WebKit DOM Programming Topics

Apple Applications > Safari

2008-10-15

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures and Tables

Introduction to WebKit DOM Programming Topics

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

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" (page 27) describes how to request data from network resources using the `XMLHttpRequest` object.

- "Using Objective-C From JavaScript" (page 31) 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

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. Read JavaScript Coding Guidelines for Mac OS X to learn more about the JavaScript language.

Read *Dashboard Programming Topics* for 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.

About JavaScript and the DOM

JavaScript is a platform-independent, object-oriented scripting language designed for the web, originally created by Netscape Communications and 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>
    <SCRIPT LANGUAGE="JavaScript"><!--

        function myFunction() {
            ...
        }
    -->
    </SCRIPT>
</HEAD>
```

The script's content is enclosed in an HTML comment by convention—it helps shield the client-parsed code from browsers that cannot interpret JavaScript, and though optional, should be used if you plan to re-use your code for other browsers.

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 13).

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>
</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 do some DOM transformations. This first example appends a paragraph to the DOM tree, following the `firstParagraph` and `secondParagraph` elements:

```
parasDiv = document.getElementById("allMyParas");
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:

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

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'/>
```

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`.

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.

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 `cancelDefault` 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 and reports whether the drag was successful.

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.

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 `-khtml-user-drag` attribute 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 -khtml-user-drag attribute

Value	Description
`none`	Do not allow this element to be dragged.
`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); -khtml-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.

Value	Description
`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 `-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 `-khtml-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:-khtml-drag` pseudoclass.

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

#divSrc4:-khtml-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 Tookit, DOM-Drag, ToolMan, Rico, and others.

Using XMLHttpRequest

Safari, Dashboard, and WebKit-based applications support the JavaScript `XMLHttpRequest` object. `XMLHttpRequest` allows you to easily fetch content from another source and use it within your webpage or widget.

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 working with either the request's response text or its response XML, as discussed in "XMLHttpRequest Responses" (page 28).

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;
```

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 URIs in the same domain as the webpage. Also, only URIs with HTTP handlers are allowed.

Using Objective-C 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.
- JavaScript arrays are mapped to NSArray 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.

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"];
```

That's all it takes. You can now access your basic address book from the JavaScript environment and perform actions on it using standard JavaScript functions. 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 will be 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` 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;
}
```

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);
}
```

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.

See *WebKit Objective-C Framework Reference* for all the information on excluding methods and properties from the JavaScript environment.

Document Revision History

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

Date	Notes
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.*