKey:withDictionary:	set$_$$_forKey_withDictionary_

Since the default construction for a method name can be confusing depending on its Objective-C name, you would benefit yourself and the users of your class if you implement `webScriptNameForSelector` and return more human-readable names for your methods.

Getting back to the BasicAddressBook, now we'll implement `webScriptNameForSelector` and `isSelectorExcludedFromWebScript` for our `nameAtIndex` method. In our BasicAddressBook class implementation, we'll add this:

```
+ (NSString *) webScriptNameForSelector:(SEL)sel
```

```
{
    ...

    if (sel == @selector(nameAtIndex:))
            name = @"nameAtIndex";

    return name;
}

+ (BOOL)isSelectorExcludedFromWebScript:(SEL)aSelector
{
    if (sel == @selector(nameAtIndex:)) return NO;
    return YES;
}
```

Now we can change our JavaScript code to reflect our more logical method name:

```
function printNameAtIndex(index) {
    var myaddressbook = window.AddressBook;
    var name = myaddressbook.nameAtIndex(index);
    document.write(name);
}
```

> **Important:** For security reasons, no methods or KVC keys are exposed to the JavaScript environment by default. Instead a class must implement these methods:
>
> ```
> + (BOOL)isSelectorExcludedFromWebScript:(SEL)aSelector;
> + (BOOL)isKeyExcludedFromWebScript:(const char *)name;
> ```
>
> The default is to exclude all selectors and keys. Returning NO for some selectors and key names will expose those selectors or keys to JavaScript. This is described further in *WebKit Plug-In Programming Topics*.

For More Information

For more information about using Objective-C from JavaScript and vice versa, see the following documents:

- *CallJS* sample code—shows how to call JavaScript from Objective-C and vice versa.

- *Birthdays* sample code—shows how to use a WebKit plug-in from JavaScript.

- *WebKit Objective-C Framework Reference*—provides more information on excluding methods and properties from the JavaScript environment.

- *WebKit Plug-In Programming Topics*—tells more about writing WebKit plug-ins in general.

Document Revision History

This table describes the changes to *WebKit DOM Programming Topics*.

Date	Notes
2010-01-20	Minor edits.
2009-06-08	Added coverage for credentials, drag-and-drop example.
2009-04-29	Added additional cross references and clarified a few things.
2009-02-05	Updated for Safari 4.0.
2008-10-15	Minor edits throughout.
2007-09-04	Added additional tips for cross-browser development.
2007-07-10	Refreshed references to other documents.
2007-06-11	Retitled document from Safari JavaScript Programming Topics. Includes new article on the XMLHttpRequest object.
2006-05-23	Corrected typos.
2006-02-07	Corrected typos.
2006-01-10	Corrected typos.
2005-11-09	Corrected typos.
2005-08-11	Corrected typos.
	Corrected typos.
	Added link to the canvas API.
2005-06-04	Corrected typos and clarified the argument types for some methods.
2005-04-29	Added article on using Objective-C from JavaScript.
2004-11-02	First version of *Safari JavaScript Programming Topics*.