WebKit Objective-C Programming Guide

Networking, Internet, & Web: Web Client

2009-07-28

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

Simultaneously published in the United States and Canada.

Contents

Figures and Listings

Introduction to WebKit Objective-C Programming Guide

Concurrency Note: The WebKit framework is not thread safe. If you call functions or methods in this framework, you must do so exclusively on the main program thread.

What Is the WebKit?

The WebKit provides a set of core classes to display web content in windows, and by default, implements features such as following links clicked by the user. The WebKit greatly simplifies the complicated process of loading webpages—that is, asynchronously requesting web content from an HTTP server over the network where the response may arrive incrementally, in random order, or partially due to network errors. The WebKit also simplifies the process of displaying content that can contain various MIME types, and multiple frames each with their own set of scrollbars.

You use the WebKit to display web content in a window of your application. It's as simple as creating a view, placing it in a window, and sending a URL load request message. By default, your WebKit application behaves as you would expect without error. The WebKit conveniently creates and manages all the views needed to handle different MIME types. When the user clicks on a link in a page, the WebKit automatically creates the views needed to display the next page.

However, the WebKit doesn't implement a complete set of web browser features. You can, however, extend the WebKit by implementing custom delegate, view, and model objects. For example, you can implement a delegate to display load status, and the current URL.

The WebKit also offers web content editing. If you enable editing in your WebView, users can edit the web content it displays. You can programmatically control the current selection and control editing behavior using a WebView delegate. You can also modify the Document Object Model directly using an Objective-C API.

You can also access JavaScript from Objective-C and vice versa.

Who Should Read This Document?

The WebKit Objective-C API is specifically designed for embedding web content in your Cocoa or Carbon applications—developing web client applications *not* web server applications or web content. It is also not suitable for implementing non-GUI applications such as web crawlers. If you are a web content creator or JavaScript programmer, refer to *WebKit DOM Programming Topics*.

Important: Currently, this API is available in Objective-C only. A minimal C API is provided for embedding web browser views in Carbon applications. You can use Objective-C in combination with C. The WebKit works with all versions of Mac OS X 10.2 that have Safari 1.0 installed.

Organization of This Document

The following articles cover key concepts in understanding how the WebKit works:

- "Why Use the WebKit?" (page 13) describes the purpose of the WebKit and why you might want to use it in your applications.
- "Core WebKit Classes" (page 15) describes the core WebKit classes and the object-oriented design that is fundamental to understanding how the WebKit works.

The following articles explain how to display web content in views:

- "Simple Browsing" (page 21) shows how to embed web content in your application by following a few simple steps.
- "Multiple Windows" (page 23) shows how to add support for multiple windows, and open windows automatically.
- "Loading Pages" (page 27) shows how to track the progress of loading frame content.
- "Loading Resources" (page 31) shows how to track the progress of loading individual resources on a page.
- "Paging Back and Forward" (page 35) shows how to implement a back-forward list and add Back and Forward buttons to your application.
- "Managing History" (page 37) shows how to maintain a history of all the visited pages, and allow the user to go to a previously visited page.
- "Spoofing" (page 59) shows how to use user-agent strings.
- "Accessing the WebKit From Carbon Applications" (page 61) explains how to embed web content in Carbon applications.
- "Determining WebKit Availability" (page 63) explains how to determine if the WebKit is available on your system.

The following articles explain how to implement web content editing:

- "Enabling Editing" (page 41) shows how to enable user editing in a WebView.
- "Saving and Loading Web Content" (page 43) shows how to save and load web content edited by the user.
- "Modifying the Current Selection" (page 45) shows how to programmatically modify the current selection.
- "Changing Editing Behavior" (page 47) explains how to use the WebView editing delegate to customize editing behavior.
- "Using Undo When Editing" (page 49) shows how to implement undo when editing web content.

The following articles explain how to use the Document Object Model Objective-C API:

- "Using the Document Object Model from Objective-C" (page 51) describes the DOM Objective-C API in terms of the specification.
- "Using the Document Object Model Extensions" (page 55) describes the WebKit extensions to the DOM API.

Read this article if you want to access JavaScript from your application:

- "Using JavaScript From Objective-C" (page 57) shows how to access the scripting environment from an Objective-C application.

You begin using the WebKit by first embedding web content in your application. Read "Simple Browsing" (page 21), and, optionally, "Loading Pages" (page 27) and "Loading Resources" (page 31) to embed web content. If you want to add more browser-like features or implement a custom user interface, read "Core WebKit Classes" (page 15) first and any other articles based on your application needs. If you want to edit web content, read "Enabling Editing" (page 41).

See Also

For more details on the Objective-C WebKit API, read:

- *WebKit Objective-C Framework Reference*
- *WebKit Plug-In Programming Topics*
- *WebKit DOM Programming Topics*

There are other technologies, not covered in this topic, that can be used in conjunction with the WebKit or separately to solve related problems.

Refer to this document for more details on the URL loading system:

- *URL Loading System Programming Guide*

If you are accessing the WebKit from a Carbon application, refer to these documents:

- *WebKit C Reference*
- *Carbon-Cocoa Integration Guide*

If you are creating web content for Safari or Dashboard, refer to these documents:

- *WebKit DOM Programming Topics*
- *WebKit DOM Reference*
- *Dashboard Tutorial*
- *Dashboard Reference*

The `/Developer/Examples/WebKit` folder also contains more in-depth code examples.

Other related text book resources are:

- *HTML and XHTML: The Definitive Guide* (O'Reilly)
- *Cascading Style Sheets: The Definitive Guide* (O'Reilly)
- *JavaScript: The Definitive Guide* (O'Reilly)

Also refer to the World Wide Web Consortium at www.w3.org for the latest information on web standards.

Why Use the WebKit?

Many applications need to display web content in windows, whether it's live content on the web, or static files on disk. Some applications are full-featured browsers, but more often applications embed web content as a convenience, as in a custom document system. HTML is the de facto standard representation of documents on the internet, so it's only natural that you will want to display that content without having to launch a web browser for each file or link clicked by the user.

Some applications might want to display web content on demand but don't necessarily want to parse it or understand its structure to display it. Applications like this may not want to open multiple windows or provide back and forward buttons. The WebKit provides a set of classes to support a variety of web content—from the most trivial embedded web content application (with web content displayed in a single window) to a full-featured web browser such as Safari.

The WebKit does this by hiding the details of the complex task of loading and displaying web content. The web is based on a client-server architecture, in which clients make asynchronous requests to web servers for page content. During this process any number of problems can occur not only when transmitting these requests and responses over the network but also when displaying the content once it's received by your application. Content may be complex. It can contain multiple frame elements, multiple MIME types, such as images and movies. Some MIME types require browser plug-ins to display them.

By default, the WebKit core classes transparently handle programmatic and client requests. The WebKit creates all the necessary model and view classes used to represent and display the incoming content. When a user clicks a link, the WebKit automatically releases the old objects and creates new ones to handle the new page. The WebKit views are designed to handle multiple frames, each with their own scroll bar, and many MIME types. You do not need to implement custom views for your application to display web content in your application.

On the other hand, if you are implementing a custom, full-featured browser or other internet application, you can extend the WebKit to handle the details of client requests, frame and resource loading, window operations, and downloading. You do this by implementing delegate objects. The WebKit provides a number of "hooks" allowing applications to customize the user interface. For example, you can specify the menu items that are displayed when the user clicks a particular type of resource. You can also implement your own document models and views to handle specific MIME types. Because of this extensibility, the WebKit can be used to develop some innovative internet applications.

Core WebKit Classes

Understanding the object-oriented design of the core WebKit classes is fundamental to understanding how the WebKit works. You can display web content in a single window by following a few simple steps. Normally, to embed web content in your application you simply create a WebView object, place it in a window, and send a load request message. However, if you want to do something more complex—for example, customize the user interface, use multiple windows, or implement any other browser-like features, such as back and forward buttons—you will want to understand better how the WebKit classes work together to load and display web content.

Important: The object-oriented design diagrams in this article are meant to show at a high level how the classes relate to each other. These diagrams are not intended to show instance variables.

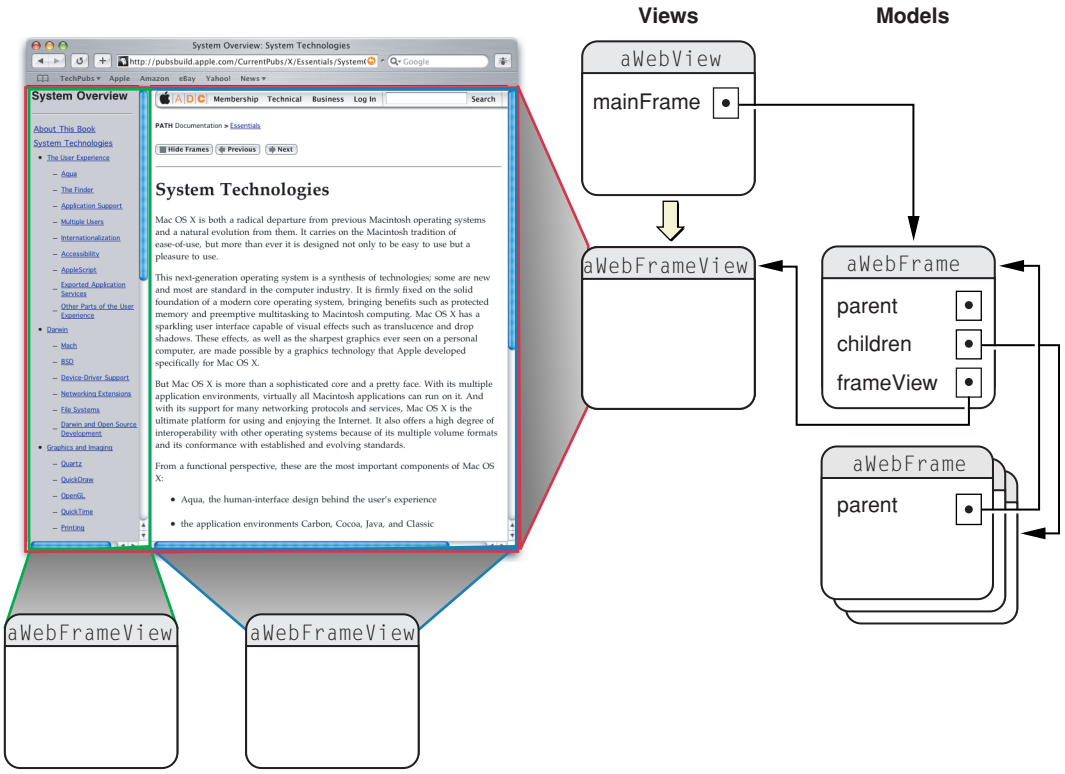
Frame Model and View Classes

The WebKit loosely follows a model-view-controller paradigm—some objects represent view-controllers that display web content, and other objects represent models that encapsulate web content.

WebView is the core view class in the WebKit. WebView objects manage interactions between WebFrameView objects and WebFrame objects. To embed web content in your application, you create a WebView object, attach it to a window, and ask its main frame to load a URL. The most common example of web content is a single frame containing multiple MIME types. The WebKit also fully supports HTML files containing compound frames.

For example, suppose a webpage contains a frame with two children frames, as illustrated in Figure 1 (page 16). To load this page, you send a request to the main frame of `aWebView`, an instance of WebView. The main frame initiates a client request. While it receives the server response (that is, loads the page content), the main frame creates WebFrame objects to encapsulate the content contained in each frame element. A hierarchy of WebFrame objects is used to model an entire webpage, where the root is called the **main frame**.

Figure 1 WebView and WebFrameView objects



As the content for each WebFrame object is loaded, a corresponding WebFrameView object is created to display that content. These WebFrameView objects are attached to the WebView's view hierarchy. Therefore, there is a parallel hierarchy of WebFrameView objects used to render an entire page. In this hierarchy, the WebView object is not only a controller object but also the root view. The details of the view hierarchy are not shown because they are private to the implementation of the WebKit and may change in the future.

Fortunately, you do not need to create these model and view objects directly. The WebKit creates these objects automatically whenever pages are loaded, either programmatically or when the user clicks a link.

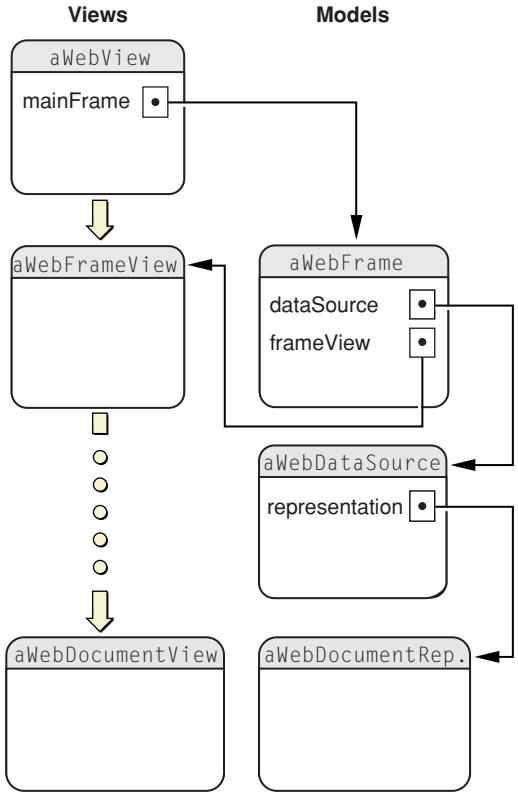
Data Model and View Classes

Once the frame hierarchies are created, the actual content for each frame needs to be loaded and displayed. Since webpages can contain different MIME types, the WebKit implements different models and views to display them. The WebKit automatically loads and displays most of the common document types (for example, HTML, XML, plain text, images, and QuickTime movies). The WebKit selects the appropriate data model and view object based on the document's MIME type. In fact, the WebKit design is extensible, allowing you to create your own data models and views for specific MIME types.

Figure 2 (page 17) illustrates the relationship between WebFrame, WebDataSource, document representation, and document view objects. For each WebFrame object, there's one WebDataSource object that loads the content for that frame. For each WebDataSource object, there's one document representation object, conforming to the WebDocumentRepresentation protocol, that encapsulates the data for a specific MIME type. For each documentation representation object, there's a document view object, conforming to the

WebDocumentView protocol, that handles the display of that data. The document view object is contained in the corresponding WebFrameView object (for example, the document view of a scroll view contained in a WebFrameView object). Again, the details of the view hierarchy are not shown because they are private to the implementation of the WebKit.

Figure 2 WebFrame and WebDataSource objects



Because document representation and view objects are separate, you can have multiple models and views of a MIME type, and extend the WebKit by defining your own. Once a data source is committed (the first byte of data has arrived), the WebKit selects an appropriate document representation and document view object based on the MIME type of the data source. The WebKit already supplies the model and view objects for most of the common MIME types. If a MIME type is not supported, you can supply your own model and view objects to handle that type, and register them with the WebView class. You can even replace the default model and view objects, although that's not recommended.

Again, you do not have to create any of these objects directly—they are automatically created when a page is loaded.

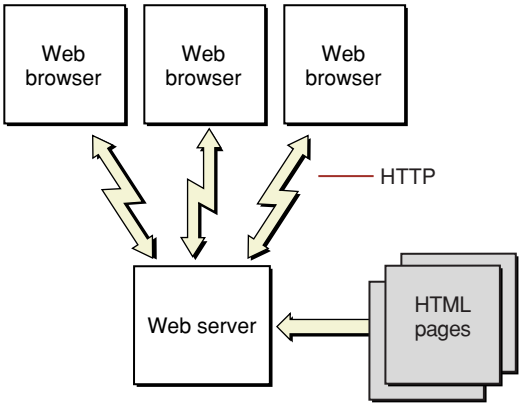
Provisional vs. Committed Data Sources

When you send a request to load a webpage, you receive an asynchronous response because the request is being sent to another process on another machine over the network. Because of this, the WebKit needs to handle the state of its objects between the time a request is initiated and the first byte of data arrives. When using the WebKit you should be aware of the transitional state of the WebKit data source objects.

In addition to requests being asynchronous, many errors can result from requesting web content over the network. For example, there can be network failures, bad URL strings, corrupted content, and no available plug-ins. Or, you may initiate a load request but find the response slow, or delayed (the content trickles in).

For example, a typical static website looks something like the one in Figure 3 (page 18). Client requests, conforming to the HTTP protocol, originate at the user's web browser. These requests are sent over the network to the web server, which analyzes the request and selects the appropriate webpage to return to the client browser. This webpage is simply a text file that contains HTML markup. Using the HTML commands embedded within the file received from the web server, the browser renders the page.

Figure 3 Typical website



In the WebKit, client load requests are asynchronous. To handle the state of objects during the time a request is initiated and content arrives, the WebKit creates what is called a **provisional** data source. The data source is provisional because it isn't known yet whether the page will load successfully. Any existing data source for a page remains valid until the provisional data source is validated. The first time a WebFrame object is loaded there's no existing data source and a blank page is displayed.

A data source becomes **committed** as soon as the first byte of data arrives. If the provisional data source becomes invalid due to some error, it never transitions to a committed data source. When a data source is committed, an appropriated document representation and document view is created for the WebFrame object.

Note that the default WebKit behavior does nothing if a load request fails. Therefore, you need to implement WebView delegates to handle load errors (for example, to display or log a message).

WebView Delegates

You customize the behavior of the WebKit by implementing WebView delegates that intercept request and response messages, and make policy and user interface decisions. WebView uses a multiple delegate model because there are so many aspects of the WebKit behavior that you can customize. And for large applications, it makes sense for different objects to handle the different sets of delegate messages. Of course, you can always implement just one delegate to handle all these areas. WebView objects have four delegates:

- Frame load delegate—intercepts frame-level request and response messages to track the progress and errors that might result in loading a webpage (see the WebFrameLoadDelegate informal protocol).

- Resource load delegate—intercepts resource-level request and response messages to track the progress and errors that might result in loading a resource (see the WebResourceLoadDelegate informal protocol).
- User interface delegate—controls the opening of new windows, augments the default menu items displayed when the user clicks on elements, and makes other window and control user interface decisions (see the WebUIDelegate informal protocol).
- Policy delegate—modifies the policy decisions that are made when handling URLs or the data they represent (see the WebPolicyDelegate informal protocol).

Because all the delegates use informal protocols, you can set the delegates and implement the delegate methods you want. If you don't implement a delegate method, the WebKit uses a default implementation. For example, by default, error messages are not reported, and new windows are not opened when a link is clicked that results in a new window request. If the WebKit cannot reach a URL, your application window displays the old content, which may be a blank page. You typically implement a frame load delegate to handle these types of errors.

Simple Browsing

By writing just a few lines of code using the WebKit, you can embed web content in your application and enable your users to navigate the web.

The code example below assumes you already have a WebView object inside a window that represents the webpage. You can create a WebView object and attach it to a window either programmatically or by using Interface Builder. Using Interface Builder, drag a WebView from the Library into a window. You should also connect the WebView object to an outlet so that you can send messages to it programmatically.

Next, you load a webpage by sending a `loadRequest:` message to the main frame of your WebView object, as in this example:

```
[[webView mainFrame] loadRequest:[NSURLRequest requestWithURL:[NSURL
URLWithString:urlText]]];
```

Here, `webView` is an instance of WebView and `urlText` is a valid URL address such as `http://www.apple.com`. You can do this after the nib file is loaded, so that your WebView object displays a default page.

When you run your application you'll notice that the URL (if it is valid) is successfully displayed in the window and most links are followed automatically when clicked. Content that contains multiple frames is automatically handled. The WebKit does this by creating a hierarchy of WebFrameView and WebFrame objects to handle frame content—even content that contains QuickTime movies, JavaScript, or Flash movies works.

Even though navigating works, this simple example doesn't contain many of the features you would expect in a web browser—namely, a text field for typing in URLs, back and forward buttons, a history menu, multiple windows, and status or error messages. You can add some of these features by assigning delegates to your WebView object and implementing delegate methods. Because the delegate methods are informal protocols, you can choose which methods you want to implement.

Multiple Windows

If you are building a simple browser application, you'll want to allow the user to open multiple windows and type in URLs. You also need to make some policy decisions about how to handle new window requests.

Opening Windows

You can implement multiple windows in a WebKit application easily by beginning with a Cocoa document-based architecture as follows:

1. Using Xcode, create a document-based Cocoa application. Your new project file will already contain the needed classes and interface files to support multiple windows (namely `MyDocument.h`, `MyDocument.m`, and `MyDocument.nib`).
2. Add the WebKit frameworks to your project.
3. Open `MyDocument.nib` using Interface Builder and drag a WebView from the Cocoa—GraphicsViews palette to your document window.
4. Create a `webView` outlet in `MyDocument.h` and read the file into Interface Builder. Connect the `webView` outlet of the File's Owner to the WebView object you created in the previous step.
5. Add code to `MyDocument.m` to load a default page. You can add this code fragment to the `windowControllerDidLoadNib:` method:

```
NSString *urlText = [NSString stringWithString:@"http://www.apple.com"];
[[webView mainFrame] loadRequest:[NSURLRequest requestWithURL:[NSURL
URLWithString:urlText]]];
```

6. Build and run your application.

When you run your application you should see a window open displaying web content. You can also open multiple windows by selecting New from the File menu. This example demonstrates multiple WebView objects independently displaying web content.

Entering URLs

If you want a user to type in her own URL, add a text field to the window and make a new load request every time the user enters a new URL. Here are the steps you follow:

1. Add a text field to the window in `MyDocument.nib`.

2. Add a `textField` outlet and a `connectURL:` action to `MyDocument.h`. Read in the changes to `MyDocument.h` from Interface Builder.

3. In Interface Builder, make a connection from the text field you created to the `textField` outlet of the File's Owner and set the action to `connectURL:`. This method is invoked when the user enters in a new URL.

4. Implement the `connectURL:` method to load the URL typed in by the user:

```
- (IBAction)connectURL:(id)sender{
    [[webView mainFrame] loadRequest:[NSURLRequest requestWithURL:[NSURL
URLWithString:[sender stringValue]]]];
}
```

5. Build and run your application.

Now when you enter a new URL in the text field, a new page is loaded.

Handling New Window Requests

By default, if you click a link that requests a new window be opened to display the content of that link, nothing happens. If you want to change this behavior, your application needs to make a policy decision about new window requests. You need to implement a `WebUIDelegate` object to handle this case. Again, implement the delegate methods you want. Here are the steps to follow:

1. Set the delegate after `MyDocument.nib` is loaded. If you have a Cocoa document-based application, then you can add this code to MyDocument's `windowControllerDidLoadNib:` method:

```
[webView setUIDelegate:self];
```

2. It is good practice for browser-like applications to set the group name of WebView objects after they are loaded from a nib file. Otherwise, clicking on some links may result in multiple new window requests because the HTML code for a link might not use the same frame name. The group name is an arbitrary identifier used to group related frames. For example, you can set the group name as follows:

```
[webView setGroupName:@"MyDocument"];
```

3. Implement the `webView:createWebViewWithRequest:` delegate method to create a window containing a web view, and load the request:

```
- (WebView *)webView:(WebView *)sender createWebViewWithRequest:(NSURLRequest
*)request
{
    id myDocument = [[NSDocumentController sharedDocumentController]
openUntitledDocumentOfType:@"DocumentType" display:YES];
    [[[myDocument webView] mainFrame] loadRequest:request];
    return [myDocument webView];
}
```

4. Implement the `webViewShow:` method, which is invoked immediately after `webView:createWebViewWithRequest:`, to display the window:

```
- (void)webViewShow:(WebView *)sender
```

```
{
    id myDocument = [[NSDocumentController sharedDocumentController]
documentForWindow:[sender window]];
    [myDocument showWindows];
}
```

5. Build and run your application.

Now, when you click a link that requests a new window, a new window is created to display the content of that link. This is just one example of many decisions your application can make about policies and the behavior of the user interface. You use the other WebUIDelegate methods and other WebView delegates to modify the behavior.

Loading Pages

As the user navigates from page to page in your embedded browser, you may want to display the current URL, load status, and error messages. For example, in a web browser application, you might want to display the current URL in a text field that the user can edit.

Sequence of Frame Load Delegate Messages

As a webpage goes through the process of loading, the WebKit sends a series of messages to the frame load delegate. The exact sequence depends on the page content. SinceWebFrameLoadDelegate is an informal protocol a message is not sent to the delegate unless it responds to it. For example, the sequence of messages for a simple page that contains a title and loads successfully might be:

1. `webView:didStartProvisionalLoadForFrame:` —invoked at the start of a load.
2. `webView:willCloseFrame:` —invoked when the WebView is done with the old frame objects (passed as an argument) just before they are released.
3. `webView:didCommitLoadForFrame:` —invoked when a data source transitions from provisional to committed.
4. `webView:didReceiveTitle:forFrame:` —invoked when the frame title has arrived which is anytime after the data source is committed and before the load is finished. This method can be invoked multiple times.
5. `webView:didFinishLoadForFrame:` —invoked when all the data has arrived for a data source

Mostly, you implement the above delegate methods to display information about the webpage and load status. You might implement `webView:willCloseFrame:` if your application maintains references to the previous page content.

However, loading a page is a complicated process, so you have to anticipate that some client requests will fail. Because a client request is asynchronous over the network, the new page may not load immediately and errors may occur when loading. Typically, the default implementation does nothing (displays a blank view) if an error occurs; therefore, your application may want to display error messages instead. These delegate methods can be implemented to handle errors:

- `webView:didFailProvisionalLoadWithError:forFrame:` is invoked when an error occurred before any data was received. Typically invoked if the URL is bad or the network failed to deliver the request.
- `webView:didFailLoadWithError:forFrame:` is invoked when a committed data source fails to load.

Messages will also be sent to the delegate if the page content contains server redirects, or the scroll position within a frame changes (this can occur when the user clicks an anchor within a frame). See WebFrameLoadDelegate for more details on these and other delegate methods.

Testing for the Main Frame

The frame load delegate is notified if a location changes for any frame in the WebFrame hierarchy. Usually, you update the display only for changes to the main frame. For that reason, your code should always test to see whether the web frame, passed as an argument to the delegate method, is the main frame, as in this example:

```
// Only report feedback for the main frame.
    if (frame == [sender mainFrame]){
        [...
    }
```

Displaying the Current URL

Whenever the user clicks on a link in a webpage, the URL changes and a new page is loaded. By default, your application is not notified of this change. If you want to keep track of the current page, you need to implement some delegate methods, specifically a frame load delegate for your WebView object. Because WebFrameLoadDelegate is an informal protocol, you need to implement only the delegate methods you want.

For example, the user clicks a link and you want to update the text field to display the current URL. You do this by implementing the `webView:didStartProvisionalLoadForFrame:` delegate method as in this example:

```
- (void)webView:(WebView *)sender didStartProvisionalLoadForFrame:(WebFrame
*)frame
{
    // Only report feedback for the main frame.
    if (frame == [sender mainFrame]){
        NSString *url = [[[[frame provisionalDataSource] request] URL]
absoluteString];
        [textField setStringValue:url];
    }
}
```

Displaying the Page Title

If you want to display the page title, implement the `webView:didReceiveTitle:forFrame:` delegate method. For example, you can display the page title on the window, as in this example:

```
- (void)webView:(WebView *)sender didReceiveTitle:(NSString *)title
forFrame:(WebFrame *)frame
{
```

```
    // Report feedback only for the main frame.
    if (frame == [sender mainFrame]){
        [[sender window] setTitle:title];
    }
}
```

Similarly, you can implement the `webView:didReceiveIcon:forFrame:` delegate method if you want to display the page icon.

Displaying Load Status

Besides implementing the `webView:didStartProvisionalLoadForFrame:` method to display the page title, you can also use it to display the status, for example, "Loading." Remember that at this point the content has only been requested, not loaded; therefore, the data source is provisional.

Similarly, implement the `webView:didFinishLoadForFrame:`, `webView:didFailProvisionalLoadWithError:forFrame:` and `webView:didFailLoadWithError:forFrame:` delegate methods to receive notification when a page has been loaded successfully or unsuccessfully. You might want to display a message if an error occurred.

Loading Resources

Your application can also monitor the progress of loading content—that is, the progress of loading individual resources on a page. A resource is any data on a page that is loaded separately, such as images, scripts, style sheets and webpages contained in frames. A webpage can include multiple resources, each having their own request and response messages. Once a page is requested, each resource for that page can arrive independently and in any order. Some may load successfully, and others may not. If you want to be notified of the progress, success or failure of loading resources, you need to implement a resource load delegate for your WebView object.

Sequence of Resource Load Delegate Messages

As a resource goes through the process of loading, the WebKit sends a series of messages to the resource load delegate. The exact sequence depends on the resource and if an error occurred during the load. Since WebResourceLoadDelegate is an informal protocol a message is not sent to the delegate unless it responds to it. For example, the sequence of messages for a resource that loads successfully might be:

1. `webView:identifierForInitialRequest:fromDataSource:` —invoked before other delegate methods to return the application-defined resource identifier.
2. `webView:resource:willSendRequest:redirectResponse:fromDataSource:` —invoked one or more times before a request to load a resource is sent.
3. `webView:resource:didReceiveResponse:fromDataSource:`—invoked once when the first byte of data arrives.
4. `webView:resource:didReceiveContentLength:fromDataSource:`—invoked zero or multiple times per resource until all the data for that resource is loaded.
5. `webView:resource:didFinishLoadingFromDataSource:` —invoked when all the data for the resource has arrived.

If the resource load failed, then `webView:resource:didFailLoadingWithError:fromDataSource:` is invoked instead of `webView:resource:didFinishLoadingFromDataSource:`. Other messages per redirects can arrive, too. See WebResourceLoadDelegate for a complete list of delegate methods.

Identifying Resources

Because resource load delegates might need to distinguish between the different resources on a page, a resource identifier is passed as one of the arguments of the messages to delegates. The identifier remains the same for each load even if the request changes. For example, a request may change if headers are added, the URL is canonicalized, or the URL is redirected.

Your application can provide this identifier by implementing the `webView:identifierForInitialRequest:fromDataSource:` method to create and return a resource identifier. Otherwise, the resource identifier passed to subsequent delegate messages will not be unique. For example, this method can return a sequential number:

```
- (id)webView:(WebView *)sender
identifierForInitialRequest:(NSURLRequest *)request fromDataSource:(WebDataSource
 *)dataSource
{
    // Return some object that can be used to identify this resource
    return [NSNumber numberWithInt:resourceCount++];
}
```

Tracking Resource Load Progress

One reason for implementing a resource load delegate is to track the progress of individual resource loads. For example, you can keep track of the number of resources successfully and unsuccessfully loaded per page by implementing the following delegate methods. In this example, the resource status is displayed as "Loaded X of Y resources, Z resource errors." Each delegate method below increments these X, Y and Z values. Follow these steps to display the resource load status messages:

1. Add these instance variables to your delegate class:

    ```
    int resourceCount;
    int resourceFailedCount;
    int resourceCompletedCount;
    ```

2. Implement the `webView:resource:willSendRequest:redirectResponse:fromDataSource:` method to update the display when the WebKit begins to load a resource. Note that this method also allows you to return a modified request. Normally you don't need to modify it.

    ```
    -(NSURLRequest *)webView:(WebView *)sender
    resource:(id)identifier
    willSendRequest:(NSURLRequest *)request
    redirectResponse:(NSURLResponse *)redirectResponse
    fromDataSource:(WebDataSource *)dataSource
    {
        // Update the status message
        [self updateResourceStatus];
        return request;
    }
    ```

3. Implement the `updateResourceStatus` method, invoked in the sample code above, to use the `resourceCount` instance variable to update the display. You increment the `resourceCount` instance variable in the `webView:identifierForInitialRequest:fromDataSource:` method as shown in "Identifying Resources."

4. Implement the `webView:resource:didFailLoadingWithError:fromDataSource:` method to increment the number of failed resource loads and update the display as in:

    ```
    -(void)webView:(WebView *)sender resource:(id)identifier
    didFailLoadingWithError:(NSError *)error
    fromDataSource:(WebDataSource *)dataSource
    {
    ```

```
    resourceFailedCount++;
    // Update the status message
    [self updateResourceStatus];
}
```

5. Implement the `webView:resource:didFinishLoadingFromDataSource:` method to increment the number of successful resource loads, and update the display as in:

```
-(void)webView:(WebView *)sender
resource:(id)identifier
didFinishLoadingFromDataSource:(WebDataSource *)dataSource
{
    resourceCompletedCount++;
    // Update the status message
    [self updateResourceStatus];
}
```

6. Implement the `updateResourceStatus` method to update your display.

7. Also, implement the frame load `webView:didStartProvisionalLoadForFrame:` delegate method to reset these variables to `0` when a new page load starts.

8. Build and run your application.

When you run your application you should see a progress message showing the total number of resources per page, and progressively how many resources are loaded successfully and unsuccessfully.

Paging Back and Forward

A WebView object uses a WebBackForwardList object to maintain a list of visited pages used to page back and forward to the most recent page. WebView provides methods to handle the action of going forward or backward in this list; therefore, you can easily add Back and Forward buttons to your application.

Enabling and Disabling the Back-Forward List

By default, every WebView object maintains its own back-forward list object. There is some overhead in maintaining this list. If you don't want to use this feature, send a `setMaintainsBackForwardList:` message to your WebView object as in:

```
[webView setMaintainsBackForwardList:NO];
```

Otherwise, history items, which encapsulate visited page information, are automatically added and removed from the back-forward list as webpages are loaded and displayed in a WebView. If you maintain a back-forward list in your WebView object, then you should add some buttons to your user interface so that users can use this feature.

Adding Back and Forward Buttons

You don't have to write a single line of code to implement Back and Forward buttons in your application. Follow these steps to add these buttons:

1. Using Interface Builder, add a Back and Forward button to your window.
2. Using Interface Builder, connect the Back and Forward buttons to the WebView object by selecting, respectively, the `goBack:` and `goForward:` actions.
3. Build and run your application.

When running your application, connect to a URL, follow a few links (to generate some history items in your back-forward list), and then click the Back and Forward buttons. The WebKit maintains the back-forward list for you, and the action methods initiate new page requests.

Setting the Page Cache

You can use back-forward lists to maintain a page cache too. Because pages in the cache are rendered quickly, using this feature improves the performance of the back-forward action methods. You can turn this feature on or off by setting the cache size using the WebBackForwardList `setPageCacheSize:` method. If you set

the page cache size to `0`, the page cache is disabled; otherwise, the size of the cache is limited to the number of pages you specify. The default page cache size may vary depending on your computer's configuration. Use the `pageCacheSize` method to get the current setting.

Setting the Capacity

You can also limit the capacity of the back-forward list itself using the `setCapacity:` method. Setting the capacity affects only the number of history items allowed in the back-forward list. It doesn't affect the page cache. Use the `capacity` method to get the current setting.

The Current Item

If you manipulate a back-forward list directly, you need to be aware of how it works. A back-forward list is like an array that can expand in both the negative and positive directions. History items are typically inserted in the order in which they are visited. Items have indices which are relative to the current item. The current item is always at index `0`, the preceding item is at index `-1`, the following item is at index `1`, and so on.

When manipulating the contents of a back-forward list be aware that some methods change the current item and therefore the indices of all other items. The `goBack` and `goForward` methods, their equivalent action methods, and the `goToItem:` method all change the current item. All other WebBackForwardList accessor methods do not effect the current item.

Managing State

If there are no history items, or there are no items preceding or following the current item, then a back or forward action does nothing. Therefore you might want to track the state of the back-forward list and enable or disable the Back and Forward buttons accordingly.

You can do this by implementing the `webView:didFinishLoadForFrame:` delegate method to check the state of the back-forward list. Sending `canGoBack` and `canGoForward` to the WebView object returns `YES` if you can go in that direction. Therefore, you might implement the `webView:didFinishLoadForFrame:` method as follows, where `backButton` and `forwardButton` are your button outlets:

```
- (void)webView:(WebView *)sender didFinishLoadForFrame:(WebFrame *)frame
{
    // Only report feedback for the main frame.
    if (frame == [sender mainFrame]){
    [backButton setEnabled:[sender canGoBack]];
    [forwardButton setEnabled:[sender canGoForward]];
    }
}
```

Managing History

WebHistory and WebHistoryItem objects are used to maintain a history of all the pages a user visits. WebHistoryItem encapsulates all the information about a page that has been visited, and WebHistory maintains an ordered collection of these history items. WebHistory doesn't just maintain a linear list. It also keeps track of the days on which a user visits pages. Therefore, you can group and present history items to users by day. For example, you can use submenus to group history items by the last three visited days.

Sharing History Objects

WebHistory objects are not automatically created by the WebKit. You enable the history feature by creating a WebHistory object and assigning it to your WebView object.

You can create WebHistory objects in one of two ways. You can assign each WebView object its own WebHistory object, or you can create one WebHistory object that is shared by all WebView objects in your application as in:

```
// Create a shared WebHistory object
   WebHistory *myHistory = [[WebHistory alloc] init];
   [WebHistory setOptionalSharedHistory:myHistory];
```

Adding and Removing History Items

Like back-forward lists, history items are automatically added to the WebHistory object as the user navigates the web. If an item with the same URL already exists in the list, then it is replaced with the newer item even if the previous item was visited on a different day. Consequently, if the user revisits a URL it is always moved to the top of the list. Otherwise, items are not removed from the history list unless your application explicitly removes them.

For example, you could implement a clear history action by sending `removeAllItems` to your WebHistory object as in:

```
// Removes all the history items
- (IBAction)clearHistory:(id)sender
{
    [[WebHistory optionalSharedHistory] removeAllItems];
}
```

After removing all the items, the `removeAllItems` method posts a `WebHistoryAllItemsRemovedNotification` notification. The `addItems:` and `removeItems:` methods post similar notifications. You typically observe these notifications to update your display of history items as follows:

```
// Observe WebHistory notifications
```

```
    nc = [NSNotificationCenter defaultCenter];
    [nc addObserver:self selector:@selector(historyDidRemoveAllItems:)
              name:WebHistoryAllItemsRemovedNotification object:myHistory];
    [nc addObserver:self selector:@selector(historyDidAddItems:)
              name:WebHistoryItemsAddedNotification object:myHistory];
    [nc addObserver:self selector:@selector(historyDidRemoveItems:)
              name:WebHistoryItemsRemovedNotification object:myHistory];
```

The `userInfo` attribute of these notifications is an array containing the added or removed items. For example, you might implement the `historyDidAddItems:` method to add a corresponding menu item to your History menu for each new history item in the `userInfo` array.

Loading a History Item

Once the user has the ability to select a history item to load, you need to implement an action to perform the load. You send `loadRequest:` to the main frame of the appropriate WebView object, passing the selected WebHistoryItem's URL string as the argument. The following example assumes `historyItem` is the selected WebHistoryItem object:

```
- (void)goToHistoryItem:(id)historyItem
{
    // Load the history item in the main frame
    [[webView mainFrame] loadRequest:[NSURLRequest requestWithURL:[NSURL
URLWithString:[historyItem URLString]]]];
}
```

Saving and Loading History Objects

Unlike WebBackForwardList objects, WebHistory objects can be persistent—that is, they can be loaded from and saved to a writable URL. You send `loadFromURL:error:` to a WebHistory object to load history items from a readable URL, and you send `saveToURL:error:` to a WebHistory object to save history items to a writable URL.

Making Policy Decisions

You can make decisions about what web content to display in a WebView by implementing a policy delegate that conforms to the WebPolicyDelegate protocol.

The protocol is very flexible, allowing you to make a wide range of policy decisions. For example, you can implement a policy delegate to log URL requests, check for patterns you might not permit, block certain websites, block types of files, and even block IP addresses. The protocol provides the hooks for you to intercept requests—you implement the actual policy decisions.

For example, follow these steps to modify the MiniBrowser application located in `/Developer/Examples/WebKit` to ignore selected URLs:

1. Open `MyDocument.nib` using Interface Builder and set the WebView's `policyDelegate` outlet to `File's Owner` (the MyDocument instance).

2. Implement `webView:decidePolicyForNavigationAction:request:frame:decisionListener:` to filter the URL requests. In this example, URL requests with host names ending in `company.com` are ignored.

```
- (void)webView:(WebView *)sender decidePolicyForNavigationAction:(NSDictionary
 *)actionInformation
        request:(NSURLRequest *)request frame:(WebFrame *)frame
decisionListener:(id)listener {
    NSString *host = [[request URL] host];
    if ([host hasSuffix:@"company.com"])
        [listener ignore];
    else
        [listener use];
}
```

Enabling Editing

By setting the editable attribute of a WebView object you can make all the web content displayed in that view editable. Editing the content will change the underlying Document Object Model (DOM). However, setting the editable attribute to `YES` does not modify the editing attributes of the DOM objects—the WebView attribute setting overrides the DOM attributes.

For example, to modify the MiniBrowser application located in `/Developer/Examples/WebKit`, set the editable attribute to `YES` after the nib file is loaded in MyDocument's `windowControllerDidLoadNib:` method as follows:

```
- (void)windowControllerDidLoadNib:(NSWindowController *) aController
{
    [super windowControllerDidLoadNib:aController];
    ...
    // Set editable flag
    [webView setEditable:YES];
}
```

Now when you build and run the application, you can add, delete, and modify the HTML content displayed in a WebView object. See "Saving and Loading Web Content" (page 43) for how to get the HTML source from the DOM.

Saving and Loading Web Content

After the user edits the content of a WebView, you need some way to access the modified document. In a Cocoa document-based application, you typically allow the user to save and load the document.

For example, in the MiniBrowser application located in `/Developer/Examples/WebKit`, you would implement MyDocument's `dataRepresentationOfType:` method to return an NSData representation of the HTML source. Then implement MyDocument's `loadDataRepresentation:ofType:` method to transform an NSData representation to HTML source and load it into the WebView. Follow these steps to add saving and loading to the MiniBrowser application.

1. First add a variable and accessors to MyDocument to store the HTML source. Modify `MyDocument.h` as follows and implement the corresponding accessor methods in `MyDocument.m`:

```
@interface MyDocument : NSDocument
{
    ...
    // Editing Support
    NSString *_source;
}
...
// Editing Support
- (NSString *)source;
- (void)setSource:(NSString *)webContent;
@end
```

2. Next, implement MyDocument's `dataRepresentationOfType:` method to get the HTML source from the DOM, set the `_source` instance variable, and convert it to an NSData object as follows:

```
- (NSData *)dataRepresentationOfType:(NSString *)aType
{
    if (![aType isEqualToString:HTMLDocumentType])
        return nil;

    [self setSource:[(DOMHTMLElement *)[[[webView mainFrame] DOMDocument]
documentElement] outerHTML]];
    return [[self source] dataUsingEncoding:NSISOLatin1StringEncoding];
}
```

3. Then implement MyDocument's `loadDataRepresentation:ofType:` method to transform the NSData object to HTML source as follows:

```
- (BOOL)loadDataRepresentation:(NSData *)data ofType:(NSString *)aType
{
    if (![aType isEqualToString:HTMLDocumentType])
        return NO;

    [self setSource:[[[NSString alloc] initWithData:data
encoding:NSISOLatin1StringEncoding] autorelease]];
    [[webView mainFrame] loadHTMLString:[self source] baseURL:nil];
```

```
        return YES;
    }
```

Modifying the Current Selection

There are a number of WebView editing methods that allow you to modify the current selection. For example, you can replace the current selection with plain text as follows:

```
[webView replaceSelectionWithText:@"SomeString"];
```

You can replace the current selection with a styled string as follows:

```
NSString *markupString = [NSString stringWithFormat:
    @"<span style='color: red; font-style: italic'>%@</span>",
    @"SomeString"];
[webView replaceSelectionWithMarkupString:markupString];
```

Changing Editing Behavior

You can control or augment user editing by implementing a WebView editing delegate, an object that conforms to the WebEditingDelegate protocol. An editing delegate may receive *should* messages before or *did* messages after an editing action. Typically, you implement an editing delegate if you want to change the default editing behavior.

Should Methods

You implement should methods if you want to control user editing actions—these messages are initiated by a user action not by simply invoking WebView editing methods programmatically. The should methods are of the form `webView:should...` and may return `YES` to permit an action or `NO` to disallow an action. Optionally, the delegate can take an alternative action and return `NO` so the sender doesn't take additional action.

For example, you can control the insertion of text by implementing the `webView:shouldInsertText:replacingDOMRange:givenAction:` method. In this example, the user may use the Shift key to alter an insertion:

```
- (BOOL)webView:(WebView *)webView shouldInsertText:(NSString *)text
replacingDOMRange:(DOMRange *)range givenAction:(WebViewInsertAction)action
{
    if ([self shiftKeyIsDown]) {
        NSString *string = [NSString stringWithFormat:@"Big-%@", text];
        [webView replaceSelectionWithText:string];
        DOMRange *range = [webView selectedDOMRange];
        [range collapse:NO];
       [webView setSelectedDOMRange:range affinity:NSSelectionAffinityUpstream];
        return NO;
    }
    return YES;
}
```

Did Methods

A WebView editing delegate is automatically registered to receive notification of editing actions. The WebView editing delegate is sent a `webViewDid...` notification message—where *`sender`* is a WebView object—after the action takes place.

Using Undo When Editing

You can undo high-level web content editing operations—edits made programmatically and by the user—without writing any additional code. However, if you programmatically modify the Document Object Model (DOM), you need to write additional code to make those operations undoable.

For example, you can undo high-level operations programmatically as follows:

```
    // Delete content
    [webView deleteSelection];

    // Restore it
    [[webView undoManager] undo];
```

However, it's not as simple to undo a DOM operation. It's not good enough to return the DOM to an equivalent state—it must be exactly as it was originally. For example, you can't just undo deleted text by creating a new node and inserting it in the old location. The links between the nodes need to be exactly as they were before deleting the text.

One approach is to retain the deleted objects so that they are not released. For example, you might retain a deleted DOM node before removing it from the tree as follows:

```
 DOMNode *containerNode = [self containerNode];

    // Must retain before removing from the document
    deletedNode = [[containerNode lastChild] retain];

    // Now remove
    [containerNode removeChild:deletedNode];

    // Make undoable
    [[webView undoManager] registerUndoWithTarget:self
selector:@selector(undoAction) object:self];
```

You can then implement `undoAction` to return the DOM to its original state:

```
- (void)undoAction
{
    DOMNode *containerNode = [self containerNode];

    // Put the node back
    [containerNode appendChild:deletedNode];

    // Make redoable
    [[webView undoManager] registerUndoWithTarget:self
selector:@selector(redoAction) object:self];
}
```

The `redoAction` simply deletes the retained node again as follows:

```
- (void)redoAction
{
    DOMNode *containerNode = [self containerNode];

    // Take node out again
    [containerNode removeChild:deletedNode];

    // Make undoable
    [[webView undoManager] registerUndoWithTarget:self
selector:@selector(undoAction) object:self];
}
```

Using the Document Object Model from Objective-C

The Document Object Model API implements the Level 2 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 intention of the DOM Objective-C API is to conform to—as close as technically possible—the W3C DOM specification. Therefore, standard Cocoa conventions such as method naming, argument titling, and exception handling may not be reflected in this API. Following a few conventions discussed in this article, you can derive the DOM Objective-C API from the specification. This article also discusses the differences between the DOM specification and DOM Objective-C API.

The WebKit extensions to the DOM specification are covered in "Using the Document Object Model Extensions" (page 55). Refer to *WebKit DOM Programming Topics* for more information on the DOM specification.

Interpreting the DOM Specification

The DOM specification is written in Interface Definition Language (IDL) files available at the W3C web site. Links to them have been provided in the list that follows. The Objective-C DOM API is defined by header files in the WebKit framework, located at: `/System/Library/Frameworks/WebKit.framework`. Here is a list of the DOM specification's IDL files and their associated Objective-C header files:

W3C DOM IDL file	WebKit DOM header file
dom.idl	`DOMCore.h`
views.idl	`DOMViews.h`
events.idl	`DOMEvents.h`
html2.idl	`DOMHTML.h`
stylesheets.idl	`DOMStylesheets.h`
css.idl	`DOMCSS.h`
traversal.idl	`DOMTraversal.h`
ranges.idl	`DOMRange.h`

Two other header files are included in the Document Object Model API. `DOM.h` is a convenience header file that merely includes all of the other DOM header files. `DOMExtensions.h` defines additions that are not specified by the DOM specification. Use the extensions to help your DOM methods interact with the rest of the WebKit, and to use some of the WebKit's higher abilities such as HTML editing. More information is available in the "Using the Document Object Model Extensions" (page 55) article.

The interfaces specified by the DOM IDL files are mapped to Objective-C classes. The names of interfaces are unchanged when the specification does not conflict with the namespaces of Objective-C, Java, or other common languages. For example, the `DOMImplementation` interface in the DOM Core IDL is reflected by the same name in the Objective-C `DOMCore` header file. Where namespaces conflict, the API prefixes the conflicting interface appropriately. For example, the DOM Core IDL's `Node` interface appears as `DOMNode` in its Objective-C mapping.

This naming scheme also extends to constants. The API groups DOM constant lists into Objective-C `enum` types, and if a namespace conflict is present, the API prefixes them with an appropriate identifier. For example, the `ELEMENT_NODE` node type constant is reflected as `DOM_ELEMENT_NODE` in Objective-C.

Some names specified by the DOM specification may conflict with keywords or other reserved words in Objective-C. When this happens, they are given custom mappings and their Objective-C names must be inferred from the headers or from documentation. The API provides the custom mappings with appropriate names. For example, the two conflicting identifiers `id` and `name` (both Objective-C reserved words) are mapped to `idName` and `frameName`, respectively, and accurately reflect the intent of the specification.

Each Objective-C class derives, directly or indirectly, from the DOM root object `DOMObject`. This is an implementation detail that accommodates cross traffic between the WebKit and the DOM. The hierarchy of interfaces as defined by the W3C specification is also maintained. For example, `DOMDocument` extends from `DOMNode` in the specification as well as in the API.

Method names are mapped into Objective-C directly from the specification with no namespace transition. However, methods with multiple arguments in the DOM specification do not specify labels for their arguments—this behavior carries on into the Objective-C methods. For example, the specification declares this method prototype:

```
Node insertBefore(in Node newChild, in Node refChild);
```

The Objective-C version is *not*, as you might expect, changed to something like:

```
- (DOMNode *)insertNewChild:(DOMNode *)newChild
                BeforeOld:(DOMNode *)refChild
```

The actual Objective-C prototype is:

```
- (DOMNode *)insertBefore:(DOMNode *)newChild
                         :(DOMNode *)refChild
```

As you can see, this varies from typical Cocoa method-naming conventions. This is done for an important reason—to match the Document Object Model specification as closely as possible. But if you are a Cocoa developer new to the DOM, you may find the naming scheme confusing.

The API maps other objects to their appropriate equivalents. For example, it maps `DOMString` objects to `NSString`. All other types (such as `void`, `boolean`, `float`, and `unsigned long`) are mapped to their respective Objective-C types. Attributes which are both readable and writable are also given Cocoa-style get/set accessor methods. For example, the `DOMCore` IDL specifies this attribute:

```
attribute DOMString nodeValue;
```

Objective-C accesses this attribute using these methods:

```
- (NSString *)nodeValue;
- (void)setNodeValue:(NSString *)nodeValue;
```

Handling Exceptions

The API also maps exceptions from the DOM specification to Objective-C. Methods that raise DOM exceptions also raise Objective-C exceptions (`NSException`), but because Objective-C exceptions are not part of a method interface as they are in the DOM specification, the exception class names are mapped to string constants. The main exception classes are raised as `DOMException`, `DOMEventException`, and `DOMRangeException`, and can alert you with any number of exception codes. See `DOMCore.h`, `DOMEvents.h`, and `DOMRange.h` for the codes.

Using the Document Object Model Extensions

The WebKit provides extensions to the Document Object Model (DOM) that provide additional functionality not specified by the W3C recommendations. The DOM extensions are part of the DOM Objective-C API but are not part of the W3C DOM specification (and not implemented in JavaScript).

The extensions currently provide additions to `DOMHTMLElement`, `DOMDocument`, and `DOMRGBColor`. An additional `DOMHTMLElement` subclass, `DOMHTMLEmbedElement`, provides an Objective-C DOM class for HTML `embed` elements.

Among the useful features of the extensions are the inner/outer HTML and text accessor methods. Given an element block of HTML (a `DOMHTMLElement`), you can dynamically get and set the HTML and text from that block using these methods: `innerText` gets the inner content of the block without its HTML tags; `innerHTML` gets the inner content of the block (with its HTML tags, but not its enclosing tags); `outerHTML` gets the entire content of the block. For example, given this HTML block:

```
<DIV id="paras">
    <P>Paragraph 1</P>
    <P>Paragraph 2</P>
</DIV>
```

The `innerHTML` method will return (as an `NSString`):

```
<P>Paragraph 1</P>
<P>Paragraph 2</P>
```

The `innerText` method will return (as an `NSString`):

```
Paragraph 1
Paragraph 2
```

And the `outerHTML` method will return (as an `NSString`):

```
<DIV id="paras">
    <P>Paragraph 1</P>
    <P>Paragraph 2</P>
</DIV>
```

Each of those methods has a corresponding set method (`setInnerHTML`, `setInnerText`, `setOuterHTML`) and can be used on any element cast as a `DOMHTMLElement` or any subclass of it.

The addition to the `DOMRGBColor` interface is also very useful, as it allows you to use the DOM to access the CSS Level 3 alpha channel of an RGB(A) color, even though the DOM Level 2 specification does not include it.

For a complete list of extensions provided by the Objective-C DOM API, see the `DOMExtensions.h` header file.

Using JavaScript From Objective-C

The web scripting capabilities of the WebKit permit you to access JavaScript attributes and call JavaScript functions from your Cocoa Objective-C applications. Refer to *WebKit DOM Programming Topics* if you want to access the Objective-C API from JavaScript.

JavaScript objects are represented by instances of `WebScriptObject` in Objective-C. The API uses instances of `WebScriptObject` to wrap script objects passed from the scripting environment to Objective-C. You can use the methods of this class to call JavaScript functions and get/set properties of the JavaScript environment.

You should not create `WebScriptObject` instances explicitly. Rather, use the `windowScriptObject` method from `WebView` to gain access to the scripting environment.

Method parameters must be objects or basic data types like `int` and `float`. Objective-C objects will be converted to their JavaScript equivalents by the WebKit:

- `NSNumber` objects will be converted to JavaScript numbers.
- `NSString` objects will be converted to JavaScript strings.
- `NSArray` objects will be mapped to special read-only arrays.
- `NSNull` will be converted to JavaScript's null.
- All other objects will be wrapped by the WebKit into appropriate objects for the JavaScript environment.

Let's look at a couple of examples. You may want to get the URL of the current WebView from JavaScript, rather than accessing the URL from the data source of your WebView's `WebFrame`. You can do this with just a few lines of code:

```
id win = [webView windowScriptObject];
id location = [win valueForKey:@"location"];
NSString *href = [location valueForKey:@"href"];
```

JavaScript makes it easy to access the attributes of a web page's window. The WebKit makes it easy to get that information from JavaScript and pass it to Objective-C. Properties, such as `location` and `href`, are only available from the script object if the class overrides `isKeyExcludedFromWebScript()` to return `NO` for the given properties. The same is true for any selectors. But remember, a key feature of the web scripting system in the WebKit is the ability to call JavaScript functions. One of JavaScript's built-in functions, `location.href`, returns the URL of the window in one call. With this in mind, you can slim your three-line URL accessor in the example above down to one simple line:

```
NSString *href = [[webView windowScriptObject]
evaluateWebScript:@"location.href"];
```

You can also call JavaScript functions with arguments. Assume that you have written a JavaScript function which looks like this:

```
function addImage(image, width, height) { ... }
```

Its purpose is to add an image to a web page. It is called with three arguments: `image`, the URL of the image; `width`, the screen width of the image; and `height`, the screen height of the image. You can call this method one of two ways from Objective-C. The first creates the array of arguments prior to using the `WebScriptObject` bridge:

```
id win = [webView windowScriptObject];

NSArray *args = [NSArray arrayWithObjects:
                    @"sample_graphic.jpg",
                    [NSNumber numberWithInt:320],
                    [NSNumber numberWithInt:240],
                    nil];

[win callWebScriptMethod:@"addImage"
            withArguments:args];
```

The second version sends the arguments right to JavaScript:

```
[win evaluateWebScript:
    @"addImage('sample_graphic.jpg', '320', '240')"];
```

For more information on using the `WebScriptObject` to open JavaScript to Objective-C, read the *WebKit Objective-C Framework Reference*.

Spoofing

Some websites will deliver different content to different programs that ask for the same page. In extreme cases, a website may deny access completely to some programs. When this happens, you can try to gain access to the site by "spoofing" as another browser.

A client browser sends a special string, called a **user agent**, to websites to identify itself. The web server or JavaScript in the downloaded webpage, detects the client's identity and modifies its behavior accordingly. In the simplest case, the string includes an application name (for example, "Navigator") and version information (for example, 4.7 or 6.0). You can use these user-agent methods in WebView to make the identity of your application known and in some cases, to hide the identity of your application, a technique called **spoofing**:

- `setCustomUserAgent:` —sets the user-agent string.
- `setApplicationNameForUserAgent:` —sets the application name used in the user-agent string.

Note that some websites use the user-agent string to determine whether they support a client browser or not. If they don't, they may send dumbed-down versions of pages or deny access to the website too. In that case, you can modify the user-agent string to pretend to be a popular browser and then access the website. Although, this may not work if the website expects your application to implement browser-specific extensions. For this reason, you should only consider spoofing as a last resort.

Accessing the WebKit From Carbon Applications

To access the WebKit from a Carbon application, you use the Carbon WebKit API to create a Carbon web view. Once you have added the view to a window, you need to load and display URL content using native Cocoa classes.

Before using any web views, you need to call the `WebInitForCarbon` function. Doing so initializes Cocoa so you can access the WebKit from your Carbon application.

To create a web view, you simply call the `HIWebViewCreate` function. This function returns an HIView reference, and as such you can use any of the standard HIView manipulation functions on it. For example, you can embed the web view within another window or view, resize it, and so on.

For example, the following code fragment creates a web view and embeds it in a window:

```
WindowRef       window;
HIViewRef       webView, contentView;
HIRect          bounds;
CFURLRef        url;

// Create your window here
…

// Get a URL here, as a CFURL
…

WebInitForCarbon(); // initialize WebKit

HIWebViewCreate( &webView ); // create the web view

// Now obtain the content view associated with the window
HIViewFindByID( HIViewGetRoot( window ), kHIViewWindowContentID,
                &contentView );

// Set the bounds of the web view to be the same as the content view
HIViewGetBounds( contentView, &bounds );
HIViewSetFrame( webView, &bounds );

// Embed the web view in the content view and make it visible
HIViewAddSubview( contentView, webView );
HIViewSetVisible( webView, true );

// Make a call to load URL content */
LoadURL( webView, url );
```

To manipulate the contents of the web view, you need access to the actual Cocoa view. You obtain a reference to the Cocoa WebView object by calling the `HIWebViewGetWebView` function. You can then make Objective-C calls to the WebKit using that object. The example function `LoadURL` shows how you could do this:

```
static void LoadURL( HIViewRef inView, CFURLRef inURL )
{
```

```
    WebView*            nativeView;
    NSURLRequest*       request;
    WebFrame*           mainFrame;

    nativeView = HIWebViewGetWebView( inView ); // get the Cocoa view

    // Use Objective-C calls to load the actual content
    request = [NSURLRequest requestWithURL:(NSURL*)inURL];
    mainFrame = [nativeView mainFrame];
    [mainFrame loadRequest:request];
}
```

See *WebKit C Reference* for more details about the Carbon WebKit functions.

Note: For more detailed information about calling Cocoa methods from a Carbon application, see *Carbon-Cocoa Integration Guide*.

Determining WebKit Availability

On Mac OS X 10.2, the WebKit framework and the URL Loading system is only available on systems that have installed Safari 1.0. This article explains how a Mach-O application that use the WebKit framework or the URL Loading system, can run effectively on versions of Mac OS X that don't have Safari 1.0 installed.

If a user runs your application on a system that does not have the appropriate frameworks installed it will fail to launch, or crash at some point during execution. In order to prevent this you must take some precautions in your project.

Testing for URL Loading System Availability

The URL Loading system is available as of version 462.6 of the Foundation framework. To determine if the classes are available you can test the `NSFoundationVersionNumber` of the installed Foundation framework.

The example code in Listing 1 (page 63) will return `YES` if the URL Loading system is available.

Listing 1 Determining if the URL Loading system is available.

```
+ (BOOL)isURLLoadingAvailable
{
    return (NSFoundationVersionNumber >= 462.6);
}
```

Testing for WebKit Availability

An application can test for the availability of the WebKit by attempting to create a bundle for the framework using NSBundle. If the framework exists, the application can use the `load` method to dynamically load the framework.

The example code in Listing 2 (page 63) will return `YES` if the framework is installed and loads successfully.

Listing 2 Determining if the WebKit framework is available

```
+ (BOOL)isWebKitAvailable
{
    static BOOL _webkitAvailable=NO;
    static BOOL _initialized=NO;

    if (_initialized)
        return _webkitAvailable;

    NSBundle* webKitBundle;
```

```
    webKitBundle = [NSBundle
bundleWithPath:@"/System/Library/Frameworks/WebKit.framework"];
    if (webKitBundle) {
        _webkitAvailable = [webKitBundle load];
    }
    _initialized=YES;

    return _webkitAvailable;
}
```

Isolating Your WebKit and URL Loading System Symbols

Simply testing the version of the Foundation framework or if the WebKit framework is installed is not sufficient. If the application contains WebKit or URL Loading system symbols it can fail before it's able to execute the test code and inform the user of the problem.

The application must ensure that the main bundle does not include any symbols that may not be available at launch.

Conditionally Loading Code

One solution is to separate your WebKit dependent code into a bundle and load it only after determining that the framework is available.

You start by creating a new Bundle target in your application's Xcode project. This target should contain all your code that directly references WebKit classes, globals and types and is should be set to link against the WebKit.framework.

Your main application target in Project Builder should specify that it is dependent on the bundle and copy it to the application directory at compile time. Your application then checks the availability of the WebKit framework and, if present, uses `NSBundle` or `CFBundle` to dynamically load the code.

Weak Linking Symbols

Mac OS X 10.2 introduced support for weak linking in Mach-O applications. It works in a similar manner to the traditional Mac OS' Code Fragment Manager's weak, or soft imports.

The technical note "Ensuring Backwards Binary Compatibility - Weak Linking and Availability Macros on Mac OS X" describes the current support for weak linking of Carbon symbols. Currently, the same level of support is not available for Objective-C classes.

Due to the dynamic nature of Objective-C it is possible to avoid using linked symbols for class names by creating an instance of a class using the `NSClassFromString()` function.

```
Class webDownloadClass=NSClassFromString(@"WebDownload");
WebDownload *download=[[webDownloadClass alloc] initWithRequest:theRequest
                                                delegate:self];
```

This is equivalent to the following code that explicitly uses the `WebDownload` class.

```
WebDownload *download=[[WebDownload alloc] initWithRequest:theRequest
                                                  delegate:self];
```

Depending on your usage, it may also be necessary to dynamically load WebKit constants using CFBundle after determining that the framework is available. The example code in Listing 3 (page 65) demonstrates testing for the framework and loading a WebKit constant using CFBundle.

Listing 3 Loading WebKit constants dynamically using CFBundle

```
 CFURLRef url = CFURLCreateWithFileSystemPath(NULL,
                         CFSTR("/System/Library/Frameworks/WebKit.framework"),
                          kCFURLPOSIXPathStyle, TRUE);
    CFBundleRef bundle = CFBundleCreate(NULL, url);
    if (bundle != NULL) {
        NSString **WebHistoryItemsAddedNotificationPointer =
              (NSString **)CFBundleGetDataPointerForName(bundle,
                                 CFSTR("WebHistoryItemsAddedNotification"));
        if (WebHistoryItemsAddedNotificationPointer != NULL) {
            NSLog(@"looked up WebHistoryItemsAddedNotification");
            NSLog(@"location is %x, value is %@",
*WebHistoryItemsAddedNotificationPointer,
*WebHistoryItemsAddedNotificationPointer);
        } else {
            NSLog(@"found WebKit, but couldn't get the pointer");
        }
    } else {
        NSLog(@"no WebKit installed");
    }
```

See Also: Technical Note TN2064 - Ensuring backwards compatibility - Weak Linking and Availability Macros in Mac OS X 10.2

Document Revision History

This table describes the changes to *WebKit Objective-C Programming Guide*.

Date	Notes
2009-07-28	Added concurrency information.
2008-10-15	Minor edits throughout.
2006-05-23	Added information about using isKeyExcludedFromWebScript() to expose selectors or keys to JavaScript.
2006-03-08	Fixed code example in "Using JavaScript From Objective-C" article, fixed miscellaneous method links, and removed references to Project Builder.
2005-04-29	Changed availability of v10.4 features to v10.3.9 and later.
	Added articles on editing, JavaScript, and DOM. Changed the title from "Displaying Web Content."
	Added two new articles: "Making Policy Decisions" (page 39) and "Using Undo When Editing" (page 49).
2003-06-11	Added Carbon code example to "Accessing the WebKit From Carbon Applications" (page 61).
2003-06-06	First release of conceptual and task material covering the core WebKit classes and protocols available in Mac OS X 10.2 with Safari 1.0.