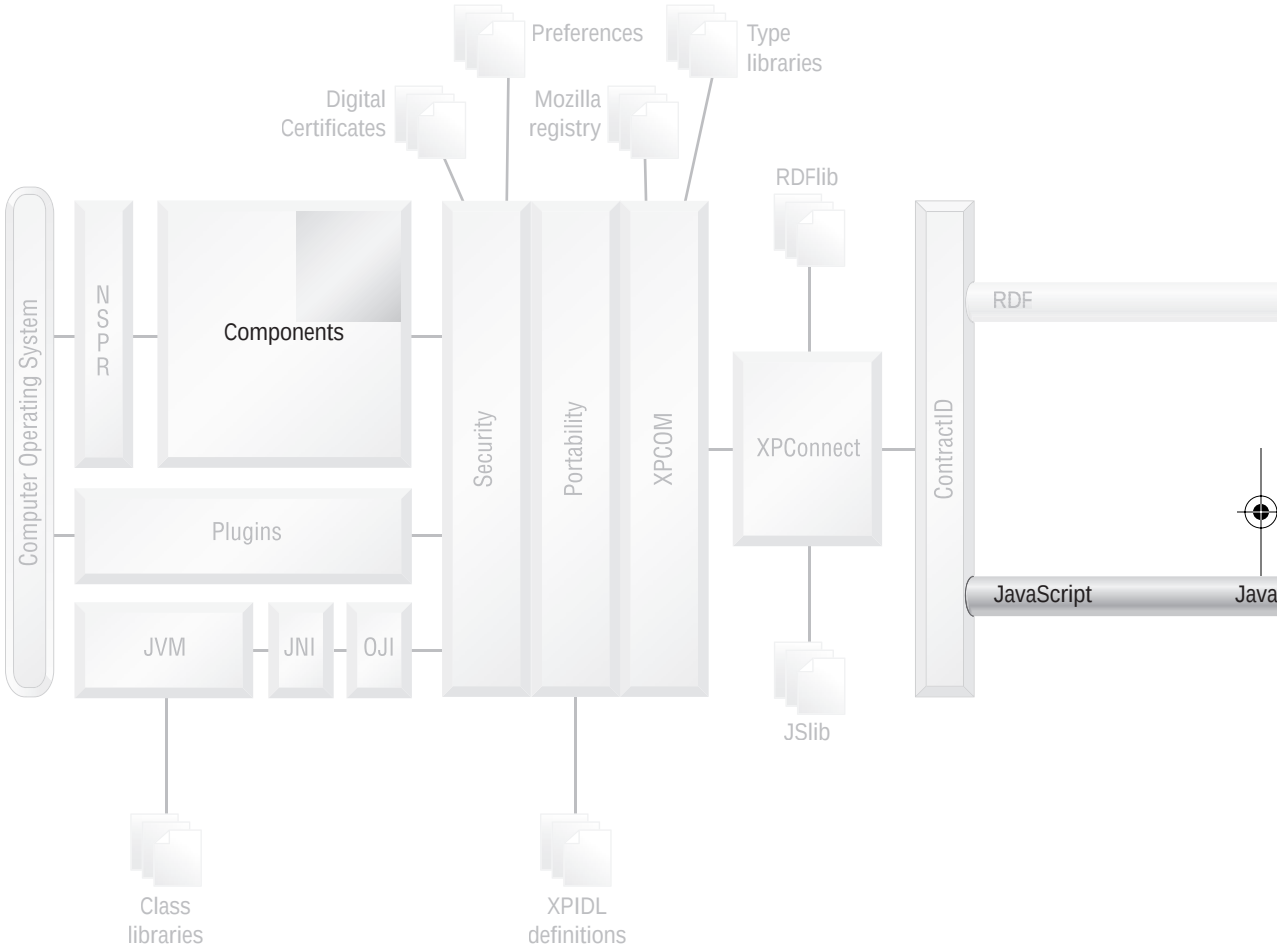
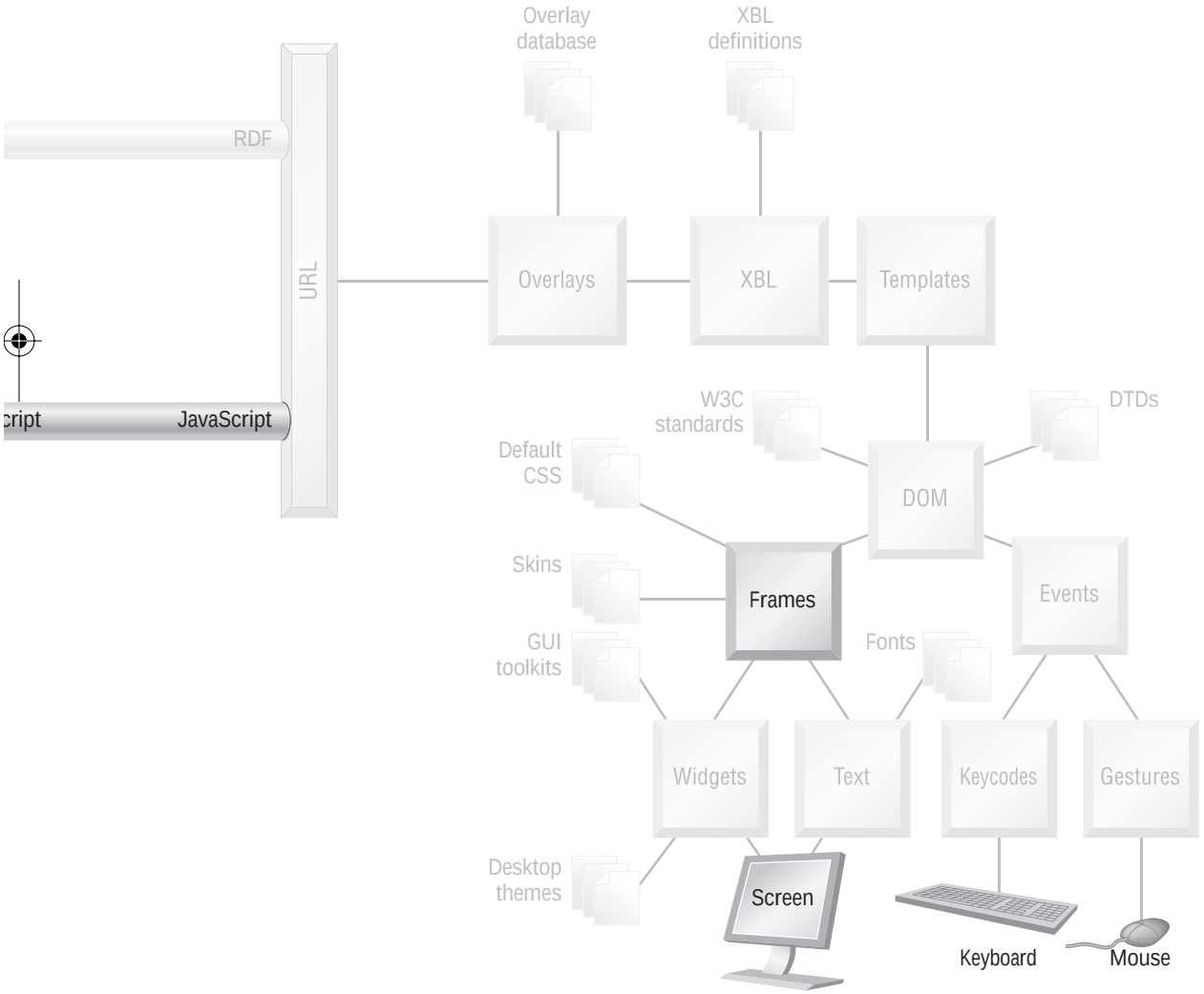

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
Plugins
JavaScript
Java
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 10

Windows and Panes



Only the most trivial of applications will fit a single window. Eventually, either part of the current window's content will need to be replaced, or a second window will be required. This chapter explains how to do both.

Some aspects of Mozilla are complicated; some aspects are simple. Working with windows is pretty simple. The tools available to XUL applications are much like the tools used in traditional Web development. If you've used the DOM method `window.open()` before, then this chapter will just be a review. Mozilla's XUL (and HTML) have a few new features to consider, but little described here is groundbreaking. Working with windows is just a matter of a few lines of code.

Mozilla provides a wide range of external windows ranging from tiny flyover help boxes (called tooltips by Microsoft and Mozilla) to large and complex application-specific XUL tags. Mozilla also provides internal (or inset or embedded) windows that appear as content inside other documents. Here these windows are called *panes*. Security hobbles devised for HTML browsers restrict these window types, but not when the application is installed in the chrome or is otherwise secure.

Mozilla's windows do contain a few surprises. XML namespaces can be employed to mix content types. There are improved tools to manage existing windows. Perhaps the most significant enhancement to window content is overlays. That subject is important enough to be covered separately in Chapter 12, Overlays and Chrome.

The NPA diagram at the start of this chapter shows the bits and pieces of Mozilla that have a little to do with window management. From the diagram, windows are a little about JavaScript and a little about frames. An object that exists outside the normal content of a document needs a container more complex than an ordinary `<box>` tag's frame, so there are several frame types to consider. There is also an XPCOM component or two worth considering. Overall, it's simple stuff.

10.1 ORDINARY `<WINDOW>` WINDOWS

Before looking at new XUL tags, we briefly revisit the `<window>` tag. The window tag supports the following special attributes:

```
sizemode windowtype screenX screenY width height
```

`sizemode` applies only to the topmost `<window>` (or `<dialog>`) tag. It records the appearance of the matching window on the desktop. It can be set to `normal`, `minimized`, or `maximized`.

`windowtype` is used to collect windows of like type. This attribute can be set to any programmer-defined string. This attribute takes effect when two or more windows with the same `windowtype` are displayed. Mozilla will make an effort to ensure that when the second or subsequent windows are opened,

that they are offset from the existing windows of the same type. This assists the user by preventing exact overlap of two windows.

`screenX` and `screenY` locate the window's offset from the top-left corner of the desktop in pixels.

`width` and `height` state the window's horizontal and vertical dimensions on the desktop, in pixels.

None of `screenX`, `screenY`, `width`, or `height` can be modified to affect the window's position once the window is displayed.

A dialog box can be built from a `<window>` tag, as described under the `window.open()` topic. On UNIX, it is not yet common for dialog boxes to be modal, so those semantics are not yet necessary. In general, it is nearly pointless to use a `<window>` tag for a dialog box when the `<dialog>` tag exists specifically for that purpose.

10.2 POPUP CONTENT

XUL content in a given window can be covered by additional content from the same document using these tags:

```
<tooltip> <menupopup> <popup> <popupgroup> <popupset>
```

Not all of these tags are recommended, however. `<menupopup>`, for example, should generally be used only inside a `<menu>` or `<menulist>` tag. The following XML attributes can sometimes be applied to other XUL tags:

```
tooltiptext grippytooltiptext popup menu context contextmenu
     contentcontextmenu
```

Not all these attributes are recommended either. The `<menupopup>` tag is discussed in Chapter 7, Forms and Menus. The other bits and pieces are discussed here.

The `<deck>` tag can also be used for visual effects. It is discussed in Chapter 2, XUL Layout.

10.2.1 Defining Popup Content

Mozilla's popup system has a little history associated with it. This history affects which tags are used to define popup content. It is best to use the XUL tags that are forward looking, rather than linger over the tags that were used in the past but that are now out of favor. Even so, older tags are still robust and unlikely to be desupported for any 1.x version of Mozilla.

There are three kinds of popup content: flyover help (tooltips), context menus, and ordinary menus.

When using the most modern XUL tags, state flyover help and context menu content inside the `<popupset>` tag. This is a user-defined tag with no special meaning, in the style of `<keyset>`. There may be more than one `<pop-`

upset> if required. Flyover help should be stated with the <tooltip> tag, and context menus, with the <menupopup> tag. Ordinary menus can also be stated inside the <popupset> using <menupopup>, but it is recommended that ordinary menus be stated elsewhere, using tags like <menu> and <menulist>.

Well-established but old-fashioned practices involve stating context menus with the <popup> tag. The <context> and <popupset> tags are rarely seen and even older than <popup>. These tags (and others) can be contained in a single very old <popupgroup> tag. <popupgroup> can only be used once per document. Do not use any of these three tags, even though <popup> is still common in the Mozilla chrome. They are deprecated and part of the past.

10.2.2 Flyover Help/Tooltips

Flyover help is a small window that appears over a GUI element (a XUL tag) if the mouse lingers in place more than about a second. The design intent is to give the user a short hint about the element in question, which might be cryptic (badly designed) or partially hidden. Tooltips can be fully styled, so the default yellow color is not fixed.

The <tooltip> tag supports the following attributes:

```
label crop default titletip
```

If the <tooltip> tag has no content, it acts like a label. The label attribute supplies label content, and crop dictates how the tooltip will be truncated if screen space is short.

If default is set to true, then the tooltip will be the default tooltip for the document. If titletip is set to true, then the tooltip is for a GUI element that is partially hidden. These two attributes apply only to tooltips that appear on the column headings of a <tree> tag.

The <tooltip> tag can contain any content—it acts like a <vbox>. Filling it with content is of limited value because the user cannot navigate across the face of the tooltip with the mouse pointer. Tooltips can be sized with minwidth and minheight, but that can confuse the layout of the parent window. Those attributes are not recommended.

To associate a tooltip with another tag, set the tooltip attribute of the target tag to the id of the <tooltip> tag. A shorthand notation that requires no <tooltip> tag is to specify the tooltip text directly on the target tag with the tooltiptext attribute. A further alternative is to include a <tooltip> tag in the content of the tag requiring a tooltip. If this is done, the parent tag should have an attribute tooltip="_child" set.

The grippytooltiptext attribute applies only to the <menubar> and <toolbar> tags and provides a tooltip for the <toolbargrippy> tag, when one is implemented.

The contenttooltip attribute applies only the <tabbrowser> tag and provides a tooltip for the embedded content of that tag.

A tooltip will also appear over the title bar of a window if the title text does not appear in full.

The currently visible tooltip's target tag can be had from the `window.tooltipNode` property.

10.2.3 Context Menus

Context menus are a form of dropdown menu. They appear the same as other menus, except that they are exposed by a context click (apple-click in Macintosh, right-click on other platforms). When the context click occurs, the context menu appears at the point clicked, not at some predetermined point in the document.

Context menus should be defined with the `<menupopup>` tag described in Chapter 7, Forms and Menus. Such a menu is associated with a target tag by setting an attribute on that tag to the `<menupopup>`'s id. The attribute set should be `contextmenu` for most tags and `contentcontextmenu` for the `<tabbrowser>` tag.

In the past, other attributes such as `popup`, `menu`, and `context` filled the role of the `contextmenu` attribute. They should all be avoided, even `context`, which still works.

The `popupanchor` and `popupalign` attributes apply to `<menupopup>` context menus, just as they do to normal `<menupopup>s`.

The currently visible context menu's target tag can be had from the `window.popupNode` property.

10.3 DIALOG BOXES

If new content to be displayed is more than trivial, then a dialog box is a simple design solution. Dialog boxes are no more than XUL windows with a very narrow purpose. Normally they are not standalone but operate in the context of (using the resources of) some other more general window.

Dialog boxes have two design purposes. The first is to expose complexity. When a user performs a command, the ideal result is that the command completes immediately and silently. Introducing a dialog box causes the user to deal with a set of further choices before their command completes. In terms of usability, this is an obstacle to the smooth flow of an application. The second design purpose is to focus the user on a specialist task. Such dialog boxes typically bring the application to a halt while the users tidy up the task thrust upon them by so-called modal dialog boxes. This second use is a disruption to the normal user navigation of the application.

Neither of these design goals is particularly compatible with a high-performance application. In a perfect world, there would be no dialog boxes at all. A well-designed application keeps dialog boxes to a minimum and avoids displaying meaningless configuration options. Too much configuration is a

symptom of poor process modeling at design time. If extensive configuration is required, it should be designed as an application in itself, like the Mozilla preference system.

When dialog boxes are required, Mozilla has plenty of support.

10.3.1 <dialog>

The <dialog> tag is the main building block for dialog boxes. It is a replacement for the <window> tag and adds extra content and functionality to a standard XUL window. The <dialog> tag does not create a modal dialog box (one that seizes the application or desktop focus) unless additional steps are taken.

To use the <dialog> tag, just replace <window> with <dialog> in any XUL document. Beware that the <dialog> tag is designed to work from within the chrome, and using it from a nonchrome directory on Microsoft Windows can freeze Mozilla completely. No such problems occur on UNIX. Figure 10.1 compares the different uses of <dialog>.

In this example, standard diagnostic styles are turned on, and the word *Content* is a <description> tag that is the sole content of the document. In the top line of the image, the same content is displayed, first with the <dialog> tag and then with the <window> tag. The <dialog> tag adds extra structural <box> tags to the document plus two <button> tags. In the second line of the image, the buttons attribute of the <dialog> tag has been used to turn on all the buttons that <dialog> knows about. In the last line, extra content has been added to the document in the form of six buttons. These buttons override the standards <dialog> buttons, and so those standard buttons don't appear in the bottom box where they otherwise would. Note that the button text in the Full and Modified cases differs. Let's examine how all this works.

The <dialog> tag has an XBL definition that supplied boxes and buttons as extra content. In the normal case, four of these buttons are hidden.

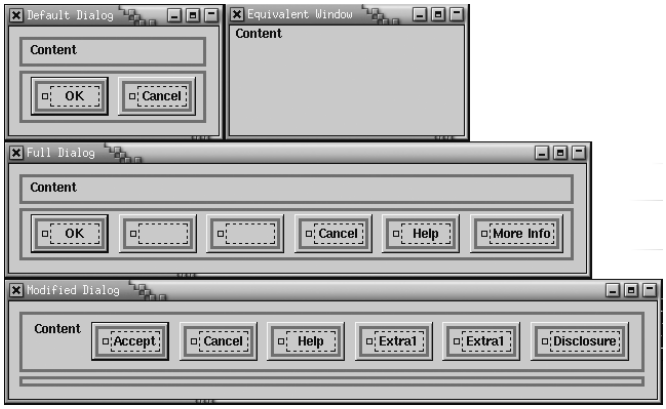


Fig. 10.1 Variations on the use of the <dialog> tag.

The tag also relies on a properties file in the chrome. That file contains button label text suitable for the current platform. Access to this properties file is one good reason for installing dialog boxes in the chrome. The <dialog> tag supports the following attributes:

```
buttonpack buttondir buttonalign buttonorient title buttons
```

The buttons attribute is a comma-separated list of strings. Spaces are also allowed. It is a feature string, similar to the *feature string* used in the window.open() method, but it's simpler in syntax. Allowed values are

```
accept cancel extra1 extra2 help disclosure
```

These values do not need to be set to anything. An example of their use is

```
<dialog buttons="accept,cancel,help"/>
```

The following event handlers, plus those of the <window> tag, are also available:

```
ondialogaccept ondialogcancel ondialogextra1 ondialogextra2
     ondialoghelp ondialogdisclosure
```

The title attribute sets the content of the title bar of the window, as for the <window> tag. buttonpack/dir/align/orient serve to lay out the <hbox> at the base of the dialog window as for the standard pack/dir/align/orient attributes. The buttons attribute states which of the standard dialog buttons should appear.

These standard buttons appear with standard content and standard behavior. For example, the cancel button shuts down the dialog box. The extra1 and extra2 buttons are available for application use, but they must include a replacement for the properties file if those buttons are to have labels. The event handlers noted earlier match the buttons one-for-one and fire when the buttons receive a command event (when they are clicked).

These <dialog> attributes are sufficient to explain the first three variations in Figure 10.1, but not the bottom one. That last variation depends on the dlgtype attribute, which typically appears on a <button> tag.

The dlgtype attribute can be set to any one of the standard button types:

```
accept cancel extra1 extra2 help disclosure
```

The dlgtype attribute has no meaning to the <button> tag itself, or to whatever tag it appears on. Instead, the <dialog> tag is quite smart. It searches all the content of the dialog window for a tag with this attribute, and if it finds one, it uses that tag instead of the matching dialog button. This brings flexibility to the dialog by allowing the application programmer some control over the placement of the dialog window controls. Only tags that support a command event may be used as replacement tags. If the replacement tag does not have a label attribute, it is supplied one by the <dialog> tag from the standard set of property strings.

A <dialog>-based window can be opened the same way as any other window. To add modal behavior, use the window.openDialog() method discussed shortly.

A <dialog> tag should not be explicitly sized with width and height attributes.

10.3.2 <dialogheader>

Although this tag is defined in the chrome, it is simple content only. It provides the heading area inside the Preferences dialog box, but it's no more than a <box> and some textual content. It has no special meaning to dialog boxes.

10.3.3 <wizard>

The <wizard> tag is an enhancement of the <dialog> tag. It and <wizardpage> are discussed in Chapter 17, Deployment.

10.3.4 Java, JavaScript, and Others

For completeness, note that dialog windows can also be created from Java, using standard Java techniques. The JVM and standard class libraries can be used to create windows that have nothing to do with the semantics of the Mozilla Platform. Such dialog boxes can exist outside the rectangular browser area occupied by HTML-embedded applets and can survive the destruction of the window that created them. They can appear in their own windows as the Java Console does. A starting point for such dialog boxes is to create a canvas.

Dialog windows can also be created from JavaScript using XPCOM and AOM/DOM resources. Some techniques for doing so are discussed next.

Embedded programmers can use languages such as Perl and Python to drive all the windows that an application based on the Mozilla Platform creates. Such uses are not documented here.

10.4 JavaScript Window Creation

New application windows can be created from JavaScript.

Recall that the JavaScript interpreter environment supplies the application programmer with a global object. In a XUL document, that global object has a property named window that is effectively the global object. This object supports many methods useful for creating windows.

The Mozilla hierarchy of JavaScript properties is a constructed system that is separate from the many XPCOM interfaces. Although some JavaScript properties, like window and window.document, appear to be identical to XPCOM components, these properties are just convenient fakes hiding the real components that make up a Mozilla window.

This last point is important. It is easy to become convinced that AOM JavaScript properties exactly match a particular Mozilla component, but this is not always true. Deep inside Mozilla, a number of separate abstractions make up a window displaying document content. It is best to view window and document as separate systems that closely match a few of the XPCOM interfaces rather than as direct replicas of some unique platform object.

The DOM Inspector, while very useful for examining content nodes like DOM elements, is not as revealing for the window and document JavaScript objects. For these objects, available JavaScript properties should be compared against the XPCOM XPIDL interfaces so that differences can be seen.

The window object reports its type as `ChromeWindow`. There is no such XPCOM object, although there is an `nsIDOMChromeWindow`. Interfaces more relevant to the features of the `ChromeWindow` object are `nsIJSWindow` and `nsIDOMWindowInternal`.

10.4.1 Navigator Versus Chrome Windows

JavaScript access to windows is designed with the traditional Web developer in mind. By default, a new window means a new Navigator window. A Navigator window is a XUL window that contains a standard set of XUL content that supplies all the features of a Web browser. This includes toolbars, other window decorations, bits of the DOM 0 document object model, a pane in which to display Web pages, and security. The alternative to a Navigator window is a chrome (plain XUL) window.

Navigator windows present a restricted interface to Web developers. The `window` and `window.document` objects apply only to the content loaded into the content pane. Web developers cannot access the XUL content that surrounds that pane; to such a developer, the XUL Navigator controls are a black box. The only options available to Web developers are a few flags that can be supplied to the `window.open()` method. These flags allow a new Navigator window to hide some of the XUL content, like toolbars.

To a XUL programmer, such Navigator windows are either a blessing or a nuisance. If the programmer is making minor enhancements to the existing Navigator system (such as overlays), then a Navigator window provides a great deal of functionality for free. If, however, the programmer is building a new application, all the Navigator functionality is just unwanted junk. In that case, simple steps (like the `-chrome` option) must be taken to avoid that content.

10.4.2 `window.open()`

The `window.open()` method opens an independent window. It is a DOM 0 interface commonly used in Web development. Its signature in Mozilla is

```
target = window.open(URL, name, features);
```

- `URL` is a valid URI and can be set to a zero-length string or "`about:blank`" for empty content. It does not need to be URL encoded.
- `name` is an identifier for the new window. It can be set to `null` or to "`_blank`" for a nameless window.
- `features` is a comma-separated list of feature keywords (a feature *string*) that controls the appearance and behavior of the new window.
- Finally, the returned value is a reference to the global (window) object of the target window. The target window's AOM can be accessed from the window that keeps this reference, provided that any security restrictions are obeyed.

The feature string is widely used but poorly documented, so here is a full discussion.

Each feature stated is a `name=value` pair. It is case insensitive. Such pairs are either boolean flags or a scalar. In the boolean case, the `=value` part can be left off or set to "`=yes`" for a value of `true`. Anything else is a value of `false`. In the scalar case, the `=value` part is a number, and any trailing characters are ignored. The features must be separated by commas and may have additional spaces, although spaces present portability problems for HTML documents displayed on older browsers. To summarize in an example:

```
"toolbar,location=yes, menubar=no,top=100, left=100px,width=100cm"
```

This example turns the Navigation toolbar on, the Location toolbar on, and the menu bar off. It positions the window 100 pixels from the top and the left, making it 100 pixels wide (`cm` is ignored).

The feature string dictates whether a window will be a Navigator browser window (the default) or a plain XUL chrome window. Navigator is the default. A plain chrome window requires this feature:

```
chrome
```

`chrome` defaults to `false`. If it is left off, then the following features are specific to a Navigator window:

```
toolbar location directories personalbar status menubar scrollbars
     extrachrome
```

These features default to `true` if no feature string at all is provided.

The `extrachrome` feature is a boolean value that determines whether a new Navigator window will load and display user-defined content (called the sidebar in the Classic Browser). In technical terms, this feature enables or disables the loading of programmer- or user-defined overlays installed in the chrome. The other features are standard Web options.

The following features apply to both Navigator and chrome XUL windows:

```
resizable dependent modal dialog centerscreen top left height width
     innerHeight innerWidth outerHeight outerWidth screenX screenY
```

If stated without a value, `resizable` and `centerscreen` default to `true`; `dependent` and `modal`, to false. An exception is that if the current window is `modal`, then the new window is made to be `modal=true`. The remainder are scalars and so must have a value supplied.

`resizable` enables the resize drag points supplied to the window by Microsoft Windows and X11 window managers. These resize drag points are separate from any `<resizer>` tag.

`dependent` ensures that the new window will be closed if the parent window is closed. The new window will always be on top of the invoking window.

`modal` ensures the original window cannot get focus until the new window is closed.

`dialog` tells the window manager to strip all buttons, except the Close button, from the new window's title bar.

`centerscreen` positions the new window in the middle of the desktop.

The remaining options are scalars that affect the position and size of the new window. If just one of `top`, `left`, `screenX`, or `screenY` is set, the other direction will be set to zero. The sizing and positioning features for a Navigator window are overridden if the `persist` attribute is used, which it is for Classic Browser windows.

Some features work only if the window is created with security privileges, for example from the chrome. These features are

```
titlebar close alwaysLowered z-lock alwaysRaised minimizable
```

`titlebar` and `close` default to `true`; the others, to `false`.

`titlebar` enables the decorations added by the window manager: the title bar and window frame borders. `close` enables the Close window button on the title bar. `alwaysLowered` and `z-lock` keep the new window behind all other windows when it gets the focus; `alwaysRaised` keeps the window in front of other windows when it loses the focus. `minimizable` appears to do nothing.

If the `close` feature is turned off, the window can still be closed by the user using the window manager. For example, on Microsoft Windows, the Windows icon in the task bar retains a close menu item on its context menu.

10.4.3 `window.openDialog()`

The `window.openDialog()` method is available only when the document calling this method is secure. Installed in the chrome is a secure case. It takes nearly the same arguments as the `window.open()` method and produces nearly the same result.

```
win = window.openDialog(URL,name,features,arg1, arg2, …);
```

`arg1`, `arg2`, and so on are arbitrary JavaScript function arguments passed to the `window.arguments` array of the new window.

The openDialog() method is equivalent to adding the feature string "chrome=yes,dialog=yes". If either of those features is set to false by the feature string, the values supplied override the openDialog() defaults. openDialog() also supports this feature keyword:

```
all
```

The all attribute turns on all features that grant the user control over the window. That means the Close and Minimize icons on the title bar and so on.

10.4.4 window.content, id="content" and loadURI()

In a Classic Browser window, the XUL tag with id="content" is a <tabbrowser> tag that represents the content panel. This tag is invisible to Web developers, but highly useful for application developers who are customizing browsers. It is heavily loaded with scriptable functionality and is worth exploring with the DOM Inspector.

If the window is a standard XUL-based browser window, then this <tabbrowser> object is available as the content property of the window object. In addition to implementing the nsIWebNavigation interface, that content property has a document property that points to the HTML (or XML) document that appears in the current tab.

The nsIWebNavigation interface provides all the methods used to control the in-place loading, reloading, and navigating of content in a single browser content pane. The <tabbrowser> AOM object has a webNavigation property that exposes this interface. The most useful method is loadURI(). loadURI() is similar to the XmlHttpRequest object discussed in Chapter 7, Forms and Menus. The main difference is that the XmlHTTPRequest object returns an HTTP request's response as pure data, whereas loadURI() also stuffs the response document into the display so that it is visible and integrated with other Classic Browser features like history. loadURI() has this signature:

```
void loadURI(uri, flags, referrer, postData, headers)
```

- uri is the Web resource to display.
- flags (default null) is a set of bitwise ORed flags that modify the load behavior. These flags are supplied as webNavigation properties prefixed with LOAD_FLAGS and provide options to bypass the browser cache, reload the page, and so on.
- referrer (default null) is the HTTP referrer for the load request. The programmer may override the referrer the Mozilla Platform supplies with this argument.
- postData (default null) is a string of HTTP POST request data.
- headers (default null) is a string of additional HTTP headers.

The last two options are rarely used from JavaScript. If you require them, then the string supplied must be specially crafted to match the `nsIInputStream` interface. To do so, use XPCOM as shown in Listing 10.1.

Listing 10.1 Construction of a string-based `nsIInputStream`.

```
var str, C, obj, iface, arg;

str   = "put data here";
C     = Components;
obj   = C.classes["@mozilla.org/io/string-input-stream;1"];
iface = C.interfaces.nsIStringInputStream;

arg = obj.createInstance(iface);
arg.setData(str, str.length);

loadURI("page.cgi", null, null, arg, null);
```

10.4.5 `alert()`, `confirm()`, and `prompt()`

These three methods of the window object are basic user feedback windows in the style of Microsoft's `MessageBox` functionality. They are specialized, modal dialog boxes. `alert()` is also the simplest way to put a breakpoint in your JavaScript code: Execution halts until the alert is dismissed. See any Web development book.

The small dialog windows created by these methods have been redesigned for Mozilla. In Netscape 4.x and earlier, and in Internet Explorer, these windows hold the output of a single C-like `printf()` statement. There was no support for Unicode characters, only ASCII. In Mozilla, these windows are full XUL windows, as Figure 10.2 reveals.

Dotted boxes in this diagram represent `<description>` tags. To see this structure, apply diagnostic styles to the `userChrome.css` file in your Mozilla user profile directory and restart the platform. This screenshot was produced with this line of code:

```
alert("Line 1\nLine2");
```

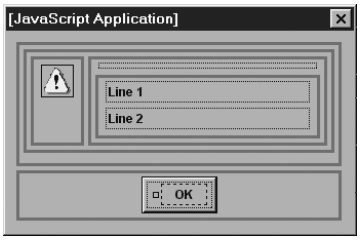


Fig. 10.2 XUL structure of an `alert()` dialog box.

Mozilla takes the string supplied to `alert()`, `confirm()`, and `prompt()` and splits it on all end-of-line characters supplied. Each split piece is allocated a `<description>` tag and, therefore, can wrap over multiple lines if required. Each `<description>` tag has a `maxwidth` of `45em`. An empty description tag at the top can be styled by the application programmer and has `id="info.header"`.

These dialog boxes originate in the chrome, in files prefixed `commonDialog` in `toolkit.jar`. The same XUL document is used for all three dialog boxes.

10.4.6 `nsIPromptService`

This XPCOM pair is the technology behind the `alert()`, `confirm()`, and `prompt()` dialog boxes:

```
@mozilla.org/embedcomp/prompt-service;1 nsIPromptService
```

An object can be created by this pair only when the document has full security access, but such an object can create a wide variety of small dialog boxes. Table 10.1 lists these boxes.

The `nsIPromptService` interface contains extensive documentation on the arguments for each of these methods. Use this interface when building a chrome-based application, rather than the simpler `alert()`. `alert()` is really intended for HTML.

Table 10.1 Mozilla CSS2 font name extensions

Method name	Dialog box created
alert()	Same as the AOM alert().
alertCheck()	Same as the AOM alert(), but with an extra line containing a checkbox and a text message.
confirm()	Same as the AOM confirm().
confirmCheck()	Same as the AOM confirm(), but with an extra line containing a checkbox and a text message.
confirmEx()	A fully customizable dialog box with up to 3 buttons, each with custom or standard text, and an optional checkbox and message.
prompt()	Same as the AOM prompt().
promptUsernameAndPassword()	A dialog box showing a user name and password field, and an optional checkbox and message.
promptPassword()	A dialog box showing a password field, and an optional checkbox and message.
select()	A dialog box showing a `<listbox>` from which a single item may be selected. Each item is plain text.

10.4.7 Special-Purpose XPCOM Dialog Boxes

Mozilla provides several special-purpose dialog boxes, all of which are located in `toolkit.jar` in the chrome. The following dialog boxes are implemented: Ask the user to specify a file; assist the user with printing; assist the user when searching content in pages; and report progress for a file being downloaded.

All of these special-purpose dialog boxes have a matching XPCOM component and must be controlled via XPCOM interfaces, not via the URLs of their chrome components.

Two of these dialog boxes are briefly examined next.

10.4.7.1 FilePicker The FilePicker is an XPCOM component and interface. It is also a set of dialog windows. Sometimes these dialog windows are Mozilla windows containing XUL, and sometimes they are standard file dialog windows provided by the operating system. Such native dialog windows cannot be inspected with the DOM Inspector.

The following XPCOM technology is used to implement the FilePicker:

```
component @mozilla.org/filepicker;1 interface nsIFilePicker
```

The `nsIFilePicker` interface has very straightforward methods for customizing the dialog box, as a brief examination of the XPIDL file will reveal. To use the dialog box, proceed as follows:

1. Create a FilePicker object using XPCOM.
2. Initialize it with the `init()` method.
3. Add filters and choose which one is selected.
4. Set any other defaults required.
5. Call the `show()` method, which blocks until the user responds.
6. Extract the user's response from the object.

The last stage, extracting the user's response, is not as straightforward as it might seem. Files nominated by the user cannot be specified with a string containing a simple path. This is because some operating systems require a volume as well as a path in order to specify a file's location fully. The Macintosh (and VMS, mainframes, and others) are examples. Chapter 16, XPCOM Objects, describes the objects and interfaces Mozilla uses to manage files. That is recommended reading before attempting to use the FilePicker.

Listing 10.2 is an example of the FilePicker used to ask the user for a file that is to be the location of written-out content.

Listing 10.2 Acquiring an `nsILocalFile` using the FilePicker dialog box.

```
var Cc = Components.classes;
var Ci = Components.interfaces;
var fp;
```

```
fp = Cc["@mozilla.org/filepicker;1"];
fp = fp.createInstance(Ci.nsIFilePicker);
fp.init(window, "Example File Save Dialog", fp.modeSave);
fp.show();

// fp.file now contains the nsILocalFile picked
```

10.4.7.2 `PrintingPrompt` Mozilla provides XPCOM objects that interface to the native printing system, objects that provide native printing dialog boxes, and some print dialog windows that are built from XUL. In very recent versions of Mozilla, XUL documents can be printed as for HTML documents. Just load the XUL content in a browser window and print. Before version 1.3 or thereabouts, only HTML documents could be printed.

The printing system is nontrivial and is not covered in detail here. Starting points for exploring the printing system are the `nsIPrintingPromptService` and `nsIWebBrowserPrint` XPCOM interfaces.

10.5 EMBEDDING DOCUMENTS IN PANES

The alternative to putting content in a new window is to create a pane in an existing window whose content can be flexibly managed. A traditional television is such a device, with many channels but only a single pane for display. In this book, a *pane* is different from a *panel*. A panel is an area of a window that displays related information in one spot. A pane is a panel whose related information is sourced from a document that is separate from the rest of the window.

Mozilla's XUL allows unrelated content to appear in part of an existing window. Several tags that can achieve this exist.

10.5.1 `<iframe>`

The `<iframe>` tag is at the core of all Mozilla's document embedding solutions. It has the same purpose as an HTML `<iframe>`, but it is less flexible. No content can be specified between start and end `<iframe>` tags, and it cannot float beside other content. A XUL `<iframe>` allows focus to enter the content in the `<iframe>`'s area. XUL `<iframe>`s can be sized and resized as for any boxlike tag:

```
<iframe minwidth="100px" flex="1" src="about:blank"/>
```

The `<iframe>` tag has only one special attribute: `src`. This can be set to the URL of the document to be displayed in the `<iframe>`. Since the contents are often HTML, it is useful to set the `name` attribute as well so that the window to which the content belongs has a name that can be used from inside the content.

In Chapter 8, Navigation, it was remarked that the <scrollbox> tag represents a special case of a boxlike tag, with its own interface. The <iframe> tag is another such special case. It is the sole example of a component named

```
@mozilla.org/layout/xul-boxobject-iframe;1
```

The AOM object for the <iframe> tag has a boxObject property as for all XUL tags that are boxlike. Like <scrollbar>, <iframe>'s boxObject.QueryInterface() method can be used to retrieve a special-purpose interface. In this case, it is called nsIIFrameBoxObject. Although this interface has only one property, docShell, that property in turn reveals the nsIDocShell interface, which is extensive.

DocShell stands for "Document Shell." A DocShell is a shell for documents in the same way that ksh, bash, or DOS is a shell for a set of operating system commands. It is used to manipulate whole documents, just as DOS is used to manipulate whole files and programs.

The extensive properties and methods of the nsIDocShell interface provide all the basic operations required to load and manage a document retrieved by the Mozilla Platform. This interface can also be used to access all the other interfaces that are handy for document management. The nsIDocShell interface is therefore something like the driver's seat of a car—most of the controls are within easy reach of it. This interface is well commented and worth looking over. The downloadable files described in the Introduction include this interface.

In simple cases, an application programmer merely uses an nsIDocShell that appears as a result of some content being loaded. Only in ambitious projects does an application programmer need to hand-create a document shell and drive the loading of documents through scripts.

Since the <iframe> tag supports this interface, its contents must have the status of a full XML document. Most boxlike tags only have the status of a single DOM element object and its children. The <iframe> tag is the fundamental tag required to create a Mozilla Browser window: It implements the pane in which the downloaded content appears. It is the heart of the Browser window.

To summarize, the <scrollbar> tag has a simple box-object enhancement that allows its content (an XML document fragment) to be moved within the tag's display area. The <iframe> tag's box-object enhancements are far more complicated: Its content (a whole XML document) can be extensively managed within the tag's display area. In both cases, the display area is an undecorated plain rectangle. Clearly, <iframe> is an extensive enhancement on a plain <box>.

<iframe> has an XBL definition in general.xml and in toolkit.jar in the chrome. This definition provides properties that are shorthand for commonly used parts of nsIDocShell:

```
docShell contentWindow webNavigation contentDocument
```

- docShell is the starting point for management of the content.
- contentWindow is the equivalent of the JavaScript window property for the content.
- webNavigation is the window.webNavigation property for the content.
- contentDocument is the window.document property for the content.

The commandDispatcher that is supplied with every XUL document has an advanceFocusIntoSubtree() method that can move the focus into an <iframe>'s content, if that content is capable of receiving the focus.

If an <iframe>'s content is taller or wider than the clipping area of the frame, and the pane's content is not XUL, then scrollbars will appear. Any XUL document displayed in an <iframe> should start with a <page> tag. <iframe> tags may be nested, provided that each level of nesting is contained in a complete document.

10.5.2 <page>

The <page> tag is an alternative to the <window> tag. The <page> tag should be used when the XUL document is to appear inside an <iframe>, rather than in its own window. The <page> tag has no special attributes.

A document that uses <page> can still be displayed in a window by itself, but that use is not recommended. <page> is intended for documents that only appear inside <iframe> tags.

10.5.3 <editor>

The <editor> tag is a specialist box-object tag like <iframe>. It displays an entire document as <iframe> does. The only special attribute for <editor> is src, which holds a URL for the displayed content. Some versions of Mozilla support an editortype="html" or "text" attribute. There is also the type="content" or "content-primary" attribute; the latter makes the window.content property refer to the editor's contentWindow. <editor> is an example of the XPCOM pair:

```
@mozilla.org/layout/xul-boxobject-editor;1 nsIEditorBoxObject
```

The nsIEditorBoxObject interface has the DocShell functionality present in the <iframe> tag, plus the very extensive functionality of the nsIEditor interface, hanging off an editor property. This second interface provides all the technology required to implement visual editing of an HTML document, like selection, insertion, and so on. The <editor> tag is the heart of the Classic Composer tool, just as the <iframe> tag is the heart of the Classic Browser tool.

The <editor> tag provides no controls for the displayed edit pane.

10.5.4 <browser>

The <browser> tag is a custom tag like <iframe>. It displays an entire document as <iframe> does and supports the src attribute as for <iframe>. <browser> is a boxlike tag and an example of the component

```
@mozilla.org/layout/xul-boxobject-browser;1
```

This component implements the nsIBrowserBoxObject interface, which is nearly identical to the nsIIFrameBoxObject interface. <browser> and <iframe> are identical at heart.

The <browser> tag differs from the <iframe> tag in its XBL definition. That definition is in browser.xml in toolkit.jar in the chrome. The XBL binding for <browser> is very extensive with many methods and properties. Implementing simple tasks with the <iframe> tag means digging through a few interfaces, finding the right properties, and making a few method calls. The <browser> tag provides a range of convenient methods that do this digging for you. The <browser> tag's XBL definition also adds and coordinates history, security, and drag-and-drop support.

The <browser> tag also adds caret browsing functionality. Caret browsing occurs if F7 is pressed: The user can then navigate a read-only HTML page by moving the insertion point around the page, as though it were an editor session.

<iframe> is more efficient than <browser> when the document displayed is static and never changes. <browser> is more convenient when the document in the pane might be replaced several times or when basic navigation is needed.

10.5.5 <tabbrowser>

Just as <browser> is an enhancement of <iframe>, <tabbrowser> is an enhancement of <browser>. Unlike the other two tags, <tabbrowser> has no XPCOM component identity; it is just one very large XBL definition in tabbrowser.xml in toolkit.jar in the chrome.

<tabbrowser> is a combination of <tabbox> and <browser> tags. Each <tabbox> tab is a <browser>, and tabs can be dynamically added. <tabbrowser> is ultimately used to implement the content display area of the Classic Browser window.

<tabbrowser> supports the following special attributes:

```
onnewtab contenttooltip contentcontextmenu
```

onnewtab is an event handler that fires when the user creates a new tab by pressing the New Tab button, or by using the menu system. contenttooltip and contentcontextmenu are stripped of the prefix content and passed to the <browser> tags inside the <tabbox> tabs.

Although it is possible to automate the actions of the <tabbrowser> through its many methods, there is little point in doing so because the tag is designed for user viewing of multiple read-only documents. The most an application programmer might do is examine the current state of the tab system.

10.5.6 <IFRAME>,<iframe> and <FRAME>,<frame>

These HTML or XHTML tags are used to place documents inside a frame in an HTML page. A XUL document can be displayed inside such a frame.

10.5.7 Non-Tags

When embedding documents, there are several do-nothing combinations.

By default, the <html> tag does nothing in a XUL document. It acts like any user-defined tag. It can be made more useful with an XML Namespace, as described in "Mixing Documents."

The <window> tag does nothing when embedded in other XUL content, so <window> tags do not nest, and neither do <dialog> or <page> tags. Outermost tags only nest when they are in separate documents and an <iframe> is between them.

The chromehidden attribute of the <window> tag does nothing for the application programmer. It is set by the window.open() method, or by toolbar logic in the Classic Browser, or by the toolbar toggle widget on MacOS X, on the Macintosh only. The windowtype attribute of the <window> tag takes a string and is used for window management (see "Managing Existing Windows"). It has no special meaning to <window>.

If there was ever a <package> tag, then it is gone now. It occasionally appears in older Mozilla documentation.

10.6 MIXING DOCUMENTS

It is possible to combine documents so that their individual tags are mixed together, without isolating one document in an <iframe>.

10.6.1 Mixing XML Document Types

A very powerful yet complex feature of Mozilla is the capbility to render a document that contains tags from several standards. This means that a document can contain HTML, SVG, MathML, and XUL content, all mixed together. This is done simply by adding XML namespaces. All tags from the added namespace are then available. Mozilla, however, will only recognize XUL content from a .xul file extension or from the correct XUL MIME type, so that much is mandatory if XUL is involved. This system was touched on in Chapter 7, Forms and Menus, where HTML and XUL forms were mixed.

There is a distinction between deeply mixing and lightly mixing types of content. The distinction is that layout is harder to perfect for deeply mixed content.

When different content is lightly mixed, a few large chunks of different content are put together. An example of light mixing is an HTML document that contains a few equations, each written in a chunk of MathML.

When different content is deeply mixed, individual tags from different content types sit side by side. This usually occurs when the document author naïvely assumes that any tag from any standard can be used anywhere.

Lightly mixing content generally works. Deeply mixing content doesn't always work. HTML and MathML is the least problematic deep mixture. Deeply mixing XUL and HTML or MathML requires care. Deeply mixing SVG with something else doesn't work because SVG content requires its own dedicated display area.

Mixing standards can be useful, but it is also something to be wary of. There are several reasons for caution:

- The layout models for different standards are not identical. A precise set of rules for laying out mixed content is very difficult to find. You can spend much time fiddling with styles.
- Mixed standards are not well tested. The number of tests required to conclude that two standards are fully compatible is huge. By the grace of a good internal design, mixing does work. If you press hard, though, you'll find effects that no one has thought of or tested yet.
- The renderable standards in Mozilla do not share the same document root. This means that the C/C++ object that implements the DOM 1 document interface is different for each standard. The set of features available depends on what the root document is. XUL embedded in HTML is not the same as HTML embedded in XUL. For heavily scripted documents, like XUL and DHTML, the right interfaces are very important.

It's very tempting to add an HTML `<A HREF=>` tag to XUL so that your splash or help screen includes a link back to the vendor's Web site. It's very tempting to add an HTML `<FORM>` tag and auto-submit forms from XUL. Both of these tasks, however, can be done without any HTML. It is better to keep your XUL clean.

The more recent DOM standards include interface methods like `createElementNS()` and `getAttributeNS()`. NS stands for Namespace. These methods let you specify the namespace of content items in a given merged document.

10.6.2 Mixing XUL Documents

The `<overlay>` tag is similar to a sophisticated version of C/C++'s `#include` directive. It allows several XUL documents to be merged seamlessly, using the `id` attribute. It is described in Chapter 12, Overlays and Chrome.

10.7 MANAGING EXISTING WINDOWS

After windows are created, the application may wish to manage them. A simple example is the File | Inspect a Window menu option of the DOM Inspector, which picks up a window for study. An even simpler example is the bottom half of the Window menu in most Mozilla applications, which moves the focus between windows.

Trying to juggle multiple windows can lead to very bad design. In most cases, the user can close a window at any time. This wipes out all the state of that window (except possibly for Java applets and shared XPCOM objects). It is difficult to maintain a rational design when state can disappear arbitrarily. Generally speaking, an application should have one master window, with any other required windows being dependent on that master. The dependent and modal features of `window.open()` are the simplest way to coordinate windows. If a desktop metaphor is required, then either all windows must be independent of each other, or a registration system must be implemented to track who is still available.

Mozilla provides several components and interfaces for managing windows. The simplest pair is

```
@mozilla.org/appshell/window-mediator;1 and nsIWindowMediator
```

This interface provides a list of the currently open windows. It is not finished (frozen) as of version 1.2.1 and may change slightly. The list has several forms, depending on which method is used. Listing 10.3 illustrates how to get a static list of currently open windows.

Listing 10.3 Retrieval of currently opened windows using `nsIWindowMediator`.

```
var C, obj, iface, mediator, enum, windows;
C     = Components;
obj   = C.classes["@mozilla.org/appshell/window-mediator"];
iface = C.interfaces.nsIWindowMediator;

mediator = obj.createInstance(iface);
enum     = mediator.getEnumerator(null);
windows  = [];

// record all windows using nsISimpleEnumerator methods
while ( enum.hasMoreElements() )
   windows.push(enum.getNext());

// do something with the first window
windows[0].document.GetElementById …
```

The `getEnumerator()` method can be passed a string that will filter the list of windows retrieved. Only windows with an `<html>`, `<window>`, `<dialog>`, or `<wizard>` tag whose `windowtype` attribute matches the string will be returned.

To return only XUL windows, use `getXULWindowEnumerator()` instead. The interface can also report stacking order for the currently open windows. Iconized (minimized) windows are at the bottom of the stacking order.

Windows open and close all the time, and an application might want to track this dynamically, as the menus noted earlier do. One way to do this is to add a listener to the window mediator object. This listener (an object implementing `nsIWindowMediatorListener`) is advised if a window opens, closes, or has its title changed. A more advanced solution, which requires little code, is to use the `rdf:window-mediator` data source directly in XUL. RDF and data sources are described in Chapter 11, RDF.

10.8 STYLE OPTIONS

The windowing aspects of XUL and HTML benefit from both trivial and systematic Mozilla enhancements to the CSS2 style system.

The trivial enhancements are as follows.

The `<tooltip>` tag displays `-moz-popup`, as for the `<menupopup>` tag discussed in "Style Options" in Chapter 7, Forms and Menus. It is less useful to display tooltips inline than menus, although that does provide a quick eye-check that all visual elements have such tips.

The `-moz-appearance` style property, used to support native (desktop) themes, also supports the following values:

```
window dialog tooltip caret
```

Systematic enhancements also exist for native theme support. It is useful for windows, and for dialog windows in particular, to appear exactly like windows created by the native desktop (e.g., Windows or GTK). Dialog boxes are disruptive enough for the user, without including bizarre colors and shapes. Mozilla includes color name and font name extensions designed to help dialog boxes (and other content) mimic the dialog boxes that the desktop would produce. These color and font names can be used anywhere in a style rule that a normal color or font name would appear.

In addition to these custom colors and fonts, Mozilla supports the desktop-oriented colors and fonts specified in the CSS2 standard, section 18.2.

Extensive and careful use of these styles in a XUL application can remove every hint that the application is Mozilla based. Well-styled applications designed this way may appear to be no different than traditional desktop applications, such as Visual Basic Applications on Windows, or GTK-based applications on UNIX.

Additional native-matching color names supported by Mozilla are noted in Table 10.2. Recent updates to these colors can be found by examining the source code file `nsCSSProps.cpp`.

Table 10.2 Mozilla CSS2 color name extensions

Native color name	Matches this native item	Additional Macintosh-specific color names
-moz-field	Background of a form field	-moz-mac-focusring
-moz-fieldtext	Foreground of text in a field	-moz-mac-menuselect -moz-mac-menushadow -moz-mac-menutextselect
-moz-dialog	Background of a dialog box	-moz-mac-accentlightesthighlight
-moz-dialogtext	Foreground of text in a dialog	-moz-mac-accentregularhighlight -moz-mac-accentface -moz-mac-accentlightshadow
-moz-dragtargetzone	Highlighted color of a drag target when dragged over	-moz-mac-accentregularshadow -moz-mac-accentdarkshadow
-moz-hyperlinktext	Clickable link text, such as Windows Active Desktop links	-moz-mac-accentdarkestshadow
-moz-visitedhyperlinktext	Clickable link text for a visited link	

Similarly, Mozilla supports the font name extensions listed in Table 10.3, but these font names don't appear to do anything as of version 1.4.

The special font name `-moz-fixed` does provide a real font and can be used to ensure that a fixed-width font is displayed. It has the special property

Table 10.3 Mozilla CSS2 font name extensions

Native font name
-moz-window
-moz-document
-moz-workspace
-moz-desktop
-moz-info
-moz-dialog
-moz-button
-moz-pull-down-menu
-moz-list
-moz-field

that it can be rendered at all point sizes, so text in that font is guaranteed to appear regardless of the value of the `font-size` CSS2 property.

10.9 HANDS ON: NOTETAKER DIALOGS

In this "Hands On" session, we'll take advantage of the windowing aspects of XUL to clean up the NoteTaker Edit dialog box some more. We'll also coordinate the application window and the Edit dialog window a little so that they work together. These two items consist of a number of tiny jobs:

1. Replace `<window>` with `<dialog>` in the Edit dialog box.
2. Replace plain `<button>` handlers with `<dialog>` button handlers.
3. Implement the `notetaker-open-dialog` command on the main browser window so that `window.openDialog()` is used to display the Edit dialog box.
4. Implement the `notetaker-close-dialog` command.
5. Find and implement a strategy for handling data that is used by more than one window.
6. Work out what kind of window a NoteTaker note is.

First, we look at the Edit dialog box. Replacing the `<window>` tag is trivial. The new dialog tag could do with a title as well. That new tag will be

```
<dialog xmlns="http://www.mozilla.org/keymaster/gatekeeper/
     there.is.only.xul"
    id="notetaker.dialog"
    title="Edit NoteTaker Note"
    onload="execute('notetaker-load')">
```

Unfortunately, we gain a little but also lose a little on the handler side. We don't need to hand-create `<button>` tags any more because `<dialog>` supplies them, but at the same time we can't easily use `<command>` tags with `<dialog>` because there's more than one possible command attached to that tag. If we desire, we could suppress the buttons that `<dialog>` shows us and keep our existing buttons. Instead, we'll do it the way `<dialog>` suggests, which is to use that tag's own buttons. This gives us standard buttons on each platform. We also might add these handlers:

```
ondialogaccept="execute('notetaker-save');execute('notetaker-close-
     dialog');"
ondialogcancel="execute('notetaker-close-dialog');"
```

In fact, the `notetaker-close-dialog` command is not needed in some cases because `<dialog>` will automatically close the window when Cancel is pressed, or when the Close Window icon on the title bar is pressed. We might

as well get rid of it from the code since <dialog> does everything for us. We can use an onclose handler at a later time if necessary. So only the ondialogaccept handler needs to be added.

That leaves the -open- and -close- commands to implement. We don't yet have a fully integrated toolbar, but we can use the fragment of XUL that we do have to test the command's implementation. The action() function used by the toolbar needs a very simple enhancement:

```
if ( task == "notetaker-open-dialog" )
{
  window.openDialog("editDialog.xul","_blank","modal=yes");
}
```

We use modal because the Edit dialog box isn't a full manager window like many of Mozilla's windows. We want the main browser window to freeze while the user is working on the dialog box. In that way, we don't need to worry about focus jumping back and forth between the two windows, or worse, between form fields on different windows.

Similarly, the action() function for the Edit dialog box requires a trivial enhancement:

```
if (task == "notetaker-close-dialog")
  window.close();
```

Penultimately, there is the matter of managing data. The NoteTaker tool is designed to maintain one note at most per URL, and the user is expected to work on one note at a time. But the tool spans two windows so far. Both the toolbar and the dialog box contain form fields that the user can enter information into. Which window holds the temporary state of the current note?

The answer depends on how that state is stored. In later chapters, the information will be stored in RDF, which is independent of a given XUL or HTML window. In this chapter, we'll store the data as a simple JavaScript object. Such an object originates in a single window. We choose the browser window to store the state because it's the window that displays the Web page to which the note is attached.

```
var note = {
  url : null;
  summary : "",
  details : "",
  chop_query : true, home_page : false,
  width : 100, height : 90, top : 80, left : 70
}
```

This object is automatically available to JavaScript in the Edit dialog code using the simple syntax:

```
window.opener.note
```

Each browser window will have one such object. This object can be further enhanced at any point with methods that perform updates to or from the form fields on the toolbar or in the dialog box. In this way, processing is centralized. If the dialog box were more complicated, it might have its own state and its own set of objects, but that is unnecessary in this case.

If we use the `opener` object, we must be very careful. Even though the data are being put into a different window, the current JavaScript context resides in the window that the script started in. Calls to `setTimeout()`, `setAttribute()`, or other trickery will always run against the current window, not the window being manipulated, even if the call is made through a function that is defined in that other window. Listing 10.4 shows logic for the dialog box's `action()` function, which is implemented in the dialog box window.

Listing 10.4 Save and load of NoteTaker dialog box data to the main window.

```
if (task == "notetaker-save")
{
  var field, widget, note = window.opener.note;

  for (field in note)
  {
    widget = document.getElementById("dialog." + field.replace(/_/,"-");

    if (!widget) continue;

  if (widget.tagName == "checkbox")
    note[field] = widget.checked;
  else
    note[field] = widget.value;
  }
}

if (task == "notetaker-load" )
{
  var field, widget, note = window.opener.note;

  for (field in note)
  {
    widget = document.getElementById("dialog." + field.replace(/_/,"-");

    if (!widget) continue;

    if (widget.tagName == "checkbox")
      widget.checked = note[field];
    else
      widget.value = note[field];
  }
}
```

These two routines are the inverse of each other. The `continue` statements let the dialog box ignore properties of the note object that the dialog box doesn't know about. With a bit more organization, we could make the object property names and the form field name the same (even though "-" is not a valid character for literal property names), which would slightly shorten the code, but we haven't bothered. The Edit dialog box now "saves" and "loads" its information back to the main window, so the toolbar logic must be expanded (later) to do a real save from the running platform to the external world. The note object is now our official record of the current note.

Last of all is the question of the NoteTaker note itself. The purpose of the note is to annotate a Web page with a comment from the viewer, so the note must appear on top of the Web page somehow. The note's data will be stored locally, and its display will be generated locally, but the Web page may come from any Web site. Because NoteTaker is installed in the chrome, and is therefore trusted, it has permission to obscure or alter any displayed Web page without restriction, including covering part of the site with note content.

One implementation strategy for the note is to use pure XUL, XBL, and JavaScript. A Web page in a browser window is displayed inside an `<iframe>` that is part of a `<tabbox>` that is part of a `<tabbrowser>`. If the `<iframe>` were wrapped in a `<stack>`, then the second card of the `<stack>` could be the Web page, and the first card of the stack could be the note. That note could be positioned using relative styles, and the Web page would "show through" everywhere except where the note was located. The note could then be any simple XUL content, like a simple `<box>` with some borders, background, and content. Think of a message written on a window pane—the garden can still be seen even though part of the glass has writing on it.

This strategy would require changes to the `<tabbox>` tag, which is defined in XBL. We can do that, but replacing the standard `<tabbox>` tag is a big move because it requires much testing. We would need to integration test all the changes we make with every application installed on the platform. That includes the browser itself. We'd rather not do that much work.

An alternate strategy is to implement the note in HTML and CSS. From the chrome-installed NoteTaker, we could reach down into the displayed Web page. Using DHTML techniques, we could add a `<span>` tag and its content. That `<span>` tag would be styled to be absolutely positioned and to have a high `z-index` so that it's always on top. There's a one-in-one-billion chance that this will clash with existing content on the page, but that's small enough for us to live with. This strategy has the benefit that it doesn't affect the rest of the chrome. This is the strategy we'll use.

A NoteTaker note will appear as shown in Figure 10.3. That figure is an ordinary Web page.

We must use the lowest-common denominator HTML because we don't know what standard the Web page will be written to. That means valid XML and valid HTML, just in case the page is displayed in strict mode. We are free, however, to use any of Mozilla's HTML enhancements because we know

Fig. 10.3 Sample NoteTaker note constructed from HTML.

the browser will always be Mozilla. Compared with normal browser compatibility issues, that's a little unusual. HTML for the previous note is given in Listing 10.5.

Listing 10.5 HTML construction of a NoteTaker note.

```
<span id="notetaker-note">
  <span id="notetaker-note-summary">
    Note Summary
  </span>
  <span id="notetaker-note-details">
    All the details go here
  </span>
</span>
```

In fact, we should go to some extra trouble to make sure that the `xmlns` prefix for the tags is always correct, but we won't bother here. This HTML code has styles shown in Listing 10.6.

Listing 10.6 CSS style construction of a NoteTaker note.

```
#notetaker-note {
    display : block;
    position : absolute;
    z-index : 2147483646; /* one less than menu popups */
    overflow : auto;

    background-color : lightyellow;
    border : solid;
    border-color : yellow;
```

```
    width  : 100px;
    height : 90px;
    top    : 80px;
    left   : 70px;
}
#notetaker-note-summary {
    display : block;
    font-weight: bold;
}
#notetaker-note-details {
    display : block;
    margin-top : 5px;
}
```

Sometime in the future, this Web note technology could get an upgrade. Each note could be draggable, resizable, and iconizable, using the mouse. Each one could be directly editable. Most of these additions are standard DHTML tricks and aren't pursued here. Such changes would also need to be detected by the NoteTaker tool, which is not hard either.

To get a basic note in place, we can capture these two pieces of HTML and CSS code in JavaScript strings. Instead of hard-coded values, we can use placeholders so that

```
"… width : 100px …"
```

is stored as

```
"… width : {width}px …"
```

Using JavaScript regular expressions, we can substitute the values entered by the user in the toolbar or Edit dialog box into this string. After we have a finished string, we can create the note as shown in Listing 10.7.

Listing 10.7 Dynamic HTML creation of NoteTaker note.

```
function display_note()
{
  var style = generate_style();
  var html = generate_html();
  var doc = window.content.document;

  var stree = doc.getElementById("notetaker-styles");
  if ( !stree )  // add the topmost <style>
  {
     stree = doc.createElement("style");
     stree.setAttribute("id","notetaker-styles");
     var head = doc.getElementsByTagName('head').item(0);
     head.appendChild(stree);
  }
  stree.innerHTML = style;

  var htree = doc.getElementById("notetaker-note");
```

```
  if ( !htree )  // add the topmost <span>
  {
    htree = doc.createElement("span");
    htree.setAttribute("id","notetaker-note");
    var body = doc.getElementsByTagName('body').item(0);
    body.appendChild(htree);
  }
  htree.innerHTML = html;
}
```

The code uses `getElementByTagName()` to locate the `<head>` and `<body>` tags in the HTML page—the `id` for those tags in unknown by us. It then creates the topmost tag for the styles or the content and appends it to the `<head>` or `<body>` tag's existing content. Mozilla's special `innerHTML` property inserts the rest of the content from a string. For this simple system, we assume that the displayed page is not a frameset, and that it contains a `<head>` and a `<body>` tag. These assumptions can be lifted, but the result is just more DHTML code, which doesn't teach us much about Mozilla. The `generate_html()` function looks like Listing 10.8 and is trivial; the `generate_style()` function is analogous.

Listing 10.8 Insertion of NoteTaker data into Dynamic HTML content.

```
function generate_html()
{
  var source =
   '<span id="notetaker-note-summary">{summary}</span>' +
   '<span id="notetaker-note-details">{details}</span>';

  source = source.replace(/\{summary\}/, note.summary);
  source = source.replace(/\{details\}/, note.details);

  return source;
}
```

These changes are not so easy to test because they require integration with the Web browser. A complete testing solution is to read about overlays, which are two chapters in the future. A temporary solution is to hack the browser code, which we'll do here.

To hack the browser, make a copy of `navigator.xul` and put it in your `notetaker/content` directory in the chrome. The original is in `comm.jar` in the chrome. If we start Mozilla using this file:

```
mozilla -chrome chrome://notetaker/content/navigator.xul
```

then, voilà, a perfectly normal browser window appears. We'll modify *the copy* of this file. First, we add `<script src=>` tags for all the scripts needed for the toolbar code. Second, we find this line:

```
<toolbar id="nav-bar" …
```

This is the main navigation toolbar. Immediately after that opening tag, we add a <toolbarbutton> like so:

```
<toolbarbutton label="Test" onclick="display_note()"/>
```

When we save and load this file, a test button appears on the navigation bar. Pressing it makes a note appear—provided an HTML page is loaded. We can install any onclick handler we want on this button, including calls to execute(), action(), and anything else. In fact, we could put the whole NoteTaker <toolbar> content into this file (temporarily) if we wanted to.

This testing requires that the tester wait for a given HTML page to load before pressing the Test button. In the final, automated NoteTaker tool, we won't have that luxury. Instead, we'll need to detect the loading page. An unsophisticated solution is simply to poll the contents regularly to see if anything arrived.

```
function content_poll()
{
  if ( !window.content ) return;
  var doc = window.content.document;
  if ( !doc ) return;
  if ( doc.getElementsByTagName("body").length == 0 ) return;
  if ( doc.location.href == "about:blank" ) return;
  if ( doc.visited ) return;

  display_note();
  doc.visited = true;
}
setInterval("content_poll()", 1000);
```

This code examines the currently displayed HTML page to see if a note needs to be displayed. If there's no document, or the document doesn't have a body yet, or it's blank, or it already has a note, do nothing. Otherwise, find the note and add it.

To summarize this "Hands On" session, we now have the windows that make up the display portion of the NoteTaker tool. We have a memory-resident version of the current note in the form of a JavaScript object. We have some coordination between windows and some logic tying the note object to the displayed note. We even have a way to store and load the note on a Web server. With a little more work tying the user input to the JavaScript note object, this tool could be finished. The note even reloads when the Web page reloads.

The major thing missing is proper treatment of the Web page's URL. Each Web page is supposed to support a unique note. With the current arrangement, we need to submit a form request to a server to get the note back—that's rather inefficient. Our only alternative so far, and that is merely hinted at, is to write all the notes to a flat file. There's a better way, and that way is to store the notes as RDF. We discuss that option in the second half of this book.

10.10 DEBUG CORNER: DIAGNOSTIC WINDOWS

Before exploring window diagnostics, let's consider a cautionary remark about Microsoft Windows. When you are in the middle of constructing a still-buggy XUL application on Windows, the Mozilla Platform can occasionally become confused. The result of this confusion is that the platform remains in memory even when the last Mozilla window is shut down. When subsequent windows are started, they attach to the existing, confused instance of the platform. The most obvious symptom of this problem is that, no matter how hard you try, your changes to source files never appear in newly displayed windows.

To detect this problem, note that the splash screen only appears when the platform is first started: no splash screen + no existing windows = confused platform. To confirm the problem, use Control-Alt-Delete to review the list of running processes, and do "End Task" on any Mozilla processes.

Fortunately, this problem is less frequently seen as the platform matures. Being mindful of it can save hours of fruitless testing, though. There are many sources of defects, and this behavior is only one. For the rest, you must look to your own code.

When analyzing a complex application, nothing beats real-time information about the state of the application, delivered neatly in a controllable window. Mozilla provides a number of alternate ways to achieve this.

The simplest method is to use the `dump()` method described in Chapter 5, Scripting. Its output appears in the Mozilla window that appears when the platform is started with the `-console` option.

Nearly as simple to use are the many windows supplied by the JavaScript Debugger. To enable the debugger, first open a debugger window by hand. Turn on all the subwindows listed under View | Show/Hide as an experiment. At the first scripting opportunity in the XUL or HTML page to be diagnosed, add this line:

```
debugger;
```

When this line of script is executed, control will be thrown to the debugger, and you can step through the page's scripts from that point on. Use the big buttons in the debugger window and examine the content of each of the small subwindows as you go.

In HTML, the `document` object has `open()`, `close()`, and `write()` methods and a progressive rendering system that displays content before the document is completely loaded. This system can be used as a logging system. An HTML window can be opened from another window using `window.open()` and diagnostic content logged to it with `document.write()` as required.

XUL does not support HTML-style incremental display, but a similar system is still possible. Load a simple, empty XUL document into the window destined for logging, using `window.open()`. That document should have no

content other than <window orient="vertical">. Use a sequence of statements as in Listing 10.9 to insert new content into that page:

Listing 10.9 Insertion of diagnostic messages into a new XUL window.

```
var win = window.open( …new window …);

function logger(message)
{
 var obj = win.document.createElement("description");
 var txt = win.document.CreateTextNode(message);
 obj.appendChild(txt);
 win.document.appendChild(obj);
}
```

Note that the document elements are created using the document in the new page, not the document in the existing page. Such a system can be enhanced with scrollbars and other controls. If the application is secure (e.g., installed in the chrome) it is just as easy to use the JavaScript Console. This is shown in Listing 10.10.

Listing 10.10 Logging messages to the JavaScript Console.

```
// Find the console
var C     = Components;
var obj   = C.classes["@mozilla.org/consoleservice;1"];
var iface = C.interfaces.nsIConsoleService;
var cons  = obj.getService(iface);

// Log a message
cons.logStringMessage("test message");
```

A similar system can be used for logging to the Java Console. This console has the advantage that it is not a XUL window and doesn't require a debug build of the browser or command-line display (which dump() needs). Messages can be logged to it without disturbing the normal state of any XUL windows. The Java Console does not appear in any list of windows retrieved using the earlier window mediator code either. This old-fashioned but familiar line from Netscape 4.x days can be used to write directly to the Java Console (it is a static class method):

```
window.java.lang.System.out.println("my message");
```

The console itself can also be exposed from code. The process is simple and shown in Listing 10.11, but it requires a secure application.

Listing 10.11 Revealing the Java Console window from a script.

```
var C = Components;
var obj   = C.classes["@mozilla.org/oji/jvm-mgr;1"];
var iface = C.interfaces.nsIJVMManager;
```

```
var cons  = obj.getService(iface);

if (cons.JavaEnabled) cons.showJavaConsole();
```

Before exposing the window, the whole Java subsystem must be checked to ensure that it is not disabled by the user.

10.11 SUMMARY

Managing windows in Mozilla is heavily inspired by technology used in traditional Web browsers. Much can be achieved with the `window.open()` method and a few highly customized XUL tags. These tags are only of interest to application developers whose applications are browser-like.

Mozilla's XPCOM architecture reveals many interesting objects used to manage documents retrieved by the platform, not the least of which is the idea of a DocShell. These objects are saturated with functionality needed for ambitious content-oriented applications.

Mozilla's highly customizable windows also benefit from styles. Using style information, application programmers can integrate a window with the standard desktop appearance, which prevents the application from appearing foreign.

Many of the XUL tags discussed so far in this book are content-like and static. Software applications, however, are often data-centric and dynamic rather than content-centric and static. Mozilla caters to these nonbrowser needs with novel support for data-oriented processing tasks. That support is the topic of the next few chapters, starting with the RDF language.