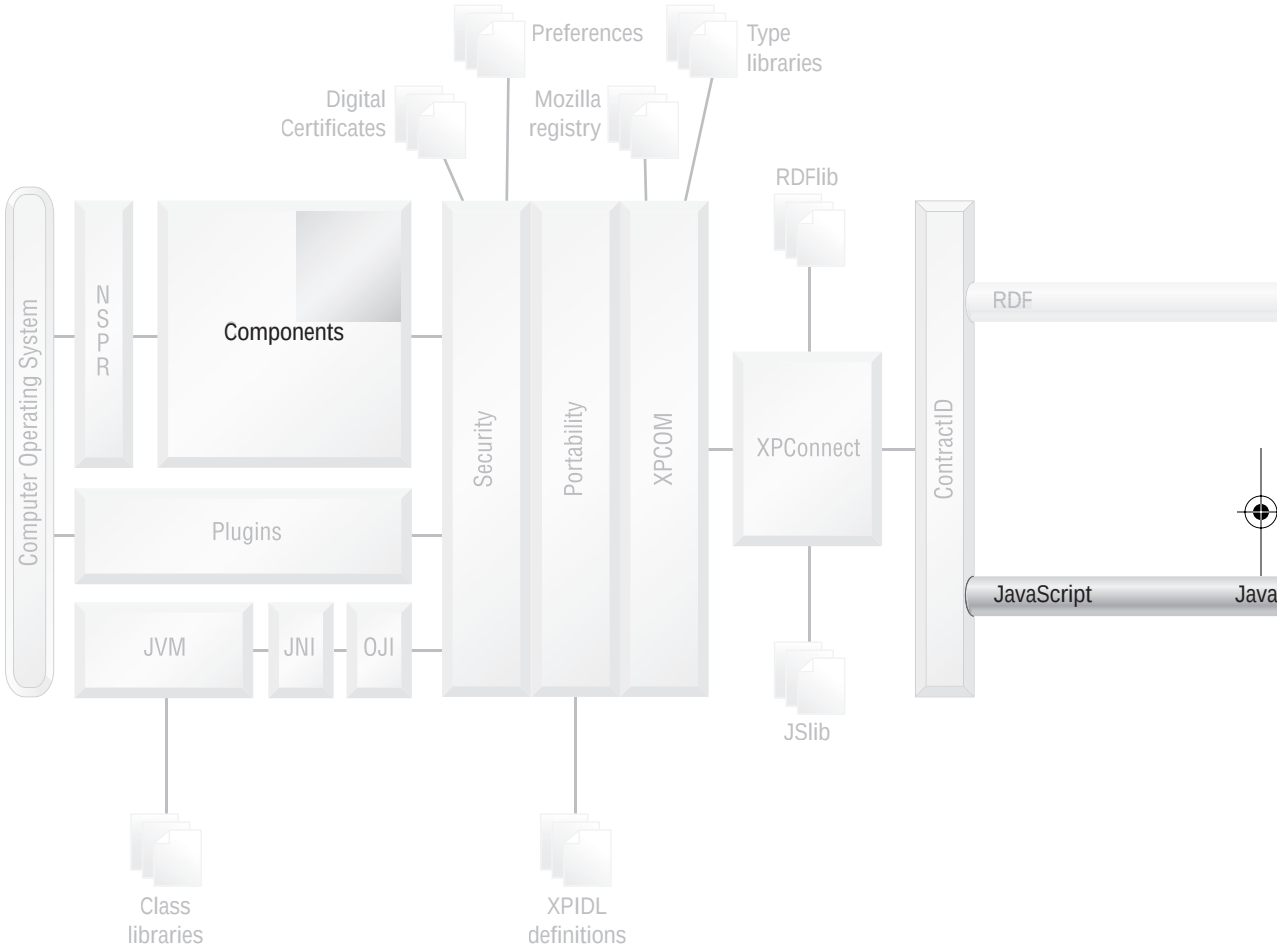
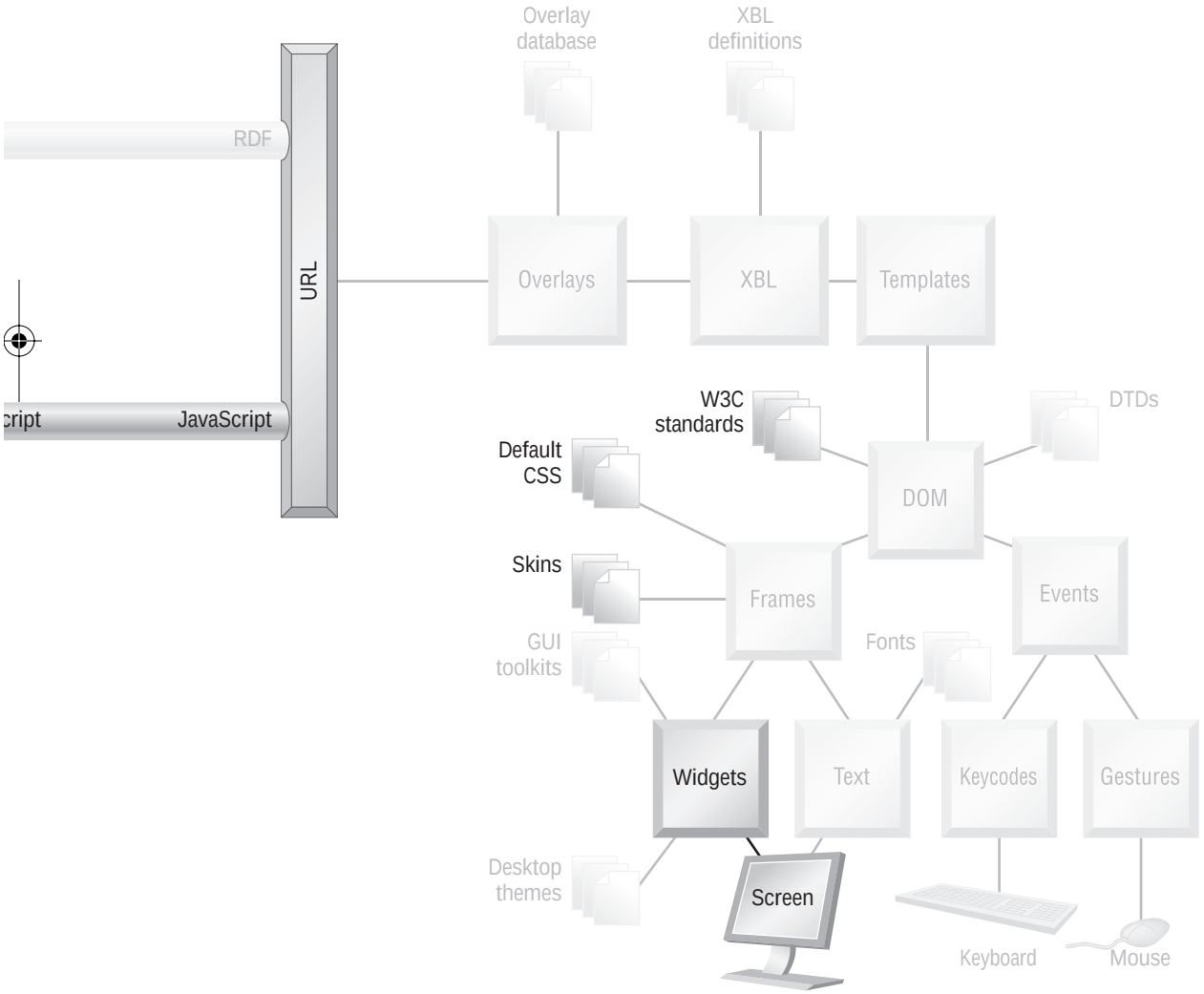

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
Plugins
JavaScript
Java
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 7

Forms and Menus



This chapter describes most of the XUL tags used for data entry. It also explains how to submit forms over the Web. Widgets and tags more concerned with user movement are covered in Chapter 8, Navigation.

The best way to assist users with their input is with guidance and feedback. For paper-based systems, this means using a particular style of layout called a form. GUI toolkits provide widgets that are electronic versions of paper forms. In Chapter 4, First Widgets and Themes, buttons were considered at length. This chapter describes the other basic form controls that go with buttons: menus, check boxes, text boxes, and so on. Each of these controls, plus the various menu controls, expresses either a uniquely new widget or a unique combination of widgets.

The NPA diagram that precedes this chapter shows where in Mozilla form and menu technology sits. Unsurprisingly, XUL form and menu tags are heavy users of the desktop's GUI toolkit. Each widget must contribute something new and unique to XUL, and that new functionality is best found in the features the desktop provides.

Forms and menus have the same constraint as the `<button>` tag: They must look like forms and menus to be recognized. Style information, therefore, plays an important part in these widgets. The NPA diagram also notes the importance of XBL definitions (bindings). Form and menu tags are usually manipulated extensively by scripts, and the programmer needs to have XBL definitions for those tags at hand in order to remember what properties and methods are available.

Finally, the NPA diagram shows that some XPCOM components are relevant to the forms environment. Submitting a form to a Web server is a classic use of the Mozilla technology, and the first example of using the platform as client software to some server.

HTML's form tags have been wildly successful. XUL forms are similar to HTML, so we begin with a brief comparison of the two. After that, we will look at all the XUL tags involved.

7.1 XUL AND HTML FORMS COMPARED

HTML/XHTML has a `<FORM>` tag. This tag collects `<BUTTON>`, `<SELECT>`, `<INPUT>`, and `<TEXTAREA>` tags into a group. The `<FORM>` tag's DOM object has methods that combine the values of the form members together into an HTTP GET or POST request. This automates the process of permanently capturing form data by sending it to some Web server. As a result, Web-based HTML-based applications are now common.

XUL has no equivalent to the `<FORM>` tag. XUL form control equivalents are not bound into groups (except via `<radiogroup>`). There is no equivalent to `<INPUT TYPE="reset">` or `<INPUT TYPE="submit">` in XUL, and XUL has no semiautomatic form submission process. If a plain XUL document is presented to the user, then nothing happens to the data the user enters.

This means that if a XUL application is to behave like an HTML form, then form submission must be added by hand. This is done with scripts, which are added to each XUL document that is to be formlike. Fortunately, such scripts are trivial to write.

On the other hand, XUL form controls are more varied than HTML ones. The `<button>`, `<textbox>`, and `<menu>` tags are all more flexible than the HTML equivalents. In addition, XUL has many high-end tags, such as `<menubar>`, `<listbox>`, and `<tree>`, that are far more sophisticated than anything HTML has to offer. These better-than-forms tags are discussed in later chapters.

Table 7.1 lists the tags in XUL that are closest to HTML's form and menu tags.

Table 7.1 HTML and XUL form and menu tags compared

HTML tag	XUL tag	Notes
<FORM>	none	Use XMLHttpRequest object.
<BUTTON>	<button>	
<INPUT TYPE="button">	<button>	
<INPUT TYPE="text">	<textbox>	
<INPUT TYPE="radio">	<radio>	Or use <button type="radio">.
<INPUT TYPE="checkbox">	<checkbox>	Or use <button type="checkbox">.
<INPUT TYPE="password">	<textbox type="password">	
<INPUT TYPE="submit">	none	Use <button> and a script.
<INPUT TYPE="reset">	none	Use <button> and a script.
<INPUT TYPE="file">	none	Use the FilePicker object.
<INPUT TYPE="hidden">	none	Use a plain JavaScript variable.
<INPUT TYPE="image">	<button>	Use a script for any form submit.
<SELECT>	<menulist> or <listbox>	
<OPTGROUP>	none	Use <menuseparator> instead.
<OPTION>	<menuitem>	
<TEXTAREA>	<textbox multiline="true">	Supports rows and cols attributes.

Table 7.1 HTML and XUL form and menu tags compared (Continued)

HTML tag	XUL tag	Notes
<LABEL>	<label>	See discussion on accessibility.
<FIELDSET>	<groupbox>	
<LEGEND>	<caption>	
<INPUT TYPE="radio" NAME=>	<radiogroup>	

7.2 Where to Find Information on Menus

Forms and menus tend to appear together in GUI-based applications. XUL's support for menus is nearly as varied as its support for buttons. Unlike the overview of buttons in Chapter 4, First Widgets and Themes, menus are described in several different places in this book. There is a brief comparison of all menu types in "Menu Variations" in this chapter. Otherwise, look here:

- This chapter describes drop-down menus used as form controls. These controls are based on the `<menulist>` tag. `<menulist>` is built from `<menupopup>`, `<menuitem>`, and more, and those tags are discussed here, too.
- Buttons that are menulike are discussed in Chapter 4, First Widgets and Themes.
- Menus that do not drop down appear flat inside a document as a multiline box. They are called listboxes in HTML and XUL. XUL's `<listbox>` is described in Chapter 13, Listboxes and Trees.
- Menus that appear in menu bars using the `<menu>` tag are described in Chapter 8, Navigation. `<menu>` used as a submenu is discussed here.
- Context menus based on the `<menupopup>` tag are described in Chapter 10, Windows and Panes.

The only real distinction between menus in XUL is between `<listbox>` and `<menulist>`. `<listbox>` is the far more sophisticated tag of the two. All the other tags and uses noted are variations on `<menulist>`. All popup menus are implemented using the `<menupopup>` tag.

7.3 Forms

The form, in which a number of user-modifiable items are collected into a structured group, is central to both HTML and XUL. HTML is hypertext, and the hypertext concept doesn't really include the idea of a form, but forms are so useful that their addition to HTML is just a matter of history. XUL, on the

other hand, is meant for forms from the beginning. If XUL achieves widespread acceptance, then the forms module of future XHTML standards might be no more than a reference to part of XUL.

Mozilla's HTML and XUL forms are quite similar. The simplest of the form elements are nearly identical. Event handlers and navigation work much the same in both. Any HTML primer that contains a little JavaScript is good preparation for XUL forms. Don't, however, expect every fine detail to be identical. The two form systems are implemented separately, although they share some common code.

The simplest of XUL's form tags are discussed here. This set of simple tags is

```
<button> <checkbox> <radio> <radiogroup> <textbox> <label>
```

7.3.1 Form Concepts

One form widget is called a form control (from Microsoft terminology) or a form element (originally from graphic design, adopted by Netscape and the W3C). Such elements might interact with the user in different ways, but they are united by some common design.

7.3.1.1 Navigation If the user is to enter anything into a form, then interacting with one form widget at a time is a simple approach. Both XUL and HTML allow one widget to be selected at a time. Interactive form widgets are also ordered within a document. In W3C terms, this is called the *navigation order*. In Mozilla, the ordered collection of these widgets is called the *focus ring* because stepping one beyond the last widget leads back to the first one. All form elements are members of the focus ring.

The focus ring is discussed in more detail in Chapter 8, Navigation.

7.3.1.2 Common Properties of XUL Form Elements In the process of developing the CSS3 standard, the W3C produced a draft document, "User Interface for CSS3." Although old, it is still available at *www.w3.org/TR/1999/WD-css3-userint-19990916*. This document is an early attempt at some of the newer features that CSS3 hopes to support. It is at least ironic and coincidental that this part of CSS3 is (1) the part most similar to Microsoft's .NET and (2) one of the slowest parts of CSS3 to be finalized.

Mozilla implements many features of this draft document. In particular, it implements four style properties that represent the interactive potential of the simple XUL form elements. In Mozilla, the `nsINSDOMCSS2Properties` interface implements these styles. They appear as `-moz` styles in the stylesheet system. The four style properties are

```
user-input user-modify user-select user-focus
```

These style properties are somewhat independent of each other and are important because they make understanding the user input system easier.

Such an understanding has been hard to come by in the past because the results of applying event handlers like `onclick` and DOM methods like `focus()` tend to depend on many of these four states at once. Now that these states are identified, it is easier to understand what effect form element handlers have. You can take a set of four states and an event and write down new states that will result if a form element with those states gets that event.

Beyond these four styles, form elements share two other concepts. The first such concept is private state. All simple form elements are stateful and share at least the disabled attribute, which can be set to `"true"`. Figure 7.1 illustrates disabled and nondisabled form controls.

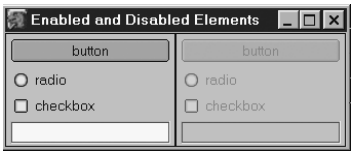


Fig. 7.1 Enabled and disabled simple form controls.

Finally, all simple form elements share a bundle of event handlers and matching object methods for interactive events like focus and blur.

7.3.1.3 Accessibility Accessibility is a feature of software designed to make it usable for those with disabilities. HTML hypertext content, links, and XUL application windows can be made accessible. Form elements are of particular interest because governments want to provide services over the Internet that supply equity of access to disabled citizens and other minorities.

In all discussion to date, the XUL `<label>` tag has appeared to be identical to the `<description>` tag, except that it can be reduced to a label attribute. In the area of accessibility, the `<label>` tag first differs from the `<description>` tag. The `<label>` tag can provide the alternate content that is needed for an accessibility system.

If a form element has a label attribute, Mozilla will present its content as the guide information that the accessibility system expresses to the user. If the form element doesn't have such an attribute, then Mozilla will look for a child tag that is a `<label>` tag and will use that tag. If no suitable child tag exists, it will look for a `<label>` tag whose id matches the id stated in the form element tag's control attribute. If that isn't found, then there is no accessibility information to supply.

Mozilla has accessibility support for all the simple XUL form elements. The `<menuitem>`, `<menulist>`, and `<tab>` tags also have accessibility support.

How accessibility works is discussed in more detail in Chapter 8, Navigation.

7.3.2 Simple XUL Form Tags

Figure 7.2 shows the simple form tags with additional styles applied so that their structure is clearer. Dashed lines indicate <label> tags, thick solid lines indicate <box> tags, and thin solid lines indicate <image> tags. The middle two sections are <radiogroup> tags, of which the second one contains <button type="radio"> tags.

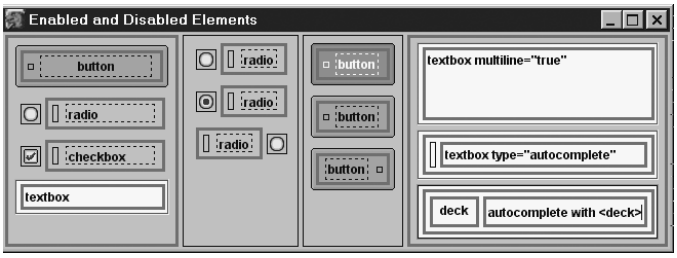


Fig. 7.2 Simple XUL form elements with internals exposed.

It's easy to see from Figure 7.2 that all simple form tags contain at least one other tag, even if that other tag is a contentless <image>. All these form tags are defined in XBL.

Inside Mozilla there are special event handlers written in C/C++. These are installed on the XUL document object with the C/C++ version of addEventListener(). These event handlers capture and process some of the event information associated with these simple controls. This is how the non-XBL form element functionality is implemented. There is no simple way to interact with these embedded handlers from JavaScript and no need to do so.

These tags have DOM interfaces that closely match the equivalent HTML interfaces. The XPCOM interface names all start with nsIDOMXUL prefixes.

7.3.2.1 <button> The <button> tag is discussed extensively in Chapter 4, First Widgets and Themes. Recall that the type attribute allows it to act like a checkbox, radio button, or menu start point if required.

7.3.2.2 <checkbox> The <checkbox> tag is made out of plain XUL content tags. It has an XBL definition. The checkbox graphics that appear when this widget is displayed are formed from ordinary images. They can be styled larger or smaller, or they can be replaced. <checkbox> has the following custom attributes:

```
src label crop checked disabled accesskey
```

src is passed to an optional <image> that appears between the checkbox and the text. crop is passed to a <label> containing the text. checked is the

boolean state of the <checkbox>, and disabled grays out the checkbox completely if it is set to true. The checkbox label can appear on the left of the checkbox if dir="rtl" is specified. Checkbox states are independent of other checkboxes. If the text (the <label>) of a checkbox is mouse-clicked, that is the same as clicking the Checkbox icon itself.

Using the command system described in Chapter 9, Commands, the <checkbox> tag sends a CheckboxStateChange event when its state changes. This event can be observed by application code if that is desired.

7.3.2.3 <radio> The <radio> tag is made out of plain XUL tags. It has an XBL definition. The radio button icons are formed from ordinary images and can be styled larger, smaller, or differently. The <radio> tag has the following custom attributes:

```
src label crop selected disabled accesskey
```

These attributes are the same as for <checkbox> except selected substitutes for checked. If the <radio> tag is in a <radiogroup>, only one such tag may have selected set. The bottom <radio> tag in the first <radiogroup> in Figure 7.2 has dir="rtl" set. Both the <label> text of a radio button and the button itself may be clicked.

Until very recent Mozilla versions, the <radio> tag occasionally became confused about when it should have focus. Look for recent reports in Mozilla's bug database to clarify the status of <radio> in your version.

7.3.2.4 <radiogroup> The <radiogroup> tag binds a set of <radio> or <button type="radio"> tags together into one unit. If any radio tag in the unit is selected, the others in the unit are deselected. The <radiogroup> tag supports the following attributes:

```
disabled selectedItem focusedItem selectedIndex
```

disabled grays out the whole radio group. selectedItem and focusedItem report the radio item in the group that is currently selected or focused. selectedIndex reports the number of the radio item in the group that is currently selected, starting with 0.

Using the command system described in Chapter 9, Commands, the <radio> tag sends a RadioStateChange event and a selected event when its state changes. These events can be observed by application code if that is desired.

7.3.2.5 <textbox> The <textbox> tag allows the user to enter text. It is four tags in one: normal, password, multiline, and autocompleting. <textbox> has an XBL definition. The <textbox> tag is implemented using the HTML <input type="text"> or <textarea> tags. The standard <textbox> tag has the following special attributes:

```
value disabled readonly type maxlength size multiline
```

These attributes match the HTML `<input type="text">` tag's attributes, except for `multiline` and `type`. `Multiline` can be set to `true`, and `type` can be set to `password` or `autocomplete`. If set to `password`, `<textbox>` acts like HTML's `<input type="password">` tag; however, they are the same otherwise. If the `multiline` attribute is set to `true`, then `<textbox>` has an alternate set of attributes:

```
value disabled readonly rows cols wrap
```

These attributes match the attributes of HTML's `<textarea>` tag. Because XUL has no anonymous content, initially displayed text cannot be included between start and end tags as it is for HTML's `<textarea>`.

For all these first three variants, an initial value for the textbox can only be set if the textbox is not multiline. For multiline textboxes, an initial value must be set from JavaScript using the `value` property of the matching object. An example for a `<textbox id="txt"/>` tag is

```
document.getElementById("txt").value = "initial text";
```

The fourth `<textbox>` variant occurs if `type` is set to `autocomplete`. It is a complicated tag, with a great deal of application-specific functionality. After substantial testing, this book recommends against using the `<textbox type="autocomplete">` tag as a basic tool for your own application. This is because it is not general enough yet.

Where this last tag excels is within the Classic Browser. A simple declaration as follows is enough to provide easy access to the browser's history, email address, and LDAP address mini-databases:

```
<textbox type="autocomplete" searchSessions="addrbook"/>
```

Because none of these mini-databases are available to a standalone application, this use of `<textbox>` is rather limited. It is possible, however, to apply it to any mini-database, even one as simple as a JavaScript array of values. To do so is to struggle with an interface that isn't all that clean or even intended for reuse. With about a day's work, you can analyze the `autocomplete.xml` XBL definition for this tag and come up with some workarounds and hacks for your data. This might do for a one-off use, but you might as well create your own autocomplete XBL definition from scratch if you have a serious application in mind.

7.3.3 Form Submission

XUL does not tie form elements to a target URL the way HTML does, and yet the purpose of filling in a form is to have the information go somewhere. In Mozilla, the options for storing data are as general and as wide as for any programming environment. That is not much comfort if you are trying to produce a quick prototype. Fortunately, two options in Mozilla allow XUL form data to be submitted to a Web server efficiently.

7.3.3.1 HTML Submission of XUL Forms The first XUL form submission method, which is very quick and dirty, is to use XML namespaces. It is possible to create a document that starts with XUL but that includes all of HTML's features. Listing 7.1 shows such a document.

Listing 7.1 Mixing of HTML and XUL forms.

```
<?xml version="1.0"?>
<!DOCTYPE window>
<window
  xmlns=
"http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
  xmlns:html="http://www.w3.org/1999/xhtml">
<vbox>
  <script>
    function copy()
    {
      var getID = document.getElementById;
      getID("h1").value = getID("x1").value;
      getID("h2").value = getID("x2").value;
      return true;
    }
  </script>
  <html:form action="test.cgi" method="GET"
             enctype="application/x-www-form-urlencoded">
    <html:input id="h1" type="hidden"/>
    <html:input id="h2" type="hidden"/>
    <radiogroup>
      <button id="x1" label="Button 1" type="radio"/>
      <button id="x2" label="Button 2" type="radio"/>
    </radiogroup>
    <html:input type="submit" onsubmit="return copy();">
  </html:form>
</vbox>
</window>
```

This document displays a form consisting of two XUL buttons and one XHTML submit button. It's not possible to put buttons in a radio group in plain XHTML, but that is done here because Mozilla supports mixing XUL into HTML. The XHTML form elements are linked to the form submission process, but the XUL form elements aren't. A simple JavaScript function copies the required data to hidden fields before the submission occurs.

It is possible to create a legal XHTML + XUL document without resorting to `xmlns` namespaces. To do that, start with a "pure" XHTML (or XUL) document, and add DTD entities for the XUL (or XHTML) application to the `<!DOCTYPE>` declaration. Such a document does not include the special `xmlns` triggers that Mozilla uses to detect the document type. That means that no special processing (support) for those extra tags will be used. This lack of detection is the reason why adding an `<A>` tag to a XUL document does not result in an XHTML link being rendered.

7.3.3.2 `XMLHttpRequest` Object The second XUL form submission technique uses the `XMLHttpRequest` object. This is a scriptable AOM object available to all XML documents, much like the `Image` object that is available to HTML documents. It allows HTTP requests to be submitted directly from JavaScript. A server response to such a request does not replace the currently displayed document. Such a response document is just read in as a big string. This `XMLHttpRequest` object is based on the following XPCOM component:

```
@mozilla.org/xmlextras/xmlhttprequest;1
```

This component implements the `nsIXMLHttpRequest` and `nsIJSXMLHttpRequest` XPCOM interfaces, which are well explained in their definition files. These interfaces allow an HTTP request to be submitted synchronously or asynchronously. Synchronous submission means that the script halts until the full response is received. Asynchronous submission is a "fire and forget" system, except that progress can be tracked, and the final result can be recalled. Listing 7.2 shows synchronous requests at work.

Listing 7.2 Examples of synchronous `XMLHttpRequests`.

```
var req    = new XMLHttpRequest(); // Request
var res    = null;                 // Response
var params = encodeURI("param1=value1;param2=value2");

// -- GET request

req.open("GET", "test.cgi" + "?" + params);
req.send("");
if ( req.status / 100 == 2 )  // HTTP 2xx Response?
  res = req.responseText;

// -- POST request

req.open("POST", "test.cgi");
req.send(params);
if ( req.status / 100 == 2 )  // HTTP 2xx Response?
  res = req.responseText;
```

The function `encodeURI()` is the ECMAScript version of `escape()`; both are supported. The second argument to `open()` is any valid URL. Because `send()` doesn't return until the HTTP request-response pair is complete, the programmer should provide the user with some kind of "waiting ..." indicator just before `send()` is called so that the user knows that the application hasn't locked up.

Asynchronous form submissions are useful when multiple HTTP requests are needed. It is more efficient to schedule all the requests at once and then to check back later on progress. The simplest way to perform an asynchronous submission is to put a synchronous submission into a function and to schedule the function with `setTimeout()`. Listing 7.3 shows a more formal and structured approach using the `nsIXMLHttpRequest` interface.

Listing 7.3 Example of an asynchronous `XMLHttpRequest`.

```
var req = new XMLHttpRequest(); // Request
var res = null;                 // Response
var url = "test.cgi?text1=value1";

// Stuff specific to the async case

function finished() { res = req.responseText; }

function inprogress() {
  if ( req.readyState != req.COMPLETED ) {
    res = "Waiting ...";
    setTimeout(inprogress, 100);
  }
}

req.COMPLETED = 4;            // from the interface
req.onload = finished;

// -- GET case (POST is similar)

req.open("GET", url, false);  // false == asynchronous
req.send();

// next statement executes immediately.
setTimeout(inprogress,100);
```

In this example, the `send()` method returns almost immediately, leaving the HTTP request still in progress. The function `finished()` is installed as an event handler that fires when the response finally does complete. In between those two times, `setTimeout()` is used in a simple way to check the request progress regularly. The user might not require progress reports when such an asynchronous request is sent. Nevertheless, the programmer must take care to ensure that actions taken by the user subsequently don't confuse the handling of the response when it arrives. For example, users buying stocks shouldn't be able to empty their bank accounts while the stock purchase is in progress.

7.4 MENUS

Menus and forms go together like wine and cheese. Mozilla's menu tags are built out of simpler XUL tags, just like most other XUL tags discussed so far. Menus, however, are a bit more complicated for several reasons:

- Popup menus don't fit into the normal flow of a two-dimensional XML document.
- Menus contain many separate values (menu items), not just one value.

- Menus have complex internal structure.
- Menus can be used separately from forms, but they still need to fit in with the constraints imposed by other form elements.

Mozilla does not use XUL menus for so-called listboxes, which lie flat inside an XML document. The `<listbox>` tag fills that role. XUL menus are only used for popup-style menus. Mozilla's menus follow the design rules that apply to simple form elements and so can be considered a simple form element of sorts.

This topic describes XUL's menu support, starting from the smallest tag. Tags that an application programmer might use for simple popup menus include

```
<menulist> <menupopup> <menuitem> <menuseparator> <menu> <button>
     <toolbarbutton>
```

The following additional structural tags might be used by an application programmer who needs to examine the XUL menu system closely. Generally, they do not have to be stated in an XUL document.

```
<dropmarker> <arrowscrollbox> <scrollbox> <autorepeatbutton>
```

Figure 7.3 illustrates the structure of a fully featured but simple XUL menu with two menu items.

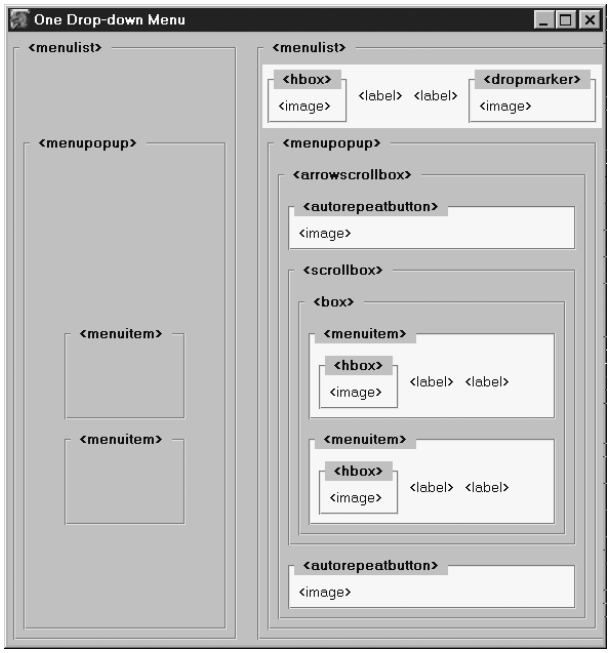


Fig. 7.3 Full structure of an XUL menu.

This figure shows two aspects of the same menu. The left-hand breakdown is the tags that the application programmer specified. The right-hand menu is made up of the tags actually created by the XUL/XBL system inside Mozilla. It's obvious that there are many tags at work. The menu system is designed so that it is always built from a full set of menu content, but the parts that aren't needed are hidden. For example, the <autorepeatbutton> tags are hidden when the list of menu items is small enough to appear all at once. A second example is the various labels and images. Each menu item can have an icon and a shortcut key, as well as the ordinary text of the menu item, for a total of one <image> and two <label> tags. These tags are exposed by tag attributes supplied by the programmer and by the circumstances of the menu display.

The complex tag structure of menus does not translate into complex event handling. The command event and oncommand event handler are all you need to hook up simple menus to your application logic.

Various tag-level customizations of XUL menus are also possible. On the inside, the <menuitem> tags can be replaced with <menuseparator> or <menu> tags. On the outside, the tags that surround the <menupopup> tag can be reorganized by changing the <menulist> tag to something else.

7.4.1 <menuitem>

The <menuitem> tag stands for a single menu option. It has several XBL definitions, of which just one is applied. Attributes with special meaning to <menuitem> are

```
type disabled
image validate src checked
label accesskey
acceltext crop
value
```

The type attribute is not used in the simple case, but it can be set to "radio" or "checkbox". These options will cause the menu item to look and act like the equivalent form control. The disabled attribute grays out the menu item so that it can't be selected.

The next row of attributes relates to the icon for the menu item. When specified, this icon appears by default on the left, but standard box alignment attributes like dir can change this. Both image and src specify the URL of the icon's image. For a standard <menulist> menu, the currently selected item appears on the menu button when the menu is not popped up. If src is used, the menu item's icon will be carried to the menu button if its menu item is selected. If both type and src or image are set, the Radio or Checkbox icon takes precedence over the specified icon. validate has the same meaning as for the <image> tag; checked has the same meaning as for the <radio> and <checkbox> tags.

The third row of attributes is related to the normal text of the menu item. This text can be missing. `label` provides that text. `accesskey` provides an accessibility key for the item, as described in Chapter 8, Navigation.

`acceltext` and `crop` apply only to menus based on the `<menu>` tag. `crop` acts on the menu item text as it does on a `<label>`. `acceltext` provides text to the right of the menu text that states what key combination selects that menu item. It is never cropped. Normally this text spells out a keyboard shortcut, like "Shift-A." If `acceltext` is stated, its value is displayed. If it is not present, and a `key` attribute is present, then the text will come from the matching `<key>` tag. That key tag will provide text from its `keytext` attribute or, failing that, from its `key` attribute or, failing that, from its `keycode` attribute. In all three cases, the text will be modified by the contents of the `<key>`'s `modifier` attribute. This simply reminds the user how any keystroke matches the menu item and links that keystroke to the menu.

The `value` attribute sets an internal data value for the menu item. This value may be used to provide an identifier for the menu item because the item's label may have been translated into another language by the platform's locale system.

Figure 7.4 shows many of these menu options at work. It is built out of several bits of real code. Note that it's not possible to drop down two menus in real life (occasionally technical writing gets an edge on programming).

It appears from Figure 7.4 that `<menuitem>` is quite flexible. This tag also has an armory of styles that can be used to modify its appearance further.

7.4.2 `<menuseparator>`

The `<menuseparator>` tag is a near-anonymous XUL tag with some style rules associated with it. It provides a horizontal line that provides visual separation between menu items. Look at the Mozilla menu toolbar for many examples. Use it in place of the `<menuitem>` tag. This tag has no specialist attributes, but it is occasionally useful to give it an id.

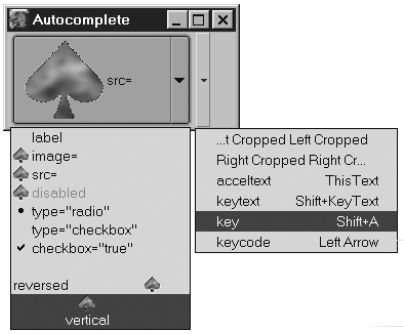


Fig. 7.4 Examples of `<menuitem>` variations.

The Mozilla Platform recognizes this tag when dealing with menu operations.

7.4.3 <arrowscrollbox> and <scrollbox>

The <arowscrollbox> and <scrollbox> tags allow the user to navigate a rectangle of content that won't easily fit on the screen. Both are discussed in Chapter 8, Navigation, but they also have applications to menus. One of each is created automatically when a menu is specified.

The <arrowscrollbox> tag is used to display a menu that is too long to fit on the screen. If the menu is lengthy, then the <autorepeatbutton>s that are part of that tag automatically appear. The easiest way to see the scrolling at work is with the Personal Toolbar of Mozilla's Navigator.

To see this scrolling, make sure that a bookmark folder appears on this toolbar, and that the folder contains many bookmark items. There should be enough items to take up more than half the screen's height. Move the navigator window so that the Personal Toolbar is about halfway down the screen, and press the bookmark folder button for the folder that contains many items. A list of bookmarks with <arrowscrollbox> icons at top and bottom should appear. They will also appear if you set the maxheight attribute on the <menupopup> tag to something small, like 200px.

The <scrollbox> tag implements the moving area inside the <arrowscrollbox>. It shows scrollbars only when they are used inside a <textbox type="autocomplete"> tag. When used inside the <arrowscrollbox>, it has no style class and generally should be left alone.

The buttons on the <arrowscrollbox> can be styled. <arrowscrollbox> cannot scroll sideways, only up and down.

7.4.4 <menupopup>

The <menupopup> tag is the heart of XUL's menu system. It is responsible for creating a window that is outside the normal flow of the XUL document's layout and that might extend outside the borders of the whole Mozilla window. In order to do this, it uses features of the native GUI toolkit. <menupopup> has significant support from the C/C++ side of Mozilla. It also has an XBL definition.

The <menupopup> tag is quite general. Within the small window it creates, it acts like a <vbox>, and most simple XUL content can be put inside. Reasons for doing so are obscure, but this feature does allow for very decorative menus. The only content that is selectable within a <menupopup> are <menuitem> and <menu> tags. A <menupopup> tag may be specified without any content. That content can be added later via JavaScript, templates, or other XUL techniques.

When creating a menu, the application programmer must specify this tag. The attributes with special meaning to <menupopup> are

```
popupanchor popupalign position allowevents
onpopupshowing onpopupshown onpopuphiding onpopuphidden
```

popupanchor chooses the corner of the parent tag that the <menupopup> will appear near—the so-called anchor point. popupalign chooses the corner of the popup menu that will be fixed in position over the anchor. Thus, the popup is aligned corner to corner with its parent tag. Both of these attributes may take on one of these values:

```
topleft topright bottomleft bottomright none
```

The position attribute is an older, less flexible way of specifying this alignment and should be avoided. It is obsolete.

The allowevents attribute can be set to true, but this is not recommended in ordinary uses. The Mozilla Platform has special event processing for menus. Figure 7.4 illustrates the complex set of tags that make up a menu. If the full DOM Event model were to apply to these tags, then selecting a menu item would generate many events. The Mozilla Platform reduces these many events down to a single, manageable command event.

The remaining four attributes are event handlers for the DOM Events noted earlier. They are paired event targets. The showing and shown variants fire at the start and end of the process that reveals the popup. The hiding and hidden variants fire at the start and end of the process that takes down the popup. The display process works in three stages, and the Mozilla code marks each stage by setting one of these three temporary attributes to true:

```
menutobedisplayed menugenerated menuactive
```

The first indicates that something is going to happen but hasn't happened yet. The second indicates that the content has been assembled for the menu. The third indicates that the menu is revealed and has the focus.

7.4.5 <menu>

The useful content that a <menupopup> can hold consists of three tags: <menuitem>, <menuseparator>, and <menu>. <menu> is very similar to <menuitem> except that it generates a new menu to the side of the existing one. The attributes specific to <menu> are

```
label accesskey crop acceltext disabled
```

These attributes have the same meaning as they do for <menuitem>. A <menu> tag cannot contain an icon representing the menu's action as content, but an icon can be sneaked in with a suitable style. It does contain an automatically supplied arrow-like icon that provides a hint that a submenu exists. The only tag that can appear inside a <menu> tag is a <menupopup>.

The <menu> tag is also used in menu bars:

```
<menulist id="foo" editable="true"/>
```

7.4.6 <menulist>

The <menulist> tag is the top-level tag for simple XUL menus. It is the wrapper around a <menupopup> tag. It gives the user something to look at and interact with when the popup is not displayed. It displays a button-like widget showing the currently selected menu item, and a dropmarker used to reveal the menu. <menulist> always has a currently selected menu item.
The <menulist> tag can only contain a <menupopup> tag. The attributes with special meaning to <menulist> are

```
src label crop accesskey disabled editable value
```

The attributes are all the same as for <menuitem>, except that disabled set to true disables the whole menu popup, and editable is new.

The editable attribute is something of a misnomer. If set to true, the label on the <menulist>'s button is replaced with an HTML <input type="text"> textbox. The user can then type in a string. When focus leaves the <menulist>, this string will be compared to the existing menu items. If it matches the main text of one such item, that item will be the currently selected one. All the menu items are treated as though they had type="checkbox" applied. The editable attribute does not provide insertion, deletion, or update of menu items.

7.4.7 Menu Variations

The <menulist> tag is a wrapper for the <menupopup> tag. There are other wrappers. For completeness, the entire list is

```
<menulist>
<menulist editable="true">
<menu>
<button type="menu">
<button type="menu-button>
<toolbarbutton type="menu">
<toolbarbutton type="menu-button">
<textbox type="autocomplete">
<menubutton>
```

Except for <textbox>, using any of these tags means adding a <menupopup> tag as the sole content. The <menubutton> tag is an anonymous tag used to hold styles and handlers for menus that appear in toolbars (see Chapter 8, Navigation).

Under MacOS, it is possible to add a Preferences menu item and submenu to the MacOS Application menu. To do this, ensure that a suitable <menuitem> tag has the special id of id="menu_preferences".

7.5 STYLE OPTIONS

As for static content and buttons, Mozilla duplicates much of the information about forms and menus in the style system.

Perhaps the most powerful style features are style rules that affect which XBL binding applies to which XUL tag. The `type` attribute (for `<button>`, `<toolbarbutton>`, and `<textbox>`) and the `editable` attribute (for `<menulist>`) are all specified in style rules. Those rules direct the basic XUL tag to a specific binding, which ultimately affects its content, appearance, and behavior. After you have learned XBL, it is possible to add more such bindings and style rules. The `type` attribute can then provide more widget variants.

7.5.1 CSS Styles from HTML Forms

Some standard HTML form styles are available for use in XUL forms. Table 7.2 lists them. See the CSS3 draft document at *www.w3.org/TR/1999/WD-css3-userint-19990916* for details of the newer styles.

Table 7.2 XUL style extensions that follow HTML forms

Property or selector	Values	Use
:-moz-drag-over	(selector)	Any element under the drag item when dragging with the mouse
:-moz-selection	(selector)	Near complete support for CSS3 ::selection, as of version 1.21
-moz-key-equivalent	See draft	CSS3 draft key-equivalent
-moz-outline-radius	As for margin, but use px	These attributes mimic the margin attribute, except that they dictate the roundness of the corners of a border or outline
-moz-outline-radius-topleft -moz-outline-radius-topright -moz-outline-radius-bottomleft -moz-outline-radius-bottomright	Px	Roundness of a single outline corner; 0 for square
-moz-outline	As for outline	Near-complete CSS2 outline
-moz-outline-color	As for color	Near-complete CSS2 outline-color
-moz-outline-style	As for border-style	Near-complete CSS2 outline-style
-moz-outline-width	As for border-width	Near-complete CSS2 outline-width
-moz-user-focus	See draft	CSS3 draft user-focus
-moz-user-input	See draft	CSS3 draft user-input
-moz-user-modify	See draft	CSS3 draft user-modify
-moz-user-select	See draft, plus -moz-all	CSS3 draft user-select

The outline-related styles follow to a degree the border styles (see "Style Options" in Chapter 2, XUL Layout, for an example). The `-moz-all` values for the `-moz-user-select` style allows more than one form element to be selected at once. This is useful if you are developing a visual IDE for XUL and need to be able to group form elements with a select action.

7.5.2 Custom Widget Display Types

The `<menupopup>` tag relies on the following style:

```
display: -moz-popup
```

The `-moz-appearance` style property makes a given tag take on the appearance of the native theme. These native theme appearance types work for the widget-like tags described in this chapter:

```
radio checkbox textfield menulist menulist-button menulist-text
     menulist-textfield checkbox-container radio-container
```

7.6 HANDS ON: NOTETAKER EVENTS AND FORMS

After many chapters laying the groundwork, we've now got some useful widgets. In this session on the NoteTaker tool, we'll clean out some of the oddments the dialog box has accumulated and replace them with normal XUL form elements. We'll get rid of the properties file and submit the entered data to a Web server. If the dialog box appears as the result of an HTTP GET request, we'll populate it with parameters from that request as well.

Cleanup comes first. We'll throw away the `subpanel` attribute and the `handle_click()` code. If we still choose to disable any of the content, it now makes sense to use the form tags' own disabling features. That means all the broadcasters and half of the actions and keys can also be cleaned out. The task that `load_data()` performs will still be needed, but we'll throw away the current implementation.

Finally, we'll throw away the `boxes.css` / `boxesTemp.css` diagnostic styles and their horrid placeholder `<box>` tags. Instead of those, we'll have

1. A single line `<textbox>` for the Summary subpanel
2. A multiline `<textbox>` for the Details subpanel
3. Two `<checkbox>` tags for the Options subpanel
4. Four small `<textbox>` tags for the Size subpanel

As shown in Listing 7.4, the matching tags are trivial.

Listing 7.4 New form element tags for the NoteTaker dialog box.

```
<textbox  id="dialog.summary"/>
<textbox  id="dialog.details multiline="true" flex="1"/>
```

```
<checkbox id="dialog.chop-query" dir="rtl" label="Chop Query"
          checked="true"/>
<checkbox id="dialog.home-page" dir="rtl" label="Home Page"
          checked="true"/>
<textbox  id="dialog.width" value="100" maxwidth="3" size="3"/>
<textbox  id="dialog.height" value="90" maxwidth="3" size="3"/>
<textbox  id="dialog.top" value="80" maxwidth="3" size="3"/>
<textbox  id="dialog.left" value="70" maxwidth="3" size="3"/>
```

This cleanup, together with a little layout adjustment, yields a dialog box similar to that in Figure 7.5.

Before replacing `load_data()`, we can experiment briefly. The DOM objects for XUL formlike tags are very similar to the DOM objects for HTML formlike tags. Either by looking at the DOM 2 HTML standard, or the XBL bindings for `<checkbox>` and so on, or just repeating from memory tricks used in Web pages, we can get a long way. Suppose that we replace this code in `load_data()`

```
desc  = document.createElement("description");
desc.setAttribute("value",value);
box   = document.getElementById("dialog." + names[i]);
box.appendChild(desc);
```

with this

```
if (box.value) box.value = value;
```

Since most form "elements" have a value property, we've succeeded in loading most of the properties file into the form with a trivial effort.

Having cleaned up the form, we would like to process the data the user enters, which is the description of a note. Until now, we have used a read-only properties file for that description. Now we want to read and write the form. There are any number of places to put the information, but we'll try using a Web server. Perhaps one day we'll be able to access that note from any Mozilla Browser in the world.

To save the NoteTaker note, we use the code described in this chapter, which we'll slightly customize and put in the `action()` method. We also keep

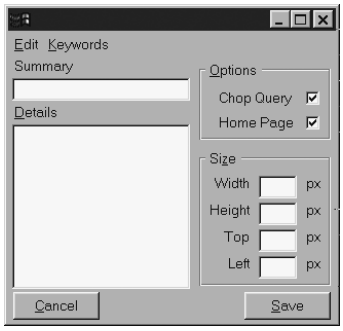


Fig. 7.5 Form-based dialog box.

the names array, formerly local to `load_data()`, but now it's a global array. Listing 7.5 shows this new logic.

Listing 7.5 Posting a NoteTaker note to a Web server.

```
if (task == "save")
{
  var req    = new XMLHttpRequest(); // Request
  var params = "";

  var i = names.length -1;  // build up params
  while (i>=0)
  {
    var widget = document.getElementById("dialog." + names[i]);
    if (widget.tagName == "checkbox")
      params += names[i] + "=" + widget.checked + ";";
    else
      params += names[i] + "=" + widget.value + ";";
    i--;
  }

  params = encodeURI(params);
  req.open("POST", "test.cgi");
  req.send(params);
  if ( req.status / 100 != 2 )  // HTTP 2xx Response?
    throw("Save attempt failed");
}
```

Of course, we also need a `test.cgi`, or some equivalent on the Web server. If we're running from the chrome, `test.cgi` can be replaced with any URL for any Web server.

The NoteTaker tool works with the Classic Browser, so it's unlikely to be loaded from a remote URL. Nevertheless, it's possible that arguments could be passed to it when the window is created. Since it shares the same programming environment as all the rest of the chrome, those arguments could be passed in any number of imaginative ways. One simple way is to append HTTP GET–style parameters to NoteTaker's URL. Instead of using

```
chrome://notetaker/content/editDialog.xul
```

the same document can be loaded using

```
chrome://notetaker/content/editDialog.xul?top=440;left=200
```

This information can be examined at any subsequent time using the `window.location` AOM property. We'll do that by creating a new task for the `action()` method: This one will be called `'load'`. Rather than matching any element of the GUI, `'load'` is a task that is only performed by scripts. The new `onload` handler of the `<window>` tag will be

```
onload="action('load')"
```

The logic added to the `action()` method is shown in Listing 7.6. The `window.location` object is one of very few AOM objects that exists in both HTML and XUL documents. It provides an object with the `nsIDOMLocation` interface and is useful in XUL only as used here.

Listing 7.6 Reading form parameters from a GET-style URL.

```
if (task == "load" )
  {
    var pair, widget;
    params = window.location.toString();
    if ( params.indexOf("?") == -1 )
      return;
    params = params.replace(/.*\?/,"");
    params = params.split(/[;&]/);
    i = params.length - 1;
    while (i >=0 )
    {
      pair = params[i].split(/=/);
      pair[0] = decodeURI(pair[0]);
      pair[1] = decodeURI(pair[1]);

      widget = document.getElementById("dialog."+ pair[0])
      if (widget.tagName == "checkbox")
        widget.checked = ( pair[1] == "true" );
      else
        widget.value = pair[1];
      i--;
    }
  }
```

This code uses the regular expression support in ECMAScript 1.3. The `replace()` method is used to chop off all the characters leading up to the `?` symbol, which starts the parameter string. `split()` divides the remaining string into an array of strings, with the separation points everywhere that a regular expression match occurs. In this case, the match occurs on each `;` or `&` character. Finally, each `param=value` pair is split again, and the two halves are used to find a form element with a matching id. That element is then updated.

7.7 Debug Corner: Diagnosing Events and Forms

Forms and events aren't that challenging, unless you are in the business of constructing your own widgets. A few standard techniques can make life easier.

7.7.1 Unpopping Menus

The `<menupopup>` tag only appears when you click on its associated button. This can be a nuisance if you just want to see that the constructed contents

are correct. You can embed the popup in the normal flow of the XUL document by changing its display style as follows:

```
menupopup { display: -moz-box ! important }
```

If you do this, the popup won't work interactively any more, but at least you can see its contents easily. Figure 7.6 shows the same document as in Figure 7.3, but with this style added.

Doing this is not recommended as an implementation strategy, or even as a stable testing technique; it just happens to work and is convenient.

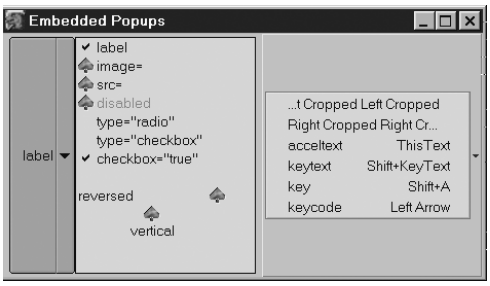


Fig. 7.6 Examples of embedded <menupopup> contents.

7.7.2 Picking Apart Complex Widgets

Complex widgets like <menulist> take some work to understand, particularly if you don't like the default content, appearance, or behavior. It always helps if you can see the unseen.

The simplest strategy is to analyze a given tag with the DOM Inspector. This tool reveals tag content, tag ids, and tag style classes at the touch of a button. It's so easy you'd be crazy not to experiment with it.

Another strategy used in this chapter and elsewhere in this book is to use a diagnostic stylesheet. Some of the examples in this book are enhanced with simple styles that provide a few extra visual hints. Listing 7.7 shows these informal styles.

Listing 7.7 Example of an asynchronous XMLHttpRequest.

```
label { border: dashed thin; padding: 2px; margin: 2px;}
image { border: solid thin; padding: 2px; margin: 2px; }

.radio-check { width:20px; height:20px;}
.checkbox-check { width:20px; height:20px;}

vbox, hbox, box, deck
{ border: solid; padding: 2px; margin: 2px; border-color: grey;}

* { font-family:Arial; font-weight: bold;}
```

This is pretty simple stuff, except possibly for the checkbox styles, which had to be dug out of the DOM Inspector. Without these two styles, the check icons are reduced to tiny sizes by the padding added in the other rules.

7.7.3 Finding Object Properties for Forms and Menus

This chapter has covered the XUL interface to Mozilla's forms and menus but not the JavaScript interfaces to those tags' DOM objects. To find AOM and DOM properties and methods, there are several approaches.

- **Guess**. Many of the XUL interface properties exactly mimic those of HTML forms. `SelectedIndex`, `focus()`, and `blur()` all work as you might expect. They're also documented in the DOM 1 Core standard, under HTML.
- **Use the DOM Inspector**. Select a tag in the left panel, and choose JavaScript Object from the toolbar menu at the top left in the right panel. All property and method names are revealed. To find values for object properties, consult this book or the CSS2 and CSS3 standards.
- **Examine XBL definitions**. The XBL bindings for each form or menu widget are located in the chrome in `toolkit.jar`. It's easy to treat those definitions as a form of help after a quick read of Chapter 15, XBL Bindings. XBL definitions list the parameter names for each method, which the DOM Inspector does not yet do.

7.8 SUMMARY

Form and menu widgets are a vital first access point for users needing to interact with Mozilla-based applications. XUL supports all the simple form controls (or elements) with which users of Web browsers are familiar. XUL's support, however, is flexible and only begins with simple form and menu controls. That support goes much further, and it is not bound rigidly to user actions as HTML forms are.

Unlike HTML, XUL applications can only submit forms across the Web via a script. Such a submission, however, can be tailored more flexibly than any HTML form submission can.

Forms and menus are all interesting enough, but fiddling with such things won't allow the user to go anywhere or explore other parts of a possibly complex application. Support for such navigational activities is the subject of the next chapter.