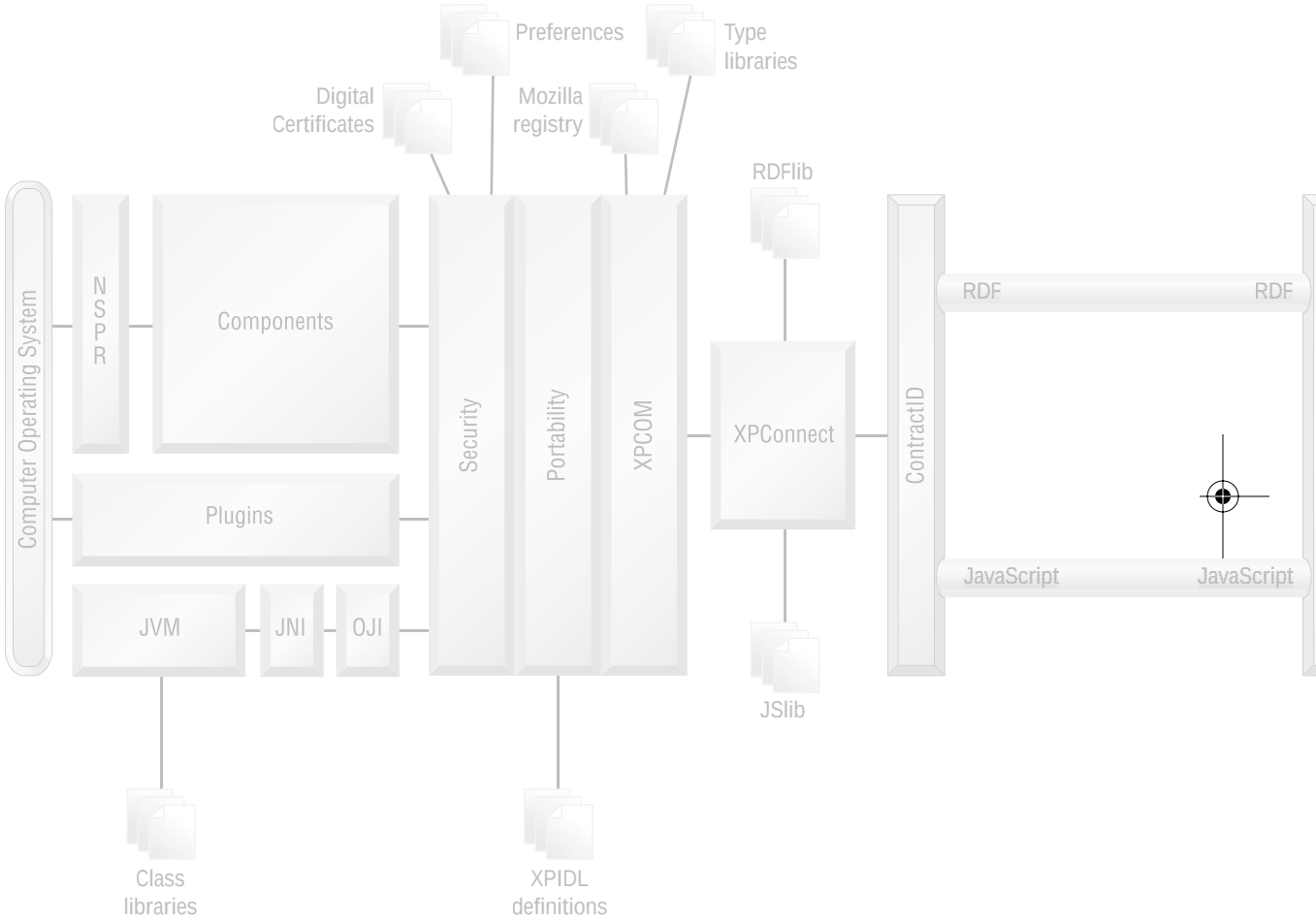
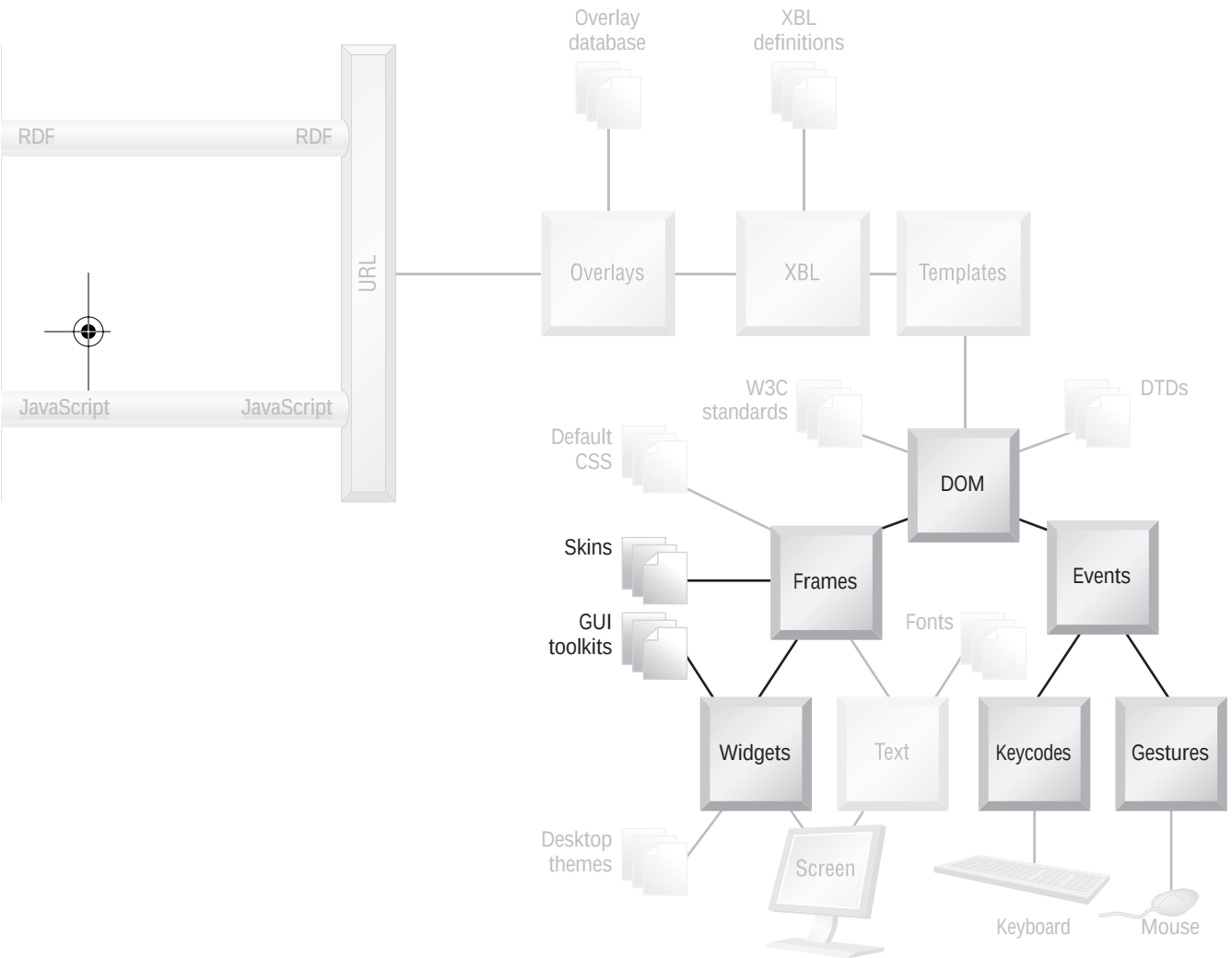

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
RDF
Plugins
JavaScript
JavaScript
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 13

Listboxes and Trees



This chapter describes the construction of XUL's most powerful widgets: <listbox> and <tree>. These tags are designed for data-intensive applications.

The <listbox> tag provides an inline, scrollable, multirecord list, similar to a menu, but it may contain more than one column. The <tree> tag provides a flat or hierarchically indented list of tree-structured records. <tree> is similar to Windows Explorer on Microsoft Windows, or better yet, the Finder in the Macintosh. <tree> can do everything that <listbox> does, and more, but <listbox> has a simpler and more direct syntax. The syntax of <tree> can become quite complex.

To see a <tree> in action, just open the Classic Mail & News client. The three-pane arrangement consists of three <tree> tags, one per pane. A similar example is the Bookmark Manager, which displays all the available bookmarks in a single <tree>. Spotting a <listbox> is harder because that functionality can also be provided by <tree>. The Appearance, Themes panel of the Preferences window in Classic Mozilla is an example of a <listbox>.

Applications focused on data entry or data management are more tightly designed than Web pages. They tend to pack an available window full of information. They don't waste space on graceful layout. A packed display needs widgets that can economically organize the display of structured data. <listbox> and <tree> have this design constraint in mind. Both tags manage content in a scrollable window that is highly interactive.

Another feature of data management applications is multirecord (or record set) displays. Applications as diverse as email clients, order-entry, point-of-sale, and network management can all display several records at once. Traversing through a set of structured data items is data browsing in the same way that clicking through hypertext links is Web browsing. The interactive, scrolling nature of <listbox> and <tree> is perfect for such uses.

These final XUL widgets require that we visit the GUI of the Mozilla Platform yet again. The NPA diagram at the start of this chapter shows the bits of Mozilla engaged by simple use of these XUL tags. Both <listbox> and <tree> are very fully featured tags. They extend right across the user-interface features of the platform, as well as extending up into the DOM, frame, and CSS2 styling infrastructure. Both tags have features that allow scripts to pry further into the frame system than any other XUL tag. Their other novel feature is support for multiple selection. This chapter covers all that, but leaves the equally complex matter of data-enabling these widgets to Chapter 14, Templates.

Mozilla presents many options for displaying structured data, such as the humble form. Before turning to <listbox> and <tree>, we briefly consider another simple system—the text grid.

13.1 TEXT GRIDS

Text input tags can be arranged into a text grid. A text grid is an informal term for a two-dimensional array of editable boxes. An obvious example of a

text grid is a spreadsheet, with its columns and rows. Small text grids are also ideal for the detailed part of master-detail forms and for working with sets of records. XUL has no direct support for text grids, but such things are easy to design using the <textbox> tag.

The Web tends to ignore the flexibility of textboxes. On the Web, it is customary to see data-entry forms designed so that individual fields are spaced well apart. Requests for contact details or for purchase order details are often displayed so that there is one form element per line. This makes text boxes appear to be bulky and spacious things.

In fact, the HTML <INPUT> tag and the XUL <textbox> tag can be styled to be quite slim. Only simple styles are required:

```
textbox {
  border : solid thin;
  border-width : 1px;
  padding : 0px;
  margin : 0px;
}

input:focus { background-color : lightgrey; }
```

The second style serves to ensure that the field with the current focus is background-highlighted when it has the focus. Recall from Chapter 7, Forms and Menus, that <textbox> also contains an <html:input> tag. Figure 13.1 shows an example application using these thinned-down textboxes.

Each <textbox> is the contents of one cell in a <grid>. The XUL code is routine. XUL's navigation model and focus ring ensures that each <textbox> can be tabbed into in turn, and that each field is background-highlighted when it receives the focus. This results in the look and feel of traditional data management applications, which are fast and efficient for data entry operators to use. Properties dialog boxes, typically accessed from content menus, can't possibly compete for speed.

A collection of <textbox> tags is hardly a complete solution—the whole back end of the application needs to be added. Such a XUL page could end up with many event handlers whose only purpose is to coordinate data against the user's navigation. There is significant scripting design work required for such a window (called a screen in older jargon).

Web-based systems do not follow this kind of look and feel for many reasons. There is the difficulty of POSTing multiple records at once; the need to

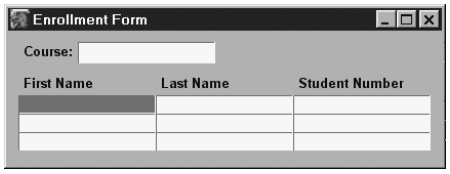


Fig. 13.1 A simple text grid using the <textbox> tag.

provide accessibility support; the complexity of implementation; and the likelihood that the sizes of browser windows will vary greatly. XUL applications, built to be vertical solutions, are not always so constrained, and the performance gains delivered to users may be tempting.

Both <listbox> and <tree> improve on and specialize the concept of a text grid. A text grid made out of XUL tags is the most general arrangement possible. It may also be the most useful if a lot of data entry is required.

13.2 LISTBOXES

The <listbox> tag is similar in construction to the <grid> tag, but in appearance and behavior it is more like the <menulist> tag. A listbox is a vertically arranged set of records or rows, where each record has one or more subparts.

HTML's <SELECT rows= > tag has a similar implementation to <listbox>. That HTML tag produces an inline menu rather than a popup one. The <SELECT> tag is both robust and standard, but the <listbox> tag is not yet either. In Mozilla versions up to 1.4 at least, <listbox> is somewhat fragile, so tread carefully when using it. Despite that weakness, it is a powerful tool when used properly.

13.2.1 Visual Appearance

To see a <listbox> at work, open the Mozilla Preferences dialog box (Edit | Preferences) and look at the Appearance, Themes panel on the right. That white panel displaying theme names like Classic and Modern is a listbox.

Figure 13.2 shows two listboxes with most of the available features visible.

The listbox on the left is a one-column listbox. This format is used for HTML's <SELECT>. The listbox has height and width style rules that have defaults of 200px each. If the box area is forced smaller by layout issues, then a vertical scrollbar will appear, and any contained <label> text might be cropped. Each row can also contain a leading icon and a leading checkbox, as shown. In a real application, all rows would be iconized or checkboxed, not just a few sample rows. In this example, row 4 was checked by clicking on the row, and then row 3 was selected as the current row.

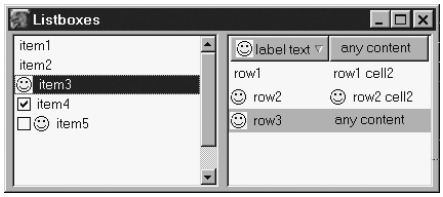


Fig. 13.2 Two listboxes showing popular features.

The listbox on the right of Figure 13.3 is a multicolumn listbox. It has two columns, but any number of columns is possible. The top row is an optional row of column headings. The first column heading has an icon on both left and right of its text. The left icon is an arbitrary image, added using a style. The right icon is a special-purpose image that shows the sort order of the rows underneath that heading, and therefore of all rows. That right icon is placed with an XML attribute. The other rows (also called *items*) of this listbox contain two cells each. Icons can be placed in these cells, although it is not very meaningful to do so. Checkboxes can also be placed in these cells, but there is no automated means of checking them, and it is almost meaningless to put them in. In the screenshot, the third row of the second listbox is selected (it appears light gray) but the second listbox doesn't have the focus. The listbox on the left-hand side has the focus—its focussed row is dark gray (blue normally).

The contents of a cell can be a simple `<label>` or a boxlike tag that holds arbitrary content. If arbitrary content is used, layout becomes more of a challenge, and the listbox doesn't neatly crop its content when resized. This can cause CSS2 overflow and other messy effects, so simple labels are the safest kind of content. A multicolumn listbox can have a row with fewer cells than there are columns, and that will work (including checkboxes), but that use is not recommended.

If a multicolumn listbox gains a vertical scrollbar, then that scrollbar does not include the optional header row.

13.2.2 Construction

Figure 13.3 repeats Figure 13.2, but with diagnostic styles turned on.

As before, thin dotted lines are labels, thin black boxes are images, and thick gray lines are boxes. Smileys are slightly squashed on the left only because extra border styles have distorted the layout slightly. The right-hand listbox is quite confusing with all its boxes revealed, but a bit of study reveals

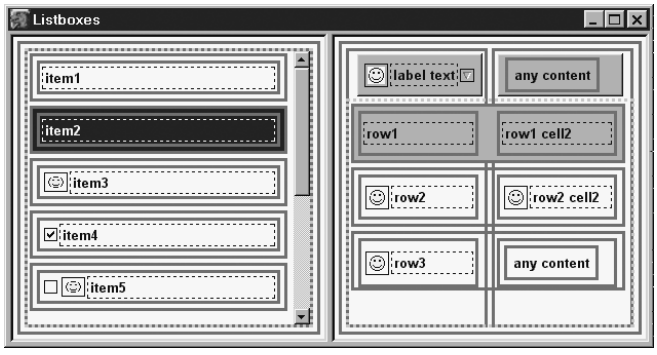


Fig. 13.3 Two listboxes with their internal structure exposed.

that those boxes are very similar to the <grid> boxes discussed in Chapter 2, XUL Layout. A grid is used as the core of a <listbox>'s layout strategy, with <button> tags for column headers and <label> tags for the default cell content. The column 2 header and the right half of row 3 show that such a label can be replaced with an arbitrary box of content. The two dark rows (one in each listbox) show the extent of the highlighting that results from selecting a row. This highlighting is just a background style applied to something equivalent to the <grid> tag's <row> subtag.

Figure 13.4 also reveals a box specific to the layout of <listbox>. This is the gray, thick, dotted line in both of the listboxes displayed. This box surrounds all the list items or rows. This box is used extensively in the implementation of the <listbox> system. It makes the <listbox> layout system unique and separate from <grid>.

The <listbox> tag, and its related tags, has XBL definitions stored in the file listbox.xml in toolkit.jar in the chrome. The Mozilla Platform has extensive C/C++ support for listboxes as well.

Which listbox tags exist depends on your perspective. The tags an application programmer uses are different from the tags used by the platform to generate the final XUL. Table 13.1 describes all the listbox tags in Mozilla and their status as of version 1.4.

<listbox> uses the pair <listcols> and <listcol> to identify list columns, but it uses <listrows> and <listhead> or <listitem> for rows. The <grid> tags, therefore, do not have exact matches for <listbox>. The special dotted box noted in Figure 13.3 is the border of the <listboxbody> tag. It is a key part of Mozilla's listbox scrolling support.

Table 13.1 Mozilla's listbox tags

Tag name	Useable tag?	Must use?	Internal use only?	<grid> Rough equivalent
<listbox>	✓	✓		<grid>
<listcols>	✓			<columns>
<listcol>	✓			<column>
<listhead>	✓			<row>
<listheader>	✓			one child of <row>
<listheaditem>			✓	one child of <row>
<listboxbody>			✓	
<listrows>			✓	<rows>
<listitem>	✓	✓		<row>
<listcell>	✓			one child of <row>

Listing 13.1 Basic <listbox> containing three items.

```
<listbox>
  <listhead>
    <listheader label="Sole Column">
  </listhead>
  <listitem label="first item"/>
  <listitem label="second item"/>
  <listitem label="third item"/>
</listbox>
```

Listing 13.1 specifies a single-column listbox of three items, with a header that reads "Sole Column." No icons or checkboxes are present. It is similar to many of the boxes present in the Preferences dialog box; for example, in the `Appearance, Themes` panel, or in the `Navigator, Languages` panel. Those examples have no header, however.

The XML that Mozilla constructs from this listbox contains many additional tags, as illustrated in Figure 13.4.

In this fully expanded tag tree, the darker tags match Listing 13.2. The lighter tags with `xul:` prefixes are extra content generated by listbox XBL definitions. This is a complete breakdown of a listbox, except that if there were two columns, the `<xul:listcol>`, `<xul:listheaditem>`, and `<xul:listcell>` tags would appear twice in each tree position, rather than just once.

This tree shows that Listing 13.2 is a specification that uses very condensed syntax—many tags in the final listbox are implied rather than stated. It also shows the similarities and differences between `<listbox>` and `<grid>`. While both have columns and rows, a `<listbox>` has at most two

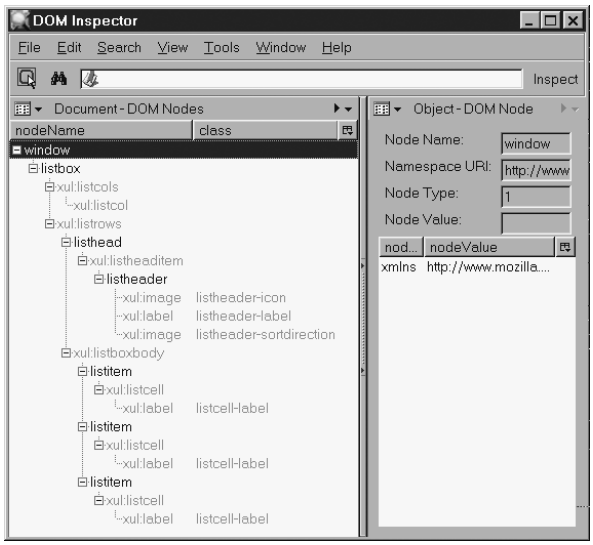


Fig. 13.4 DOM Inspector view of three-item `<listbox>`.

immediate rows, whereas a <grid> can have any number. The items displayed in the listbox are nested within the second row, as though that row were a <vbox>. The first row, which contains the header line, is left out if no header is specified.

Some basic rules of <listbox> construction follow:

1. For each <listbox>, there should be at most one <listcols> and at most one <listhead>.
2. For a <listcols>, if it is present, there should be at least one <listcol>
3. For a <listhead>, if it is present, there should be at least one <listheader>.
4. Items per <listitem> should equal the number of <listcols>, or be one if no <listcols> exists.
5. Do not ever state <listrows>, <listboxbody>, and <listheaditem> explicitly. These tags are for internal use only.

This complex construction process has its pitfalls. The main pitfall is that XUL cannot handle the combination of tags that it in turn generates for <listbox>. If you create a piece of explicit XUL that matches the tags and structure shown in Figure 13.5, it will not work as a listbox, and Mozilla will probably crash. This means that the XML specification of a listbox and its XML implementation are separate and different.

Mozilla may also crash if you use any of the tags in construction rule 5. It will crash if you specify any content for the <listcol> tag. It may become confused, do poor layout, or possibly crash if you deviate much at all from the assumptions that the XUL/XBL processor makes once it sees a <listbox> tag.

Listing 13.2 shows the most extended listbox specification that XUL supports.

Listing 13.2 Extended <listbox> showing all options.

```
<listbox>

  <listcols>      // tag and content optional
    <listcol/>    // can be repeated
  </listcols>

  <listhead>      // tag and content optional
    <listheader>  // can be repeated
    </listheader>
  </listhead>

  <listitem>      // can be repeated
    <listcell>    // can be repeated
    </listcell>
  </listitem>

</listbox>
```

The open and close tags for <listitem>, <listcell>, and <listheader> can be collapsed into singleton <tag/> tags, and attributes can be used in place of the removed tag content. These attributes are discussed under the individual tags. All these tags have XBL definitions in listbox.xml in toolkit.jar in the chrome.

13.2.3 <listbox>

The <listbox> tag has the following special attributes:

```
rows size seltype suppressonselect disableKeyNavigation
```

rows and size dictate the height of the listbox in number of line items. The size calculation is based on the tallest line item that exists, multiplied by the value of the attribute. This is the same as setting the minheight attribute to a fixed number of pixels for each line item. Each line item will be expanded to the height of the tallest item, and so line items are always equally sized. The size attribute is deprecated for XUL; use rows instead. A <listbox> may be dynamically resized by setting the rows attribute from JavaScript.

The rows attribute (and size and minheight) are passed internally to the <listboxbody> tag for processing, so the space any header row takes up is not included in the calculations.

The <listbox> tag works poorly with the maxheight attribute. If <listbox> tags are inside an <hbox>, and sibling tags of the <listbox> have a maxheight that is less than the <listbox>'s maxheight, then the content can overflow downward, resulting in messy layout. The recommended approach when <listbox> has large siblings is to set the <listbox>'s height with height and avoid setting rows entirely.

The seltype attribute determines if multiple rows of a listbox can be user-selected. If it is set to multiple, then that is possible. If it is set to anything else, only a single row will be selected. The dynamics of this arrangement are discussed in section 13.2.11.

The suppressonselect attribute can be set to true. When a user selects an item in a <listbox>, the XBL code for that tag fires a select DOM event, which can be picked up by an event handler on the <listbox> tag. This attribute prevents that event from being created.

The disableKeyNavigation attribute can be set to true. This prevents alphabetic keypresses from changing the current selection when the <listbox> tag has the input focus.

See the "AOM and XPCOM Interfaces" section for a discussion of JavaScript access to <listbox>. <listbox>'s support for templates is discussed in Chapter 14, Templates.

13.2.4 <listcols>

The <listcols> tag is a container for <listcol> tags and has no other purpose. It does not have any special attributes and is not displayed. The <list-

cols> tag should appear before all other content inside a <listbox>, if it appears at all.

If this tag is omitted from a <listbox>, then that is the same as

```
<listcols>
  <listcol flex="1"/>
</listcols>
```

This tag is also a possible site for template-based sort attributes.

13.2.5 <listcol>

The <listcol> tag is never displayed and should never have any content. This tag has no special attributes of its own. In a well-formed <listbox>, the number of columns is determined by the number of <listcol/> tags.

The <listcol> tag has two other purposes. It can be used to give an id to the column, and it can control the layout of the column it stands for. This can be done by adding flex and width attributes or by setting hidden or collapsed to true.

This tag is a possible site for template sort attributes.

13.2.6 <listhead>

The <listhead> tag is a container tag for <listheader> tags. If <listhead> is not present, then there will be no header row for the listbox. It has no special attributes or purpose. This tag wraps all the <listheader> tags in a single <listheaditem> tag. This ensures that each column header has a box-like tag.

13.2.7 <listheader>

The <listheader> tag is based on the <button> tag, meaning that it is itself a <button>. There should be one <listheader> tag per column. From the XBL definition, if no content is supplied, then this tag is equivalent to

```
<button>
  <image class="listheader-icon"/>

  <label class="listheader-label"/>
  <image class="listheader-sortdirection"/>
</button>
```

The first <image> tag can only be set using a style. The <label>'s value and crop attributes are set from the <listheader>'s label and crop attributes. The second image is styled into place according to <listheader>'s only special attribute:

```
sortDirection
```

A value of "ascending" yields an up arrow. A value of "descending" yields a down arrow. A value of "natural" (or anything else) results in no arrow.

If content is supplied, that content appears inside the displayed button.

This tag is also a possible site for template sorting.

13.2.8 <listitem>

The <listitem> tag is used to specify a row in a <listbox>. Use of any other tag to specify a row can cause Mozilla to crash. A <listitem> tag with no user-supplied content is given a single <label> as content. You can alternately specify your own content as one or more child tags of this tag. In that case, there should be one child tag per column, and <listcell> is the obvious choice for that content.

<listitem> supports these special-purpose attributes:

```
label crop flexlabel disabled type checked image selected current
     allowevents value
```

All of these attributes work only for <listbox>es that are single column, unless otherwise stated.

label, crop, and disabled are passed to the interior <label>. The value of flexlabel is passed to the <label>'s flex attribute. A row with disabled set to true can still be selected, but is grayed-out.

The type attribute can be set to checkbox, in which case a checkbox appears to the left of the row. The position of this checkbox cannot be changed with dir. disabled set to true will gray out the checkbox and stop the user from ticking or unticking it.

If the <listitem> has the class listitem-iconic, it can contain an icon. This icon's URL can be set with the image attribute.

The remaining attributes apply to both single- and multicolumn listboxes.

The current attribute is set internally to true by <listbox> processing during selection. If it is true, that means that the <listitem> is the currently selected <listbox> item, or the item just selected in the case where multiple item selection is allowed.

The plain selected attribute is also set to true if this item is selected.

The allowevents attribute, which can be set to true, allows DOM mouse events to pass through the <listitem> tag and into the content that makes up the item. Normally, those events are stopped from propagating when <listitem> receives them. If this attribute is set, then the current row cannot be selected.

The value attribute states the data value that the <listitem> represents. This is for programmer use and is not displayed anywhere.

13.2.9 <listcell>

The <listcell> tag is used to specify a single column entry (a cell) for a row in a <listbox>. Column entries can be specified with a user-defined tag, but the Mozilla Platform checks for <listcell> in a number of places, so it is the right thing to use. The default XBL content of a <listcell> is a single <label>, unless user content is substituted.

The <listcell> tag supports all the attributes that <listitem> supports, except for current, selected, and allowevents. To add an icon to a <listcell>, use the class listcell-iconic. If the <listbox> is multicolumn, then checkboxes will not work when set on a single <listcell>.

The checkbox, icon, and label in a listbox can be reversed with dir="rtl". Other box layout attributes like orient can also be applied to <listcell>, if it makes sense to do so.

That concludes the <listbox> tags.

13.2.10 RDF and Sorting

<listbox> and its related tags can be connected to an RDF document. If this is done, the content of a <listbox> derives from the content of the RDF document. Under such an arrangement, the data in a <listbox> can then be sorted. See Chapter 14, Templates, for detailed instructions.

13.2.11 User Interactions

Listboxes allow for more user interaction than simple XUL form tags. They are at least as versatile as menus.

Listboxes support both keyboard and mouse navigation and have accessibility support. Navigation keys include Tab, Arrow, Paging, Home, and End keys and the spacebar. Mouse support includes clicks, key-click combinations, and the use of scroll wheels. To use the accessibility support, specify the <listitem> contents as label attributes or <label> tags.

Listboxes are members of the focus ring for the currently displayed page. If a <listbox> does not have a currently selected row, then navigating into the listbox from the last member of the focus ring does not provide any visual feedback, but the listbox still has the focus.

The selection of listitems is a flexible matter. If only single item selection is enabled, then selection is much the same as for a menu. If, however, seltype is set to multiple, then multiple list items can be picked. With the mouse, this is done by shift-clicking, to select a contiguous range of items, or by control-clicking, to pick out individual list items not necessarily next to each other. A range of items can also be selected with the keyboard, using shift-arrow combinations. The keyboard cannot be used to pick out multiple separate list items. Only the mouse can do that.

A single row of a listbox can also be selected by typing a character. The <listbox> must first have the input focus. The single typed character is matched against the label attribute of the <listitem> tags in the <listbox>. If the label starts with the same character, there is a match. This system selects one row only and works as follows. The <listitems> are treated as a circular list of items, like a focus ring. The starting point is either the currently selected <listitem> or the first <listitem> if no selection yet exists. The list is scanned until a match for the character is found. This allows the user to cycle through the list multiple times.

The user cannot resize the columns of a multicolumn listbox, unless the application programmer enhances the listbox widget with extra event handlers. The same is true of hiding or collapsing columns.

13.2.12 AOM and XPCOM Interfaces

A <listbox> can be used for more than just display; it can be used to manage the data it contains. The need to insert, update, and delete that data, or to get and set it, means that robust interfaces are needed from the programming side.

The XBL definition for the <listbox> tag makes a number of properties and methods available to the JavaScript programmer. Many of these features mimic the actions of the DOM 0 and DOM 1 standards. Table 13.2 documents them. This table is drawn from the version 1.4 XBL binding named listbox.

Table 13.2 Properties and methods of the DOM object for <listbox>

Property or method	Description
accessible	The XPCOM accessibility interface for <listbox>
listBoxObject	The specialized boxObject for <listbox>
disableKeyNavigation	Turn alphabetic keyboard input on (true) or off (null)
timedSelect(listitem, millisec delay)	Select a single <listitem> with a pause that allows page layout (scrolling) to keep up, and the selection to be paced at user speed
selType	Same as attribute seltype
selectedIndex	Get or set the current selected item, starting from 0; returns -1 if multiple items selected
value	Current value of the sole selected item, or fails if more than one item is currently selected
currentItem	The <listitem> most recently selected
selectedCount	The number of <listitem>s selected
appendItem(label, value)	Add a <listitem> to the end of the <listbox>

Table 13.2 Properties and methods of the DOM object for <listbox> (Continued)

Property or method	Description
insertItemAt(index, label, value)	Add a <listitem> at position index
removeItemAt(index)	Remove the row at position index
timedSelect(listitem, millisec delay)	Select a single <listitem> with a pause that allows re-layout (scrolling) to keep up
addItemToSelection(listitem)	Add this DOM <listitem> to the currently selected items (and select it)
removeItemFromSelection(listitem)	Deselect this DOM <listitem>
toggleItemSelection(listitem)	Reverse the selection state for this <listitem>
selectItem(listitem)	Deselect everything and then select this sole <listitem>
selectItemRange(startItem, endItem)	Deselect everything and then select all <listitem>s including between these two items
selectAll()	Select all rows in the <listbox> except the header row
invertSelection()	Flip the select state of all rows in the <listbox>
clearSelection()	Deselect everything
getNextItem(listitem, offset)	Go offset <listitem>s forward from the supplied item and return the item found
getPreviousItem(listitem,offset)	Go offset <listitem>s backward from the supplied item and return the item found
getIndexOfItem(listitem)	Return the index of this <listitem> within the <listbox>
getItemAtIndex(index)	Return the <listitem> at index in the <listbox>
ensureIndexIsVisible(index)	Scroll the listbox contents until the <listitem> with this index is visible
ensureElementIsVisible(listitem)	Scroll the listbox contents until this <listitem> is visible
scrollToIndex(index)	Scroll the listbox content to the <listitem> with this index
getNumberOfVisibleRows()	Return the number of items currently visible
getIndexOfFirstVisibleRow()	Return the index of the <listitem> that currently appears at the top of the <listbox>
getRowCount()	Return the total number of rows; unreliable at this publication date

These properties and methods are listed here because <listbox> widgets generally benefit from scripting, and these interfaces are different from standard Web development experiences. These properties and methods should be used instead of the DOM 1 Node and Element interfaces, or else the internals of the <listbox> can become confused. In general, a basic understanding of XBL allows these properties and methods to be read straight out of the XBL binding for <listbox>. There is nothing new in this table; it is just reformatted XBL and comments.

One critical feature of this interface is the use of an index argument. This index refers to any viewable item in this listbox. The viewable items are those rectangular boxes of content inside the listbox that can be scrolled into view. The index does not refer to any list of tags in the listbox's construction. For a listbox, this difference is trivial, because each visible rectangle of content has exactly one <listitem> tag. Later we'll see that a similar index used with the <tree> tag matches nothing but the rectangles of content displayed.

In Chapters 7, 8, and 10, we noted that the <menupopup>, <scrollbox>, and <iframe> tags (amongst others) are special kinds of boxlike tags. They are special because they support additional processing on their box contents. These tags' DOM objects contain a boxObject property that can yield up a specialist interface. This specialist interface gives access to that additional content processing.

<listbox> is another example of such a specialist boxlike tag. It supports the nsIListBoxObject interface. That interface provides traversal and scrolling operations on the listbox contents. It can also be had from the component

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

One of the reasons that Table 13.2 is so big is because most (but not all) of the features of nsIListBoxObject are exported by the <listbox> XBL definition. They appear as the bottom third of Table 13.2. In object-oriented design pattern terms, the <listbox> XBL definition is a façade for this special interface.

In fact, most of the methods and properties in Table 13.2 match a published XPCOM interface. Interface nsIDOMXULSelectControlElement is also implemented by menu lists, radio groups, and tab controls. The nsIDOMXULMultiSelectControlElement is only implemented by the object that XBL creates for <listbox>.

That concludes the discussion of <listbox>. XUL trees, covered next, can do just about everything the <listbox> tag can do, and more.

13.3 TREES

If the <listbox> tag is a blend of <grid> and <menulist> concepts, then the <tree> tag is a blend of <listbox> and <iframe>. A <tree> widget is a vertically arranged, scrollable set of records like <listbox>. <tree> allows a

simple containment hierarchy to be imposed on the displayed rows so that rows are indented different amounts and decorated with graphical hints. When this hierarchy is not imposed, <tree> looks much like <listbox>.

<tree> gives the user room to control the display of the records presented. The user can collapse and expand subparts of the tree interactively. Columns can be reordered, hidden, and resized, and their contents can be sorted and selected.

<tree> gives the application programmer many data processing options. Sort and view features give the programmer direct control over data presentation. When integrated with overlays or templates, the tree widget provides a highly dynamic panel in which data can be managed. Trees can be extensively styled.

If whole-of-document widgets like <iframe> are ignored, then <tree> is the most complex widget that XUL provides.

13.3.1 Visual Appearance

To see a <tree> at work, look at any of these Classic Mozilla windows: The Preferences dialog (left panel); the Messenger window (two panels); the Manage Bookmarks window; the Download Manager. All these windows contain <tree> tags. In fact, there are dozens of trees used throughout the Mozilla application.

Figure 13.5 is a tree that shows the features that XUL trees support. This screenshot uses the Modern skin.

From the diagram, a <tree> appears as a set of columns, much like a <listbox>. Unlike a <listbox>, there is a dropdown menu under the icon in the top-right corner. This menu (not shown) is a set of checkboxes that can be used to hide or redisplay any column.

A <tree> has several kinds of special-purpose columns. Column A in Figure 13.5 is a primary column. A primary column shows the hierarchical organization of the rows in the tree. Looking at this column closely, there are four top-level rows: rows 1, 2, 7, and 9 (so this tree is a “forest” of trees). The second of these has a subtree that is revealed for display—it is open. The small downward-pointing triangle (a *twisty*) also indicates that the subtree row is open. That subtree has four child rows, and one of those rows has its own sub-

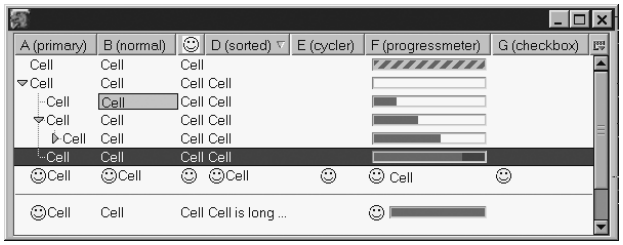


Fig. 13.5 <tree> example showing most features.

tree, which is also open. Finally, that second subtree has a single child row that again has a subtree. This final subtree is closed (the twisty icon points right), so the display doesn't show how many rows are in that final subtree. Twisties can be clicked open or closed by the user. The short lines between twisties and rows just indicate the level of the tree to which the current row belongs. Finally, the column A cells are indented to match their tree level.

The remaining columns in the diagram are less complex. Column B is an ordinary column. Column C is an ordinary column whose column header text has been replaced with an image. Column D is sortable: The column can be used to force the order of rows in the tree, breaking the normal tree structure. This sorting is indicated by the small arrow in the column header. Technology discussed in Chapter 14, Templates, needs to be added to a `<tree>` before sorting will work; in Figure 13.5, it is just present for the sake of completeness. Column E is a cycler. Such a column contains a clickable image only and has a special interaction with the user. Column F holds a `<progressmeter>` tag. Column G is designed to hold a checkbox, but that functionality is not finished and does not work in versions 1.4 and less.

The parentheses in the column names are not specially constructed; they are just part of the column name text. The scrollbars on the right of the tree appear and disappear as required, based on the amount of tree content. The scrollbar in Figure 13.6 is part of the `<tree>` tag, not some other part of the chrome window. Trees do not support horizontal scrollbars.

Figure 13.5 can also be examined row by row. The first interesting row is row 3. One cell of that row has an alternate background color and a border. Mozilla has a special styling system for trees, which is described under "Style Options" in this chapter. Row 6 is the currently selected row so the tree has the input focus. Row 7 shows that cell contents can have an image prepended to the cell content; this is not a `list-style-image` style. Row 7 also shows how an image replaces all the cell content in a cycler column and how a progress-meter column's cell content can be overridden with ordinary content. Row 8 (a single horizontal line) is a `<treeseparator>`; it acts as `<menuseparator>` does for menus. Row 9 shows that cell content is cropped if there isn't room to display it. If very little space is available in the cell, then not even ellipses will be displayed. Finally, the last row is completely empty. This is probably a bad idea in a real application, but it is technically possible.

A tree cell cannot contain arbitrary XUL content. It can only contain a line of text and the few variations noted previously. It cannot contain a `<box>`.

13.3.2 Construction

Figure 13.6 repeats Figure 13.5 with diagnostic styles turned on.

Little of the tree's internal structure is revealed with these styles. Only the column heading bears some resemblance to other widgets like buttons and labels. Obviously, `<tree>` is not based on a gridlike structure and is very different from `<listbox>`. Something unusual is going on.

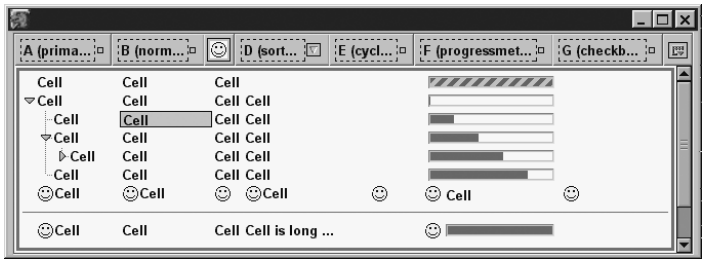


Fig. 13.6 `<tree>` example with diagnostic styles.

In fact, the content area of a `<tree>` is a little like an `<iframe>`. It is a rectangular area in the XUL document whose content is stored separate from the rest. In an `<iframe>`, the `<iframe>` content comes from a separate XUL document. In the `<tree>` case, the `<tree>` content has no separate XUL document, but it is still held apart from the other content.

It is not possible to style part of a tree using normal CSS2 styles. A special style system exists instead. The reason is that the individual cells and rows of a tree have frames that are not fully exposed to the styling system. This just happens to be the way `<tree>` is designed and implemented.

Figures 13.5 and 13.6 require over a hundred lines of XUL, so Figure 13.7 shows a simpler example with just two rows.

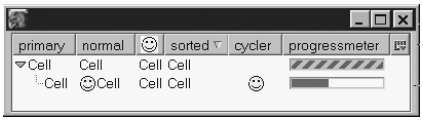


Fig. 13.7 Simple two-row `<tree>` example.

This tree is constructed in a XUL document as for all XUL content. The content fragment required for this tree is shown in Listing 13.3.

Listing 13.3 Basic construction of `<tree>` content.

```
<tree flex="1">
  <treecols>
    <treecol flex="1" id="A" label="primary" primary="true"/>
    <treecol flex="1" id="B" label="normal"/>
    <treecol flex="1" id="C" label="icon" class="treecol-image"
          src="face.png"/>
    <treecol flex="1" id="D" label="sorted" sortDirection="ascending"/>
    <treecol flex="1" id="E" label="cycler" cycler="true"/>
    <treecol flex="1" id="F" label="progressmeter" type="progressmeter"/>
  </treecols>
```

```
  <treechildren id="topchildren" flex="1">
    <treeitem container="true" open="true">

      <treerow>
        <treecell label="Cell"/>
        <treecell label="Cell"/>
        <treecell label="Cell"/>
        <treecell label="Cell"/>
        <treecell label="Cell"/>
        <treecell label="Cell" mode="undetermined"/>
      </treerow>

      <treechildren>
        <treeitem>
          <treerow>
            <treecell label="Cell"/>
            <treecell src="face.png" label="Cell"/>
            <treecell label="Cell"/>
            <treecell label="Cell"/>
            <treecell src="face.png" label="Cell"/>
            <treecell label="Cell" mode="normal" value="40"/>
          </treerow>
        </treeitem>
      </treechildren>

    </treeitem>
  </treechildren>
</tree>
```

The <tree> tag has a <treecols> child, in which the columns are defined, and a <treechildren> child. Column ids are very important for trees. The top-level <treechildren> tag is a little like <listbox>'s <listboxbody>, except that it is specified by the application programmer and can be nested inside other tags. A tree "item" is a whole subtree of the tree; therefore, a list item appears as a series of rows. This example code has a single top-level <treeitem> tree item. The container attribute says that this item is not just a row but also the top of the subtree. open says that the next level of the subtree is visible. If a second or third top-level <treeitem> were to appear, it would appear after the bottom </treeitem> tag. The sole top-level tree item has two parts: the row content and the subtree children. That is all it can hold. The subtree started with the second <treechildren> tag also has a single tree item, but this time it is not a subtree, it is just a row. If it were to have a second tree item, that would appear after the inner </treeitem> tag.

XBL definitions for <tree> are stored in tree.xml in toolkit.jar in the chrome.

There are other structural aspects to trees: RDF, sorts, views, builders, and templates. An overview of each is provided after the XUL tree tags are explained.

13.3.3 <tree>

The <tree> tag surrounds all of a tree's content. More than one tree can be specified in a given XUL document. The <tree> tag has the following special attributes:

```
seltype hidecolumnpicker enableColumnDrag disableKeyNavigation
```

- seltype set to multiple (the default) allows the user to select multiple items at once in the tree. Set to single, only one item at a time can be selected.
- hidecolumnpicker set to true collapses the <treecolpicker> tag in the top right corner of the tree.
- enableColumnDrag set to true allows a user to reshuffle the order of columns.
- disableKeyNavigation set to true prevents the user from selecting rows using alphabetic keystrokes.

The <tree> tag and many of the other <tree>-like tags also support RDF and templates. The attributes relevant to those features are discussed in Chapter 14, Templates.

A maximum display height for <tree> can be set with the standard box layout attribute height. <tree> does not support a rows attribute.

13.3.4 <treecols>

The <treecols> tag encloses the column definitions for a tree. This tag has no special attributes. It might be assigned an id if the tree is partially built from overlays. The <treecols> tag is a simple container tag. This tag must be the first child tag inside <tree>. It is not optional.

<treecols> can contain two types of tag: <treecol> and <splitter>.

The number of <treecol>s inside a <treecols> tag gives the number of columns in the tree. At least one <treecol> tag must be present. At most, one <treeecol> tag per tree can have primary="true" set.

A splitter represents a drag point that can be moved by the user. If a <splitter> tag is specified, it must appear between two <treecol> tags. If all possible <splitter> tags are supplied, <treecol> and <splitter> tags must alternate across the tree header. The net result of such a drag is that the columns on either side of the splitter are resized. The XBL definition for <tree> includes logic that supports such <splitter> tags and drag gestures.

Any <splitter> tags in a tree should be styled with class="tree-splitter" so that the tags have zero width. If this is not done, the column headings and columns may not line up. Because of the way Mozilla identifies the current tag under the mouse cursor, the <splitter> tag can be the current tag even when it has no visible area. It can still be dragged when of zero size.

13.3.5 <treecol>

The <treecol> tag defines a single column of a tree. It cannot contain any tags. Each <treecol> tag must have a unique id. This id is used internally by the Mozilla Platform. The column header for the tree is a <button> containing a <label> and an <image>, or just an <image> for a column header with the treecol-image class. Column headings can be styled with list-style-image if an additional image is required.

The attributes with special meaning to <treecol> are

```
label display crop src hideheader ignorecolumnpicker fixed
     sortDirection sortActive sortSeparators cycler primary
     properties
```

- label specifies the text to appear in the column header for that column. A <label> tag cannot be used as content.
- display specifies the text to appear in the column picker for that column.
- crop applies to the <label> content of the column.
- src replaces the column label with an image. The style class="treecol-image" must also be applied to the <treecol> if this is to work.
- hideheader="true" removes the button-like appearance of the column header. The space for the header still exists—the header is neither collapsed nor hidden. If this attribute is used, the label for the column should be removed as well. This attribute can be used on all headers and, when coordinated with the hidecolumnpicker attribute, can completely collapse the column header area.
- ignorecolumnpicker="true" prevents the dropdown menu from hiding or showing this column.
- fixed="true" prevents this column from feeling the effects of dragging any splitter next to it. In other words, this specific column cannot be resized unless the whole tree is resized. This is a consequence of the <splitter> tags in the tree header.
- If sortActive="true", then this column is sorted. sortDirection can be set to ascending, descending, or normal. If sortSeparators is set to true, then special sorting occurs that keeps rows between the <treeseparator> tags that they started between. Sorting only occurs automatically when RDF and templates are used—see Chapter 14, Templates.
- cycler="true" means that this column is a cycler column and will contain only an icon. That icon can be made button-like by adding onclick handlers. If a cycler column is clicked, the row under the mouse pointer is not selected.

- `primary="true"` means that this column id is the primary column for the tree and should show a hierarchical view of the tree items. This attribute can only be applied to one `<treecol>` tag per tree.
- The `properties` attribute supports Mozilla's special tree styling system. See "Style Options" in this chapter.

A column can be hidden or collapsed using standard XUL attributes.

13.3.6 `<treechildren>`

The `<treechildren>` tag is both the ultimate container tag for all of a tree's rows and the container tag for each subtree in the tree. It has no special attributes.

The `<treechildren>` tag can contain only `<treeitem>` and `<treeseparator>` tags.

This tag has two uses. A `<tree>` tags second child *must* be a `<treechildren>` tag. A `<treeitem>`'s optional second child tag can *only* be a `<treechildren>` tag. If used in the second way, then the `<treechildren>` tag should always have at least one `<treeitem>` content tag. If it does not, the twisty icon for that subtree will act strangely.

The `<treechildren>` tag is the tag used for style rules that exploit Mozilla's special tree styling system.

13.3.7 `<treeitem>`

The `<treeitem>` tag represents one horizontal item in a tree. An item is one of

1. A single row of cells without any subtree.
2. A single row of cells with a subtree that is displayed.
3. A single row of cells with a subtree that is hidden.

This tag can contain either a `<treerow>` tag (case 1) or a `<treerow>` tag followed by a `<treechildren>` tag (cases 2 and 3). `<treeitem>` cannot contain multiple `<treerow>` or `<treechildren>` tags.

The attributes with special meaning to `<treeitem>` are

```
container open properties
```

If `container` is set to `true`, then the item has a subtree and should contain a `<treechildren>` tag as its second content tag. If `open` is set to `true`, then the items' subtree is displayed (case 2). By default, neither attribute is set. `properties` is used by the style system that is described under "Style Options."

The `<treeitem>` tag also supports template-related attributes such as `uri`. See Chapter 14, Templates, for more on that.

13.3.8 <treeseparator>

The <treeseparator> tag draws a horizontal line across the tree. It can control sort behavior, as described in Chapter 14, Templates. This horizontal line is not indented, so <treeseparator> is only useful as a substitute for a top-level <treeitem> tag. It has one special attribute:

```
properties
```

This is used by the special style system described under "Style Options." <treeseparator> is meant to be a visual cue only.

13.3.9 <treerow>

The <treerow> tag holds the contents of a single row in the tree. It can contain only <treecell> tags. There should be <treecell> content tags equal to the number of columns. Specifying fewer <treecell> tags is also possible, but it is not very meaningful and is not recommended. Doing so reduces the number of cells visible in that row. The only attribute special to treerow is

```
properties
```

This attribute has the same use as it does for <treeitem>.

13.3.10 <treecell>

The <treecell> tag is responsible for the content of a single cell in a tree. In a primary column, it is not responsible for the indentation, twisty, or any connecting lines.

<treecell> cannot have any content, except for the special case of the column that holds <progressmeter> content. In that case, a single <progressmeter> tag is automatically added. <treecell> cannot contain a <label> tag.

The attributes with special meaning to <treecell> are

```
label src value mode allowevents properties
```

- label sets the displayed content for the cell. Text content cannot wrap lines as <label> can.
- src can be used to prepend an image to the content of the cell.
- mode can be set to normal or undetermined, provided the matching column has type="progressmeter". It provides the mode attribute for a <progressmeter> tag.
- value is the percent value of any mode="normal" progress meter that is present.
- If allowevents="true", then clicks that would normally select the row go through to the cell for handling. The row is not selected in this case.
- properties is used by the system described in "Style Options."

<treecell> also supports RDF template attributes like resource and ref. See Chapter 14, Templates, for these.

13.3.11 <treerows> and <treecolpicker>

These tags are used only inside XBL definitions. They are used automatically to construct the contents of the <tree> tag. <treerows> holds all the rows of the tree. It plays the same role for <tree> that <listboxbody> plays for <listbox>. <treecolpicker> holds the image and dropdown menu for the column picker. The dropdown menu is generated dynamically when the tree is first created. <treecolpicker> has an ordinal attribute that is set to a very high number. This ensures that it always appears to the right of the tree columns.

The <treecolpicker> tag can be styled as for any XUL tag. Its icon has class tree-columnpicker-icon.

There is no need to use these tags directly in a XUL document. The XBL code that implements <treecolpicker> is a useful guide for applications needing a similarly dynamic widget.

13.3.12 Nontags

<tree> was once called <outliner>, but no longer. The <outliner> tag does not exist any more. When <tree> was called <outliner>, <listbox> was called <tree>, but that <tree> was different from the contemporary <tree> tag. Beware of these ancient names in very old documentation that sometimes appears on the Mozilla Web site, in newsgroups, or in the bug database.

The <treecolgroup> tag is an old name for <treecols>. It lingers in a few Mozilla chrome files but should never be used. Use <treecols> instead.

The <treecolpicker> tag is part of the XBL definition for <tree>. It is meant for internal use only and shouldn't be specified in a XUL document.

The <treeindentation> and <treeicon> tags have no meaning as XUL tags. The <treehead>, <treecaption>, <treefoot>, and <treebody> tags have no meaning as XUL tags. All these tags are old and experimental at best. They are not supported.

13.3.13 RDF and Sorting

As for <listbox>, <tree> and its related tags can be connected to an RDF document. If this is done, the content of a <tree> derives from the content of the RDF document. Under such an arrangement, the data in a <tree> can then be sorted. See Chapter 14, Templates, for more detailed instructions.

13.3.14 User Interactions

Trees have all the user interactivity options that listboxes have, and more.

The most important user interactions that XUL trees support are more

about application semantics than they are about keystrokes or mouse gestures. When a tree displays hierarchical structure, it allows the user to participate in drill-down, summarizing, and classification actions. These tasks should be properly supported. Drilling down bears some further thought.

When the user clicks a twisty to reveal a subtree, that is a drill-down action. In such an action, the user is asking for more detail on a given subject. Data displayed in a hierarchy should always support this kind of exploration with data that is an answer to the user's request. The rows exposed should not be irrelevant: They must be about the parent row.

Studies have shown that users cannot handle drilling down many levels—they get lost and it is inefficient navigation. It is better to have a wide, shallow tree that scrolls a lot, than a very structured and deep tree whose subtrees easily fit the window.

The lower-level interactions that trees support closely match those of the listbox and are noted as follows.

Trees support keyboard and mouse navigation and have accessibility support. Navigation keys include the Tab, Arrow, Paging, Home, and End keys, and the spacebar. Mouse support includes clicks, key-click combinations, and the use of scroll wheels. Accessibility follows automatically because `<treecell>` tags always require labels.

Trees are members of the focus ring for the currently displayed page. If a `<tree>` does not have a currently selected row, then navigating into the tree from the previous member of the focus ring does not provide any visual feedback, but the tree still has the input focus. This behavior may be improved after version 1.4, using a workaround in Classic Mail & News that relies on styles.

The selection of tree items is a flexible matter. If only single-item selection is enabled, then selection is much the same as for a menu. If, however, `seltype` is set to `multiple`, then multiple list items can be picked. With the mouse, this is done by shift-clicking, to select a contiguous range of items, or by control-clicking, to pick out individual list items not necessarily next to each other. A range of items can also be selected with the keyboard, using shift-arrow combinations. The keyboard cannot be used to pick out multiple separate list items. Only the mouse can do that. If a `<treeitem>` that contains a subtree is selected, then only the row at the root of the subtree is selected, even if the subtree is collapsed.

A single row of a tree can be selected by typing a character, as for `<listbox>`. The `<tree>` must first have the input focus. The single typed character is matched against the `label` attribute of the `<treeitem>` tags in the `<tree>`. If the label starts with the same character, there is a match. This system selects one row only and works as follows. The `<treeitem>`s are treated as a circular list of items, like a focus ring. The starting point is either the currently selected `<treeitem>`, or the first `<treeitem>` if no selection yet exists. The tree is scanned until a match for the character is found. This allows the user to cycle through the tree multiple times.

If `enableColumnDrag` is set, then tree columns can be reordered using a mouse gesture. Just drag the column header across the face of the tree. The column picker icon cannot be moved.

If `<splitter>` tags are used between `<treecol>` tags, then these splitters can be used to resize the columns with a drag gesture on the splitter tag. The column picker icon cannot be resized.

The column picker, if it is not disabled or ignored, can be used to hide or show any of the columns of a tree. Hidden columns are persistent across Mozilla application sessions if `persist="hidden"` is set.

Finally, sorting is implemented with a simple mouse click on a column header.

13.3.15 AOM and XPCOM Interfaces

The scriptable features of XUL trees are quite complex. This topic provides a concept overview of these features and a detailed look at the interfaces that apply to simple trees. A simple tree is a tree that doesn't involve RDF or templates. All the examples of trees in this chapter are simple trees. More complex trees are covered in Chapter 14, Templates.

`<tree>` is an example of a specialist boxlike tag, just like `<listbox>`, `<iframe>`, and `<scrollbox>`. `<tree>` is a boxlike tag and so the DOM object for `<tree>` has a `boxObject` property. That property supports a tree-specific interface. That interface provides scrolling, navigation, selection, and data extraction methods for the tree content. Like `<listbox>`, the XBL definition for `<tree>` makes this special interface immediately available. This interface can also be had from the component and interface:

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

This interface is similar in many ways to the `nsIListBoxObject` interface. Unlike the `<listbox>` tag, few of the features of this interface are exported to the `<tree>` tag's XBL binding. That means you must work on the `nsITreeBoxObject` object directly. That object is exposed as the `treeBoxObject` property of the `<tree>` tag's DOM object.

Table 13.3 shows the precise control that this interface gives over the screen area taken up by the tree.

Table 13.3 Properties and methods of the nsITreeBoxObject interface

Property or method	Description
view	Any nsITreeView view associated with the tree
focussed	True if the <tree> has the focus
treeBody	The <treebody>'s DOM element

Table 13.3 Properties and methods of the nsITreeBoxObject interface (Continued)

Property or method	Description
selection	An nsITreeSelection object that understands which rows are currently selected
rowHeight	The height in pixels of a row (all rows are the same height)
getColumnIndex(id)	The ordinal number of the column with id="id"
getColumnId(index)	The id of the column with ordinal number index
getKeyColumnIndex()	The ordinal number of the primary column
getFirstVisibleRow()	The row index of the topmost visible row
getLastVisibleRow()	The row index of the bottommost visible row
getPageCount()	Total rows divided by the number of rows that fit the tree area; equals the number of pages of displayable rows
ensureRowIsVisible(index)	Scroll the tree content until the index'th row is visible
scrollToRow(index)	Scroll the tree content until the index'th row is at the top
scrollByLines(count)	Scroll down (>0) or up (<0) count lines; stop scrolling as soon as there is no more to scroll
scrollByPages(count)	Scroll down (>0) or up (<0) count number of pages; a page is the number of rows that fit inside the tree's area
invalidate() invalidateColumn(id) invalidateRow(index) invalidateCell(index, id) invalidatePrimaryCell(index) invalidateRange(index1, index2) invalidateScrollbar()	Tell Mozilla to redisplay (repaint) the stated part of the tree; index is a row index; id is a column id
getRowAt(x,y)	Return the index of the row under the given relative (x,y) coordinates, or return -1
getCellAt(x,y,r,c,type)	Return the row index, column id, and type of the cell at relative (x,y) coordinates; r, c, and type must be empty objects: {}; each object gains a value property that contains the returned data
getCoordsForCellItem(index, id, type, x, y, w, h)	Return the x-, y-, width-, and height- layout for the element with type held in the index'th row in column id; type may be "cell," "twisty," or "image." x, y, width, and height must be empty objects: {}; each object receives a value property
isCellCropped(index, id)	Return true if the index'th row in column id

Table 13.3 Properties and methods of the nsITreeBoxObject interface (Continued)

Property or method	Description
rowCountChanged(index, total)	Rows equal to total starting from the row at index have changed, so redisplay
beginBatchUpdate()	Tell the tree to stop re-laying out and repainting the tree after every little change
endUpdateBatch()	Tell the tree to catch up on changes that require layout or repainting
clearStyleAndImageCache()	Remove all style information in the tree in preparation for a theme change

XUL trees also have very flexible implementation options. Not only are there familiar interfaces, but there are some important design concepts to understand as well.

XUL trees are built around the design pattern called Model-View Controller but use different terms for each of these things. To recap, in this design pattern, the Model holds the data; the View displays the data; and the Controller coordinates the other two based on input from the outside world.

A XUL tree implements this design pattern with a graphical widget, seminal data, a builder, and a Mozilla view. A Mozilla view is a piece of code that provides an arrangement of the seminal data that is suitable for display. In MVC terms, the seminal data, assisted by the Mozilla view, makes up the MVC model, *not* the MVC view. The widget is part of the MVC view, which is completed by the builder. The builder can take the role of MVC controller as well. There is only one tree widget; it is shown in Figure 13.6. When constructing a tree, an application programmer has choices in each of the other three areas: the builder, the Mozilla view, and the seminal data.

Some of these new tree concepts require templates. In the following discussion, the parenthetical remark (Chapter 14, Templates) means that concept is discussed in detail in the next chapter.

13.3.15.1 Seminal Data

Seminal data are the data that the content of a tree comes from. Seminal data is not a technical term; it is just a descriptive term that avoids reusing other technical terms. The `<tree>` tag is always required for a XUL tree, but the data that make up the tree items, rows, and cells can come from one of three places: XUL, JavaScript, or RDF.

Listing 13.4 is an example of tree data specified in XUL. The content of the tree is stated literally and directly in the document containing the `<tree>` tag. This is a straightforward way to specify tree content. Even if overlays are used to contribute content from other documents, this is still a pure XUL solution.

Tree content can also be specified directly in JavaScript. There are two ways to do this. The first way is to use the DOM 1 interfaces to create DOM 1

Element objects with calls to `document.createElement()`. By modifying the DOM tree, plain XUL-based trees can be dynamically added to. This is no different from any other use of the DOM. Listing 13.4 is an example of adding a row to the tree in Listing 13.3. This is routine DOM 1 manipulation, with the new tags being created and added bottom-up.

Listing 13.4 DOM manipulation of a XUL tree.

```
var doc = document;
var tree = doc.getElementById("topchildren");
var item = doc.createElement("treeitem");
var row  = doc.createElement("treerow");

for (var i=1; i!=7; i++)  // there are six columns
{
  var cell = doc.createElement("treecell");
  cell.setAttribute("label","NewCell"+i);
  row.appendChild(cell);
}
item.appendChild(row);
tree.appendChild(item); // item, row and cells now appear
```

The other way to use JavaScript as seminal data for a tree is to create a custom view. Views are described shortly, but to look ahead, a set of JavaScript methods can be used to serve up all the tree's content.

Finally, seminal data can come from an RDF document. This is achieved using XUL templates and a little ordinary XUL content (Chapter 14, Templates).

13.3.15.2 Builders Builders are a somewhat confusing topic in XUL, mostly because they are obvious only when trees are used. When used with a tree, special cases distract from the core reason that builders exist. We first consider what a builder is in the ordinary case.

Any XUL tag starts life as a simple textual string. If it is a visual tag, like `<button>`, it must end up as pixels on a display. The Gecko styling, layout, and rendering engine inside Mozilla is responsible for that transformation. If the tag is relatively simple, like `<box>`, then the tag's information might be sent directly to that engine.

There are only a few simple XUL tags. `<button>`, for example, can end up as a collection of tags that might include `<label>` and `<image>`. The XBL definition for `<button>` generates these tags and sends the results to the display engine. So XBL processing is an extra preparatory step before the display engine gets something to work with.

Some XUL tags require preparation beyond what XBL can provide. `<menulist>` is an example. Some part of the platform must construct and destroy the popup menu of a `<menulist>` when it is used. XBL does not do that work. A piece of functionality that is built into the platform must do it.

Such a piece of functionality is called a builder, merely because it assembles and disassembles content that is to be displayed (or undisplayed).

Every XUL tag has a builder, at least conceptually, but in most cases the builder is trivial. In nearly all cases, the builder is invisible to applications and runs automatically. Only the most complex tags might have a builder sophisticated enough to be exposed to applications. `<listbox>` has a sophisticated builder, but it is invisible. Only `<tree>` and `<template>` tags have visible builders, but even those builders are visible only part of the time. Mozilla contains two different tree builders.

The XUL *content builder*, or *default tree builder*, is used to construct plain XUL trees and trees constructed via the DOM. No programming effort is required to use this builder. This builder creates a tree using a batch process, and the whole tree is created in one step. If DOM operations change the content of a plain XUL tree, the builder is not involved. Instead, the pieces of the already built tree are intelligent enough to absorb those changes directly.

This tree builder is an invisible builder. It has no XPCOM component or interface. It is not scriptable. It is given a name merely to separate it from the *template builder*, which does have some visibility. The content builder, or default tree builder is the bit of Mozilla that acts like a builder, when no specialist builder is present.

The XUL template builder (Chapter 14, Templates) is used to construct all template-driven content, including templated trees. It is chosen automatically when templates are used. It has a specialized version specifically for building trees, called the *tree builder*. The tree builder should really have a more descriptive name, like builder-for-special-combination-of-template-and-tree. The tree builder can construct a tree "lazily," which means that parts of the tree are left unbuilt (and undisplayed) until needed later on. The tree builder can also be accessed and controlled by the application programmer. To do its job, that builder may use the content builder. Alternately, it may do part of the building work itself and rely on a separate object for the rest of the work.

This builder also supports application-programmer-specified observers, which further assists the building work. It has a scriptable component:

```
@mozilla.org/xul/xul-tree-builder;1
```

The builder interfaces (Chapter 14, Templates) on this object are

```
nsIXULTemplateBuilder nsIXULTreeBuilder
```

The second interface is part of the customization process. If a builder exists for a given tree, then the builder property on the tree's DOM Element will contain that builder.

13.3.15.3 Views A builder might do all the work required to create a tree out of seminal data. Or, it could hand part of the work to a subcontractor who specializes in making the data ready to use. The builder would then be free to

spend most of its energy overseeing the process. Such a specialist subcontractor is in this case called a view. It provides a view of the seminal data, not a view of the graphical (GUI) result. In object-oriented terms, this approach is called delegation. Without the view, the builder is incomplete and can't do any work. Without the builder, the view is ready to use, but has no boss telling it where to do work.

A view is used by a builder, but it can also be used by an application programmer. In some cases, the view can also be created by a programmer. In all cases, the view created must have these interfaces:

```
nsITreeView nsITreeContentView
```

Table 13.4 shows the properties that the XBL definition of `<tree>` creates to support views.

Table 13.4 JavaScript <tree> properties that are the result of views

Property name	Related to other properties?	Contents
treeBoxObject.view	Yes	View object exposing nsITreeView
view	Yes	View object exposing nsITreeView
contentView	Yes	View object exposing nsITreeContentView
builderView	Yes	View object exposing nsIXULTreeBuilder

If a view is replaced with another, in theory all these properties should be updated. In practice, it is enough to update the `treeBoxObject.view` property or the `view` property and avoid using the other properties afterwards. Table 13.5 shows the interface that such a view provides.

Table 13.5 Properties and methods of TreeView interfaces

Property or method	Useable in XUL content builder?	Description
nsITreeContentView		
root		Points to the <tree> tag's DOM object
getItemAtIndex(index)	✓	Returns the <treerow> at the index'th visible row in the tree; rows count if they can be scrolled into view, but not if they require a hidden subtree to be revealed; counts from 0
getIndexOfItem(treerow)	✓	Returns the index position of the <treerow> in the tree

Table 13.5 Properties and methods of TreeView interfaces (Continued)

Property or method	Useable in XUL content builder?	Description
nsITreeView		
canDropBeforeAfter(index)	✓	True if a dropped item can be inserted before or after this row
canDropOn(index)	✓	True if the given row can be a drop site for a drag-drop operation
cycleCell(index, id)		Fires when a cell (in row index, column id) in a cycle="true" column is clicked
cycleHeader(id, element)		Fires when the element tag in the column with this id is clicked
drop(index, where)		Fires when a dragged row is dropped; index is the row, where indicates the drop target and is 0, 1, or 2
getCellProperties(index, column_index, array)	✓	Fills the nsISupportsAarray with the values found in the cell's properties XML attribute and returns the array
getCellText(index, id)	✓	Returns the label= value in the cell at row index and column id
getCellValue(index, id)	✓	Returns the value= value in the cell at row index and column id
getColumnProperties(index, column-id, array)	✓	Fills the nsISupportsAarray with the values found in the column's properties XML attribute
getImageSrc(index, id)	✓	Returns the URL for any image prefixed to the cell with row index and column id
getLevel(index)	✓	Returns the depth of this row in the tree
getParentIndex(index)	✓	Returns this row's parent row index, or -1 if there is no parent
getProgressMode(index,id)	✓	Returns the type of <progressmeter> in the cell with row index and column id (returns 1, 2, or 3)
getRowProperties(index, array)	✓	Fills the nsISupportsAarray with the values found in the row's properties XML attribute and returns the array
hasNextSibling(index, start_index)	✓	True if the first sibling of this row after start_index
isContainer(index)	✓	True if the row has container="true" (has a subtree of zero or more elements)

Table 13.5 Properties and methods of TreeView interfaces (Continued)

Property or method	Useable in XUL content builder?	Description
isContainerEmpty(index)	✓	True if the row has container="true" and zero child nodes of <treechildren>
isContainerOpen(index)	✓	True if the row has open="true" and container="true"
isEditable(index, id)	Always false	Returns true if the cell at row index and column id is editable
isSeparator(index)	✓	True if the row is a <treeseparator>
isSorted()	Always false	True if any column in the row is sorted
performAction(command) performActionOnRow(command, index) performActionOnCell(command, index, column id)		Send the given command to the whole tree, to the row alone, or to a single cell
rowCount	✓	Reports the total rows in the tree
selection	✓	Returns an nsITreeSelection object containing details of the current selection
selectionChanged()		Fires when the selected row(s) in the tree changes
setCellText(index, id, value)		Sets the cell text at row index and column id to value
setTree(nsITreeBoxObject)		Used during initialization—avoid
toggleOpenState(index)	✓	Fires when a subtree container is opened or closed; can be called direct

Mozilla has half a dozen existing views written in C/C++, and about a dozen views written in JavaScript. One specific view belongs to the XUL content builder. It is called simply the "tree content view" and is the view that gives access to trees based on plain XUL content. This simple view is not a full XPCOM component; it is just a subpart of the XUL content builder. It does, however, support the preceding interfaces properly.

When the XUL content builder builds a tree, an object with this interface is attached to the view property of the `<tree>`'s DOM object. This view is available to the application programmer. It is used by the XUL content builder during tree construction, and it can be used by the application programmer after the tree is displayed.

There is one restriction to use of this view. It is a read-only system. The `isEditable()` view method reports back false, which means that the `set-`

`CellText()` view method does nothing. The methods that "fire" can only be called internally by the tree system, not by the application programmer. Also, because of the way this system is hooked up to the tree, the methods of this interface can't be replaced with user-defined ones. All that can be done with this view interface is extract information about the tree.

If a view is to be read-write, or if a view is to be created by the application programmer, then the tree must be the base tag of a template (Chapter 14, Templates).

In addition to the tree content view, many application-specific XPCOM components have `nsITreeView` interfaces and can be used as ready-to-go views. They, however, are highly specific components. Using these components requires extensive study of existing applications such as the Messenger. At version 1.4, the components with tree views are

```
@mozilla.org/addressbook/abview;1
@mozilla.org/filepicker/fileview;1
@mozilla.org/inspector/dom-view;1
@mozilla.org/messenger/msgdbview;1?type=quicksearch
@mozilla.org/messenger/msgdbview;1?type=search
@mozilla.org/messenger/msgdbview;1?type=threaded
@mozilla.org/messenger/msgdbview;1?type=threadswithunread
@mozilla.org/messenger/msgdbview;1?type=watchedthreadswithunread
@mozilla.org/messenger/server;1?type=nntp
@mozilla.org/network/proxy_autoconfig;1
@mozilla.org/xul/xul-tree-builder;1
```

To use any of these components except for the last (the default), it is recommended that their existing uses within the chrome files be studied carefully first.

The Classic Mozilla chrome also contains many pure JavaScript implementations of `nsITreeView`, whose implementation can be casually studied. The Navigator View | Page Info functionality has five views (in `pageInfo.js` in `comm.jar` in the chrome) and is the easiest to understand—see also `pageInfo.xul`. The DOM Inspector has two (in `jsObjectView.js` and `stylesheets.js`), the XUL `<textbox type="autocomplete">` functionality has one (in `autocomplete.xml`), and the Navigator `about:config` URL functionality has one (in `config.js`). The JavaScript Debugger and several other tools such as the Component Viewer implement JavaScript-based views as well.

Several of these JavaScript applications have created reusable prototype objects for custom views. These objects attempt to reduce the work required to create a view.

13.4 STYLE OPTIONS

Mozilla has a special styling system for `<trees>`. `<listbox>`, on the other hand, is mundane.

13.4.1 `<listbox>`

The listbox system has no Mozilla extensions unique to the CSS2 style system. It does, however, have an extensive set of style rules and id'ed tags to which styles can be applied. These rules appear in `xul.css` in `toolkit.jar` and in `listbox.css` in the global skin (e.g., in `classic.jar`). All those files are in the chrome.

13.4.2 `<tree>`

XUL has some unique and specific style functionality. This functionality is used for trees, which are styled differently than all other tags. All the body of a given tree, specified as content of the `<treechildren>` tag, can be styled directly from a `treechildren` selector. A style extension makes this possible.

Styling of trees is done using new pseudo-classes. Here is an example style, based on a tree that represents a company organization chart. If a staff member has been hired recently, then his or her entry is yellow:

```
treechildren:-moz-tree-row(hired)
  { background-color : yellow };
```

The `-moz-tree-row` pseudo-class identifies what is to be styled, in this case the rows of the tree. This selector is passed a list of zero or more parameters. Each of these parameters is a text string. Such a text string appears in some content tag of the tree as an argument to the `properties` keyword. For example,

```
<row properties="hired,causeNewDept,dateJune">...</row>
```

Any and all tags within a tree's body that have a property of `hired` are styled according to the given rule, so that includes the example `<row>` tag. If the rule appeared thus

```
treechildren:-moz-tree-row(hired,dateJuly)
  { background-color : yellow };
```

then the example row would not be styled because it does not contain both properties listed in the style rule.

Three pieces of information are required to make a style built with this system work:

1. The right pseudo-class name needs to be chosen.
2. A suitable property name needs to be decided.
3. The style properties available for the pseudo-class need to be reviewed.

Each of these items is covered in turn here.

13.4.2.1 Tree Pseudo-Classes

Table 13.6 lists the tree pseudo-selectors that are Mozilla extensions to the CSS2 standard.

Table 13.6 CSS2 pseudo-class extensions for XUL trees

Pseudo-selector name	Matching part of the displayed tree
:-moz-tree-row	A whole row, but without leading indentation
:-moz-tree-cell	One cell in a row
:-moz-tree-column	A whole column
:-moz-tree-cell-text	Text within a cell
:-moz-tree-twisty	The icon (twisty) clicked on that controls subtree expansion
:-moz-tree-indentation	Blank space to the left of an indented row
:-moz-tree-line	The small lines connecting parent, child, and sibling rows in the primary column
:-moz-tree-image	An image that prefixes a cell's contents
:-moz-tree-separator	<treeseparator>
:-moz-tree-drop-feedback	The line that appears between rows when dragging a row around
:-moz-tree-progressmeter	A cell whose column is type="progressmeter"

13.4.2.2 Built-in Property Names CSS2 uses keywords rather than literal strings for most purposes. It's important to remember that special tree-styling properties are just arbitrary strings of text that the application developer makes up. They are application-specific and have no meaning to the style system.

Some of the property-naming work is done for you. Special property names are automatically applied to a tree's contents when it is created and when the user interacts with it. These names still have no meaning to the style system. They are meaningful only in terms of the structural arrangement of a tree, and for use in custom pseudo-selectors. These names are automatically added to properties lists by Mozilla and can be selected just like user-defined properties. Table 13.7 lists them.

Table 13.7 Style pseudo-selector automatic properties for XUL trees

Property string	Meaning
container	The thing to style is part of an internal node.
leaf	The thing to style is part of a leaf node.
open	The thing to style is part of an internal node, and that node is uncollapsed so that any subtree contents are showing.

Table 13.7 Style pseudo-selector automatic properties for XUL trees (Continued)

Property string	Meaning
closed	The thing to style is part of an internal node, and that node is collapsed so that any subtree contents are hidden.
selected	The thing to style is part of a selected row.
current	The thing to style is part of the currently selected row.
focus	The tree is the currently focused document element.
sorted	The tree rows are sorted.
primary	The thing to style is part of the primary tree column.
progressmeter	The thing to style is part of a <treecol type="progressmeter">.
progressNormal	The thing to style is a part of a progress meter that reports progress as it occurs.
progressUndetermined	The thing to style is part of a progress meter that only reports when it's underway or finished.
progressNone	The thing to style is part of a progress meter that doesn't report progress.
dragSession	The user is dragging a tree element with the mouse.
dropOn	The user's dragged object is over the thing to style.
dropBefore	The user's dragged object is just above the row that the thing to style is in.
dropAfter	The user's dragged object is just below the row that the thing to style is in.

Recall that simple tree structures are built from internal nodes, which contain other nodes, and leaf nodes, which contain data. XUL tree nodes are either leaf nodes or internal nodes, but not both.

There is a second group of automatically available properties. All ids of all tree columns are available as properties. You should therefore ensure that those ids are legal CSS2 names.

Regardless of whether automatically available properties are used, you are always free to make up new property names.

13.4.2.3 Matching CSS2 Properties The third aspect of this custom styling system is quite tricky. Each of the pseudo-classes supports only a small number of the CSS2 style properties. If you choose a property that isn't supported, then nothing will happen. Table 13.8 sketches out which CSS2 properties are available for each pseudo-class.

Table 13.8 CSS2 properties supported by new pseudo-selectors

Pseudo-selector	Tag to use for properties attribute	Types of CSS2 sstyle property supported
:-moz-tree-row	<treerow>	Backgrounds, borders, margins, outlines, padding, display, -moz-appearance
:-moz-tree-cell	<treecell>	Backgrounds, borders, margins, outlines, padding, visibility
:-moz-tree-column	<treecol>	Margins, text styles, visibility
:-moz-tree-cell-text	<treecell>	Foreground color, fonts, visibility
:-moz-tree-twisty	<treecell>	Margins, padding, borders, display, -moz-appearance, list styles, positioning
:-moz-tree-indentation	<treeitem>	Positioning
:-moz-tree-line	<treeitem>	Borders, visibility
:-moz-tree-image	<treeitem>, <treecell>	List styles, margins, positioning
:-moz-tree-separator	<treeseparator>	Display, borders, -moz-appearance
:-moz-tree-drop-feedback	<treerow>	Margins, visibility
:-moz-tree-progressmeter	<treecell>	Foreground color, margins

Putting together Tables 13.6, 13.7, and 13.8 yields the following example, which is entirely constructed out of names that Mozilla is aware of:

```
treechildren:-moz-tree-cell(leaf,focus)
  { background-color : red; }
```

This says that any row in the tree that is a leaf row and that has the current input focus will have its cell background changed to red. Because background styles are supported for tree cells, this style rule both is sensibly constructed and will have the desired effect. Compare that with the earlier examples of this system which use custom styles consisting of known targets and known pseudo-selectors but use application-specific property strings.

13.4.3 Native Theme Support

As of version 1.4, trees do not have native theme support on Microsoft Windows XP.

Where native themes are supported, the `-moz-appearance` style property can be set to these values:

```
listbox listitem treeview treeitem treetwity treetwistyopen treeline
     treeheader treeheadercell treeheadersortarrow
```

13.5 HANDS ON: NOTETAKER: THE KEYWORDS PANEL

This "Hands On" session is about using standard, scripted XUL to master <listbox> and <tree>. First, we'll build a static page out of pure XUL, and then we'll enhance it to include scripting effects. We'll also experiment a little.

At last it's time to complete the layout and XUL content of the NoteTaker dialog box. Until now the Keywords panel of the displayed <tabbox> has contained only a placeholder. We'll fix that by adding a listbox, a tree, and some other form elements.

The Keywords panel allows the user to add keywords to, and delete keywords from, the current note. It also lists the current keywords. Finally, it displays keywords related to the current keyword, which provides a memory jogger to the user.

To design this pane, we must go back to Chapter 2, XUL Layout, and start with a rough diagram and then work on layout, static content, form elements, and so on. Rather than repeat that process here, we'll just summarize the important results.

- A <textbox> will allow the user to enter a keyword.
- A <listbox> will display the current set of keywords.
- An Add <button> will copy the <textbox> contents into the <listbox> as a new item.
- A Delete <button> will remove the <textbox> contents from the list box if it already exists.
- Clicking on a <listbox> item copies it to the <textbox>.
- A <tree> will display the keywords related to the keywords in the <listbox>.

Both the <listbox> and <tree> tags will have dynamically changing content. In this chapter, we'll implement those with JavaScript, the DOM, and tree views. For now, the related keywords we'll use will come from a small, fixed set. In the next chapter, we'll replace part of this system with a better solution based on templates. We're not going to use the RDF model designed in the last chapter. We'll do that in the next chapter.

Altogether, this result will be a dialog box as shown in Figure 13.8.

13.5.1 Laying Out <listbox> and <tree>

Without further ado, the structure of the Keywords panels is shown in Listing 13.5.

Listing 13.5 New panel content for the NoteTaker Edit dialog box.

```
<tabpanel>
  <vbox>
    <hbox>
```

```
      <vbox>
        <description value="Enter Keyword:"/>
        <textbox id="dialog.keyword"/>
        <hbox>
          <button id="dialog.add" label="Add"/>
          <button id="dialog.delete" label="Delete"/>
        </hbox>
      </vbox>
      <vbox>
        <description value="Currently Assigned:"/>
        <listbox/>
      </vbox>
    </hbox>
    <description value="Related:"/>
    <tree/>
  </vbox>
</tabpanel>
```

The panel is made of two boxes stacked vertically. The top box has a left and right half. In this listing the `<listbox>` and `<tree>` tags are not filled in, for brevity. The `<listbox>` content is shown in Listing 13.6.

Listing 13.6 Static `<listbox>` content for current NoteTaker keywords.

```
<listbox id="dialog.keywords" rows="3">
  <listitem label="checkpointed"/>
  <listitem label="reviewed"/>
  <listitem label="fun"/>
  <listitem label="visual"/>
</listbox>
```

In the completed version of NoteTaker, these keywords will come from an RDF file that will be composed from user input. Here we're starting out with some fixed keywords. Similarly for the `<tree>` tag, we start with some fixed related keywords. The content of the `<tree>` tag appears in Listing 13.7.

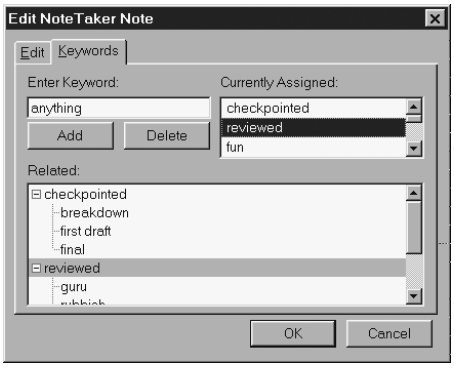


Fig. 13.8 NPA diagram showing `<list>` and `<tree>` technology.

Listing 13.7 Static <tree> content for related NoteTaker keywords.

```
<tree id="dialog.related" hidecolumnpicker="true" seltype="single"
          flex="1">
  <treecols>
    <treecol id="tree.all" hideheader="true" flex="1" primary="true"/>
  </treecols>
  <treechildren flex="1">
    <treeitem container="true" open="true">
      <treerow> <treecell label="checkpointed"/> </treerow>
      <treechildren>
        <treeitem>
          <treerow> <treecell label="breakdown"/> </treerow>
        </treeitem>
        <treeitem>
          <treerow> <treecell label="first draft"/> </treerow>
        </treeitem>
        <treeitem>
          <treerow> <treecell label="final"/> </treerow>
        </treeitem>
      </treechildren>
    </treeitem>
    <treeitem container="true" open="true">
      <treerow> <treecell label="reviewed"/> </treerow>
      <treechildren>
        <treeitem>
          <treerow> <treecell label="guru"/> </treerow>
        </treeitem>
        <treeitem>
          <treerow> <treecell label="rubbish"/> </treerow>
        </treeitem>
      </treechildren>
    </treeitem>
    <treeitem container="true" open="true">
      <treerow> <treecell label="fun"/> </treerow>
      <treechildren>
        <treeitem>
          <treerow> <treecell label="cool"/> </treerow>
        </treeitem>
      </treechildren>
    </treeitem>
  </treechildren>
</tree>
```

The keywords in this tree are displayed as a hierarchy, but it's not implied that child keys are subtopics of parent keys. That is a tempting way to think of them. They're just related concepts. A Web page could receive a note with the "guru" keyword, but not the "reviewed" keyword if the user decided that "guru" meant the page author was well-known, rather than that the page contained well-thought-out information.

A hierarchical display such as the <tree> tag lends itself well to hierarchically broken-down material, but it can be applied to other problems as well.

We're using it to show relationships that form a simple network. Instead of viewing the network as a whole, we're discovering parts of it from a set of starting points. That's not unlike the RDF queries in Chapter 14, Templates, but here it's familiar XUL and JavaScript technology.

13.5.2 Systematic Use of Event Handlers

Here is how this dialog box will work. The top-left part of this box is where the user enters new keywords. Clicking Add puts the typed-in keyword into the list at top right; clicking Delete removes it from the list. Clicking an item in the list copies that item to the textbox. If a list item is selected, then the matching item in the tree is selected and scrolled to. If an item in the tree is clicked, it is copied to the textbox.

All these actions could be implemented as commands. Some of these actions, however, are fairly trivial. It can be overkill to make every fragment of script a command. None of these actions sound like formal "transactions," "instructions," or "operations." They're just GUI tweaks. We'll use plain event handlers to implement them.

We could add these event handlers to individual `<listbox>` and `<tree>` content tags (`onclick= ...`), but we won't. Tree views also support custom actions, which are like localized commands (see the tree view methods prefixed with `performAction`). We're not doing that either. For our plain events, we'll need:

- `onclick` on the Add `<button>`
- `onclick` on the Delete `<button>`
- `onselect` on the `<listbox>`
- `onselect` on the `<tree>`

Rather than use XUL syntax, we'll choose a more interesting option. We'll use the DOM 2 Events "`EventTarget`" interface, in particular, the `addEventListener()` method. This method is available on every DOM Element object, which means most XUL tags. Using just scripts, we'll install all the event handlers when the Edit dialog box first loads. Listing 13.8 shows how this is set up.

Listing 13.8 Event handler installation for NoteTaker Keywords panel.

```
var ids = {};

function init_handlers()
{
  var handlers = [
    // id                 event    handler function
    ["dialog.add",       "click",  add_click],
    ["dialog.delete",    "click",  delete_click],
    ["dialog.keywords", "select", keywords_select],
    ["dialog.related",  "select", related_select]
  ];
```

```
  for (var i = 0; i < handlers.length; i++)
  {
    var obj = document.getElementById(handlers[i][0]);
    obj.addEventListener(handlers[i][1], handlers[i][2], false);
    ids[handlers[i][0]] = obj;
  }
  // also spot this final tag
  ids["dialog.keyword"] = document.getElementById("dialog.keyword");
}

window.addEventListener("load",init_handlers, true);
```

The `init_handlers()` function is installed as a handler that runs when the document is first loaded. When it runs, it installs five more handlers, using the tag ids, event names, and functions supplied in the handlers array. Those handler functions all accept a single argument, which is an Event object. In the case of this simple pane, each handler is used in only one place, so the Event object is not that useful. The function also stores in the `ids` object the DOM objects for each handler. This is just a performance optimization that saves retrieving those objects over and over later on in the handlers.

Each of these handlers is described in turn. The listbox and tree widgets are more complex than a simple button, so we'll need to roll our sleeves up a bit—there's plenty of code. Other programming environments have their headers, libraries, and modules; Mozilla has documentation on XBL, XPIDL, DOM, and this book. We'll need to use all that documentation to do a professional job.

Listing 13.9 shows the `add_click()` handler's code:

Listing 13.9 `add_click()` handler for NoteTaker keywords.

```
function add_click(ev)
{
  var listbox = ids["dialog.keywords"];
  var textbox = ids["dialog.keyword"];

  // getRowCount() workaround
  var items = listbox.childNodes.length;

  if ( textbox.value.replace(/^ *$/,"") == "" )
    return;     // don't add pure whitespace

  for (var i = 0; i < items; i++)
  {
    if ( listbox.getItemAtIndex(i).label == textbox.value )
      return;   // already exists
  }

  listbox.appendItem(textbox.value, textbox.value);
  listbox.scrollToIndex(items > 1 ? items - 2 : items);
}
```

This function adds the typed keyword to the listbox of existing keywords. It looks through the <listbox> items to see if the new item is already there, and if not, that item is added and the listbox scrolled to it so that the user can see it.

Most of the method calls in this function come from Table 13.2, but we also had to peek at the XBL definition for <listitem> and <textbox> to find the label and value properties.

Nothing is perfect, and as this book goes to press, the <listbox> getRowCount() XBL method has a bug, which will probably be fixed by the time you read this. That function returns the total number of existing list items (when it works properly). As a workaround, we go underneath the AOM widget level, which means using basic XML methods from the DOM 1 Core. The third line of the code returns a DOM 1 Core NodeList object, whose length property is the number of direct children of the <listbox> tag. That works for us because we know there's no <listcols> tag in our listbox.

Listing 13.10 shows the code for the delete_click() handler.

Listing 13.10 delete_click() handler for NoteTaker keywords.

```
function delete_click(ev)
{
  var listbox = ids["dialog.keywords"];
  var textbox = ids["dialog.keyword"];
  var items = listbox.childNodes.length;

  for (var i = 0; i < items; i++)
    if ( listbox.getItemAtIndex(i).label == textbox.value )
    {
      listbox.removeItemAt(i);
      return;
    }
}
```

This handler is a very minor variation on the add_click() handler. It deletes the textbox item from the listbox.

Listing 13.11 shows the code for the keywords_select() handler.

Listing 13.11 keywords_click() handler for NoteTaker keywords.

```
function keywords_select(ev)
{
  var listbox = ids["dialog.keywords"];
  var textbox = ids["dialog.keyword"];
  var tree    = ids["dialog.related"];
  var items   = document.getElementsByTagName('treecell');
  var item = null, selected = null;

  try { listbox.currentItem.label; }
  catch (e) { return; }

  textbox.value = listbox.currentItem.label;
```

```
  var items = document.getElementsByTagName('treecell');
  for (var i = 0; i < items.length; i++)
  {
    if (items.item(i).getAttribute("label") == textbox.value)
    {
      item = items.item(i).parentNode.parentNode;
      break;
    }
  }

  if ( item )
  {
    selected = item;
    if ( tree.view.getIndexOfItem(item) == -1 )
    {
      while (item.tagName != "tree")
      {
        if (item.getAttribute("container") != "")
          item.setAttribute("open","true");
        item = item.parentNode.parentNode;
      }
    }
    // tree.currentIndex = tree.view.getIndexOfItem(selected);
    // only supplies the focus,not the selection.
    i = tree.view.getIndexOfItem(selected);
    tree.treeBoxObject.selection.select(i);
    tree.treeBoxObject.ensureRowIsVisible(i);
  }
}
```

This function copies the label of the currently selected listbox item to the textbox. That takes one line of code. The rest of the function seeks and highlights the same keyword in the tree, if it exists. The tree view interfaces described in Table 13.4 work only on the currently rendered rows of the tree (including rows hidden by the surrounding scrollbox). If a keyword isn't displayed, the view interfaces won't find it. So we must use the DOM to search through the tree. After we find a matched item, we can work through the tree view to manipulate the display of the tree.

When grabbing the selected keyword, we have a second awkward problem with `<listbox>`. The `currentItem` XBL property isn't maintained correctly; specifically it is poorly created when the listbox has no currently selected elements. Although this is not obvious, it will spew errors to the console if we don't do something. These errors occur because our `onselect` handler also happens to be invoked when the listbox is scrolled. The `try {}` block catches this special case and aborts the handler, since there's nothing to do in that case.

To find the keyword in the tree, we use the very heavyweight DOM 2 Core method `getElementsByTagName()`. It searches the whole DOM for a given tag and returns all instances as a collection. We happen to know the only

<treecell> tags in the document are in the required tree, so it is safe and convenient to use this method. The actual keyword is stored in a <treecell> inside a <treerow> inside the <treeitem> tags returned. So we use the DOM 1 Core parentNode attribute to reach up to the <treeitem> tag, which we'll need to manipulate the visual appearance of the tree.

If we find a keyword, we need to expose the item displaying it in the tree. We start at the <treeitem> and reach up the tree, opening any containers we find by marking the <treeitem> tags with open="true". Again this is done with the DOM 1 Core. After the treeitem with the keyword is displayed, it will be visible to the tree view interface.

To finish up, we select the row of the tree with the keyword. The user will see this and can then easily see other keywords that might be relevant, both above and to a single level below in the hierarchy. To do this we use the tree view interface, which is exposed by the XBL property tree.view. We could have just as easily (but slightly more verbosely) acquired the view by using the AOM tree.treeBoxObject property, but we would've also had to use XPCOM's QueryInterface() method to extract the correct interface on that object. So we've done it the quick way.

We can't use the tree.currentIndex XBL property to select the tree row because that property records only the row that's focused, not the row that's highlighted. If you experiment with it, note the dotted line that appears around the correct row, indicating that the focus is present. Instead we use the AOM treeBoxObject.selection object, which implements the interface nsITreeSelection. The simple method select() highlights the row in question. Finally, we go back to the AOM treeBoxObject to scroll the tree view so that the selected row is not clipped by the scroll box that surrounds the tree.

Listing 13.12 shows the code for the related_select() handler.

Listing 13.12 related_select() handler for NoteTaker keywords.

```
function related_select(ev)
{
  var textbox = ids["dialog.keyword"];
  var tree    = ids["dialog.related"];

  textbox.value = tree.view.getCellText(tree.currentIndex, "tree.all");
}
```

After keywords_select(), related_select() is a very simple job. It picks out the currently selected keyword from the tree view and copies its value to the textbox. No DOM operations are used at all.

That concludes the event handler logic for the Keywords panel. Some commands, like notetaker-save and notetaker-load, need an update to accommodate this new pane. We'll put off doing that, because we're going to change everything to RDF in the next chapter.

13.5.3 Data-Driven Listboxes and Trees

We have now created the dialog box so that most of its trivial interactivity is in place, but the initial keywords are restricted to a statically defined set of XUL tags. Our ultimate solution is to feed content into these widgets using RDF, but as a brief experiment, we'll feed content to these widgets from plain JavaScript.

A sufficient reason for this experiment is the problem of related keywords. If A is related to B, then surely B is also related to A. Even though hierarchical XML is not so good at handling these two-way relationships, we'd like the tree widget to show related keywords in both directions. The tree display will still be organized hierarchically (that's all it can do), but our custom processing will feed a better set of information to the widget.

To feed content to `<listbox>`, the only option we've encountered so far is to use the raw DOM operations. This is sometimes called a dynamic listbox. To feed content to `<tree>`, we can use a tree view. Because we'll create that *view*, it is a *custom tree view*, or *custom view*.

There are several ways to create these things. For example, existing JavaScript libraries inside the `cview` and DOM Inspector tools provide JavaScript objects that are designed to make custom tree views easier. Also some treelike data, such as IMAP and SNMP data, place performance or functionality restrictions on how that data can be accessed. Here we'll stick to a from-scratch, basic approach.

In terms of the Model-View Controller design pattern, we'll implement the MVC model as a simple JavaScript data structure for both `<tree>` and `<listbox>`. The View will be built on top of (a) Mozilla's fundamental layout system, (b) the DOM hierarchy, and (c) the specialist box objects for `<listbox>` and `<tree>`. In the `<listbox>` case, the standard DOM interfaces are the meat and drink that our MVC view will be based on. In the `<tree>` case, we only need to create a special view object to implement the MVC view. Finally, the MVC in the `<listbox>` case is up to us to write, but in the `<tree>` case, it's the tree's existing builder, and so we don't need to do anything there.

Before exploring this functionality, we'll turn off the event handlers created previously. They're not part of this experiment:

```
// window.addEventListener("load",init_handlers, true)
```

We also need to remove the existing, static content for the `<listbox>` and `<tree>` tags in the XUL document. Those tags will be reduced to these XUL fragments:

```
<listbox id="dialog.keywords" rows="3"/>

<tree id="dialog.related" hidecolumnpicker="true" seltype="single"
      flex="1">
  <treecols>
```

```
        <treecol id="tree.all" hideheader="true" flex="1" primary="true"/
         >
      </treecols>
      <treechildren flex="1"/>
    </tree>
```

As usual, we have some initialization code to set everything up. Listing 13.13 shows this code.

Listing 13.13 Setup for data-driven NoteTaker keyword widgets.

```
var listdata = [ "checkpointed", "reviewed", "fun", "visual" ];

var treedata = [
  [ "checkpointed", "breakdown" ],
  [ "checkpointed", "first draft" ],
  [ "checkpointed", "final" ],
  [ "reviewed", "guru" ],
  [ "reviewed", "rubbish" ],
  [ "fun", "cool" ],
  [ "guru", "cool" ]
];

function init_views()
{
  var listbox = document.getElementById("dialog.keywords");
  var tree = document.getElementById("dialog.related");

  listbox.myview    = new dynamicListBoxView();
  listbox.mybuilder = new dynamicListBoxBuilder(listbox);
  listbox.mybuilder.rebuild();

  tree.view = new customTreeView(tree);
}

window.addEventListener("load",init_views, true);
```

The `listdata` and `treedata` arrays hold the seminal data for the listbox and tree content. In this example, that content is still static, but this code is easy to modify so that dynamic changes to that data are also pushed to the widgets. The pairs of values in the `treedata` variable are related pairs of keywords. If the first such pair were expanded a little, it could almost pass for a fact:

```
<- "checkpoint", related, "breakdown" ->
```

This approach, therefore, is slowly leading us in the direction of RDF. There's no RDF in this experiment, however.

The `init_views()` function creates three custom JavaScript objects—two for the listbox and one for the tree. There is less to do for the tree because it has a built-in builder that automatically goes to work when the `<tree>` tag is displayed. Listbox also has a builder, but it is not exposed to the application

programmer until we learn about templates. We've used properties `myview` and `mybuilder` to emphasize that in the listbox case we're not overriding anything that the platform already has. This initialization step is done when the document loads, and there are no other event handlers that run later; but see the section "Custom Tree Views."

13.5.3.1 Dynamic Listboxes A dynamic `<listbox>` has all its displayed rows generated from a script. The script needs to implement both a view and a builder. Not only do these two objects create a working effect, they also give us some insight into the harder `<tree>` case. In the `<tree>` case, the builder (and sometimes the view) is hidden inside the platform's own code. The `<listbox>` case fully exposes these objects (because we create them), and we can imagine how the `<tree>` case works by analogy.

The first object, shown in Listing 13.14, is the view object for the listbox.

Listing 13.14 JavaScript object for the dynamic listbox View.

```
function dynamicListBoxView() {}

dynamicListBoxView.prototype = {
  get rowCount ()
  {
    return listdata.length;
  },
  getItemText: function (index)
  {
    return listdata[index];
  }
};
```

This object is a simple case of data-abstraction. The original data for the interface is hidden behind the view's interface. This object does more than that, however. It provides an interface that is expressed in terms of displayed rows in the `<listbox>`—rows or items. This association with the visible rectangular rows of the listbox is what makes it a view. It is convenient that the object uses an array of data, and that each member of that array exactly matches one listbox item. If, however, the underlying array were replaced with some other, more complex, data structure, the object would still be a view. This is because, to the outside world, its unchanged interface would still make the set of listbox items look like a simple list of viewable rows.

By itself, the View does nothing. It is the builder object that exploits that view. It appears in Listing 13.15.

Listing 13.15 JavaScript object for the dynamic listbox builder.

```
function dynamicListBoxBuilder(listbox) {
  this._listbox = listbox;
}
```

```
dynamicListBoxBuilder.prototype = {
  _listbox : null,
  rebuild : function () {
    var rows, item;

    while (_listbox.hasChildNodes())
      _listbox.removeChild(_listbox.lastChild);

    rows = _listbox.myview.rowCount;

    for (var i=0; i < rows; i++)
    {
      item = document.createElement("listitem");
      item.setAttribute("label", listbox.myview.getItemText(i));
      _listbox.appendChild(item);
    }
  }
}
```

The builder is also simple—it contains one method: `rebuild()`. When `rebuild` is called, the object goes to work on the `<listbox>`'s DOM 1 Core Element object. First, it removes all the existing listbox rows; then it adds back in a full set of up-to-date rows. Assumptions are made that the `<listbox>` has only `<listitem>` children and no column specifications, and for our case that is true. Again, it is convenient, but not necessary, that each displayed row equals exactly one DOM Element (a `<listbox>` element).

When the builder goes to work, it relies entirely on the view object. The only thing the builder knows about the content to be displayed is what the view tells it.

Both the view and the builder objects could be enhanced so that the listbox can be changed after it is built. In the current implementation, updating the listbox requires that (a) the array be changed and then (b) the builder's `rebuild()` function be called. Each such change is a 100% update of the listbox. To make only partial changes, extra methods would need to be added to the builder object, and some system would need to be put in place so that the builder knows when to make those partial changes, and what those changes are.

13.5.3.2 Custom Tree Views A customized tree is more complicated to understand than a dynamic listbox. Several pieces are in place already: Trees have an existing builder and view, and both are advised when a user's clicking opens or collapses part of a tree, or scrolls the tree's viewport. When parts of trees are opened or closed (but not when they are scrolled), the number of rows in the tree's display changes, even if the data underlying the displayed tree does not. So ordinary XUL trees are inherently more dynamic than listboxes because the number of viewable rows can change, whereas ordinary listboxes have a fixed number of viewable rows.

To proceed, we intend to fit in with the existing tree content builder and override the existing tree content viewer with our own view. That means creating an object with the `nsITreeView` interface. Such an object is quite large, so we attack it a piece at a time. The overall structure of this object is shown in Listing 13.16.

Listing 13.16 Skeleton of a JavaScript Custom Tree View object.

```
function customTreeView(tree) {
  this.calculate();
  this._tree = tree;
}

customTreeView.prototype = {
  // 1. application specific properties and methods
  calculate : function () { ... },
  ...
  // 2. Important nsITreeView features
  getCellText : function (index) { ... },
  ...
  // 3. Unimportant nsITreeView features
  getImageSrc : function (index) { ... },
  ...
 }
```

Here is a breakdown of this object. The `customTreeVew()` constructor is specific to our application. It calculates some information held internally in the object and retains a reference to the `<tree>` object. That is just preparation work. The prototype for the object contains all the ordinary properties and methods. We're free to add whatever features we want, as long as we also implement the `nsITreeView` interface. Part 1 of the prototype is these additional features, which the tree builder knows nothing about and doesn't need or use. Part 2 of the prototype is a portion of the `nsITreeView` interface. That interface has methods for many different aspects of `<tree>` content: drag and drop, styles, specialist content like progress meters, and so on. Some aspects of the interface are critical—either for the builder or for our own purposes—and that portion of the interface we will implement properly. Part 3 is the other part of the `nsITreeView` interface. For this other part, we'll provide stub routines that do nothing or almost nothing.

In fact, it's possible to omit some of the less-used methods of the `nsITreeView` altogether. If the builder decides to call those missing methods, which it might or might not do, then an error will appear in the JavaScript console. It's better to provide a minimal implementation of everything than gamble that something will never be needed.

Let's now address the three parts of this custom view object.

The first part, the application-specific part, is by far the most complicated. It is complicated because in this example we're going to design most of the requirements of the builder into this part. It's also complicated because

our initial data (the `treedata` array) is nothing like a hierarchical tree. We must fix that.

Our overall goal is to transform the raw data into other data structures. The first data structure is a hierarchy that will match the hierarchy that the tree displays. This data structure might represent all the possible data, or just the currently open-for-display subtrees. In our case, it's the open subtrees only. The second data structure is an indexed list of the displayed items in the tree. Even though a primary column of a `<tree>` can be hierarchical, the rectangular tree items of the tree form a simple ordered list. It is the index numbers of this simple list that are passed as arguments from the builder to the view and back again, and that make the view a view. This is the same as the dynamic listbox case. The custom view we make must be able to handle these indexes.

Our tactics for these data structures follow:

- Create a `relatedMatrix` data structure that represents all the keyword-to-keyword pairs. This will be handier to use than the simple list we are given and has nothing to do with custom views.
- Create an `openTree` data structure so that we can keep track of what the tree looks like. We need to stay synchronized with the user's actions on the tree.
- Create a `viewMap` data structure. This is a collection in the form of an array that maps from the tree item index to the `openTree` data structure.

These data structures are created in Listing 13.17, which is Part 1 of the custom view object's prototype.

Listing 13.17 NoteTaker specific part of a custom view object.

```
_ relatedMatrix : null,
 _openTree : null,
 _viewMap : null,

 calculate : function () {
   this.calcRelatedMatrix();
   this.calcTopOfTree();
   this.calcViewMap();
 },

 calcRelatedMatrix : function () {
   this._relatedMatrix = {};
   var i = 0, r = this._relatedMatrix;
   while ( i < treedata.length )
   {
     if ( ! (treedata[i][0] in r) )
         r[treedata[i][0]] = {};
     if ( ! (treedata[i][1] in r) )
         r[treedata[i][1]] = {};

     r[treedata[i][0]][treedata[i][1]] = true;
```

```
      r[treedata[i][1]][treedata[i][0]] = true;
      i++;
    }
  },

  calcTopOfTree : function () {
    var i;
    this._openTree  = [];

    for (i=0; i < listdata.length; i++)
    {
      this._openTree[i] = { container : false,
                            open : false,
                            keyword : listdata[i],
                            kids : null,
                            level : 0
                          };
      if ( listdata[i] in this._relatedMatrix )
        this._openTree[i].container = true;
    }
  },

  calcViewMap : function () {
    this._viewMap = [];
    this.calcViewMapTreeWalker(this._openTree, 0);
  },

  calcViewMapTreeWalker : function(kids, level) {
    for (var i=0; i < kids.length; i++ )
    {
      this._viewMap.push(kids[i]);
      if ( kids[i].container == true && kids[i].open == true )
        this.calcViewMapTreeWalker(kids[i].kids, level + 1);
    }
  },
```

That's a lot of code, but it's straightforward. First, there are three properties, all prefixed with underscore to indicate that they shouldn't be touched by users of the object. They will hold the view's internal data structures. Then there's the `calculate()` method, which builds these data structures, using a specific method in each case. The rest are those three specific methods.

The `calcRelatedMatrix()` makes a better-organized copy of the raw data. For the raw data in Listing 13.13, it makes the data structure of Listing 13.18.

Listing 13.18 Example keyword-to-keyword relationship matrix.

```
_relatedMatrix = {
  "checkpointed" : { "breakdown" : true,
                     "first draft" : true,
                     "final" : true },
  "reviewed" :     { "guru" : true,
```

```
                    "rubbish" : true },
  "fun" :           { "cool" : true },
  "guru" :          { "cool" : true,
                    "reviewed" : true },
  "breakdown" :     { "checkpointed" : true },
  "first draft" :   { "checkpointed" : true },
  "final" :         { "checkpointed" : true },
  "rubbish" :       { "guru" : true },
  "cool" :          { "fun" : true,
                    "cool" : true }
};
```

Every keyword in the original list of pairs appears as a property of the `_relatedMatrix` object, and every keyword that is related to it has a property on a subobject for that keyword. With this arrangement, both the forward- and backward-related cases are recorded, and it's easy to test if a given keyword is related to another one. This data structure is our official keyword reference in the view.

The next step is to create a hierarchical version of this related data. Each tree item can have zero or more children, and any such children are ordered. This means that each node of the tree (a data structure node, not a DOM node) is a list of children (a bucket). We'll use an array for each node. The `calcTopOfTree()` starts this data structure off. It appears in Listing 13.19.

Listing 13.19 Creation of the root node in a tree data hierarchy.

```
  calcTopOfTree : function () {
    var i;
    this._openTree  = [];

    for (i=0; i < listdata.length; i++)
    {
      this._openTree[i] = { container : false,
                            open : false,
                            keyword : listdata[i],
                            kids : null,
                            level : 0
                          };
      if ( listdata[i] in this._relatedMatrix )
        this._openTree[i].container = true;
    }
  },
```

This method creates an array of keyword records as the top-level node and puts into it the keywords that appear in the listbox—the ones in the listdata array. So the listbox keywords are all at the top level of the displayed tree. Each keyword record is an object that holds the keyword string, a reference to the immediate children of this node, and some housekeeping information. The housekeeping information is: whether this item is a container (`container="true"` in XUL), whether this container is open in the tree dis-

play (`open="true"` in XUL), and the depth of this node in the tree. When the time comes, other methods of the view will add to this tree.

Finally, we need to maintain an indexed list of the rows visible in the tree. Listing 13.20 shows how this is done.

Listing 13.20 Creation of the root node in a tree data hierarchy.

```
calcViewMap : function () {
  this._viewMap = [];
  this.calcViewMapTreeWalker(this._openTree, 0);
},

calcViewMapTreeWalker : function(kids, level) {
  for (var i=0; i < kids.length; i++ )
  {
    this._viewMap.push(kids[i]);
    if ( kids[i].container == true && kids[i].open == true )
      this.calcViewMapTreeWalker(kids[i].kids, level + 1);
  }
},
```

The `calcViewMap()` method is the starting point for constructing this list. It merely passes the root of the tree hierarchy to `calcViewMapTreeWalker()`, which is a recursive function. This function walks through the open parts of the tree left-to-right, which is the same order as the top-to-bottom order displayed in the `<tree>` tag. At each found keyword, it adds a reference to that keyword's record to the index item list. So each keyword record is tracked by both the tree and this list.

Altogether, these routines create data structures that are otherwise not yet used. These data structures are created inside the view object. The `relatedMatrix` structure is entirely static, unless it is re-created. The other two structures change over time.

Let's turn to the important parts of the `nsITreeView` interface. Listing 13.21 shows most of these methods. The tree's built-in builder will call these methods when it needs to access the data that the view provides. If any custom scripts are needed later, they can call these methods, too.

Listing 13.21 Most important methods of the `nsITreeView` interface.

```
get rowCount() {
    return this._viewMap.length;
  },

  getCellText: function(row, column) {
    return this._viewMap[row].keyword;
  },

  isContainer: function(index) {
    return this._viewMap[index].container;
  },
```

```
isContainerOpen: function(index) {
  return this._viewMap[index].open;
},

isContainerEmpty: function(index) {
  var item = this._viewMap[index];
  if ( ! item.container ) return false;
  return ( item.kids.length == 0); // empty?
},

getLevel: function(index) {
  return this._viewMap[index].level;
},

getParentIndex: function(index) {
  var level = this._viewMap[index].level;
  while ( --index >= 0 )
    if ( this._viewMap[index].level < level )
      return index;
  return -1;
},

hasNextSibling: function(index, after) {
  var level = this._viewMap[index].level;
  while ( ++index < this._viewMap.length )
  {
    if ( this._viewMap[index].level < level )
      return false;
    if ( this._viewMap[index].level == level && index > after )
     return true;
  }
  return false;
},
```

These methods and the `rowCount` property show how important the `viewMap` is—we can directly read out the needed results. This is because of our early design effort with data structures. Because keyword records contain a precalculate level value, the more complex of the methods, like `hasNextSibling()`, are still easy to implement. We need only look up or down the `viewMap` until the level changes to find the required row. In the case of `getParentIndex()`, this means looking up the list until the level decreases by one. In the case of `hasNextSibling()`, this means looking down the list for an item at the same level, but with no intervening items at lesser levels.

A final important method is the `toggleOpenState()` method. It is called when the user opens or closes a subtree. That action is unusual because it changes the number of rows in the list of items displayed by the tree. The implementation of `toggleOpenState()` must keep the view's data structures up to date when this happens, and it must also tell the underlying layout sys-

tem to redraw (repaint and re-layout) the tree. If these two things are not done, the view will be out-of-date in its understanding of the currently displayed view, and the tree display won't change until the mouse cursor leaves the tree's XUL box. Listing 13.22 is the code for this method.

Listing 13.22 The important `toggleOpenState()` method of `nsITreeView`.

```
toggleOpenState: function(index) {
  var i = 0;
  var node = this._viewMap[index];
  if ( ! node.container )
    return;
  if ( node.open )
  {
    node.open = false;
    node.kids = null;
    i = index + 1;
    while ( this._viewMap[index].level > this._viewMap[i].level)
      i++;
    i = i - index;
  }
  else
  {
    node.open = true;
    node.kids = [];
    for (var key in this._relatedMatrix[node.keyword])
    {
      node.kids[i] = { container : false,
                       open : false,
                       keyword : key,
                       kids : null,
                       level : node.level + 1
                     };
      if ( typeof(this._relatedMatrix[key]) != "undefined" )
        node.kids[i].container = true;
      i++;
    }
  }
  this.calcViewMap();
  this._tree.treeBoxObject.rowCountChanged(index,i);
},
```

This function aborts if the item supplied is not a container; otherwise, it handles both the open closed subtree and close opened subtree cases. In each case, it performs these tasks: updates the keyword record to match the new toggle state, updates the `openTree` hierarchy by trimming a closed subtree or adding a set of children records, and calculates the number of rows added or deleted using the `viewMap`. In the open case, the `relatedMatrix` is consulted to see how may children (related keywords of the current keyword) need to be added. After either operation, the `viewMap` is entirely recalculated to bring the view's understanding of the displayed rows up to date. The

final step is to tell the tree to refresh its display. To do that, we need to interact with the tree's special box object. Normally we only use that box object to scroll, but its `rowCountChanged()` method (from the `nsITreeBoxObject` interface) is exactly what we need. It accepts an index and a number of rows from that index as a hint that describes the part of the tree that needs to be re-painted.

In a simple tree with a custom view, such as this case, opening or closing a subtree is the only way to change dynamically the list or rows currently displayed. If, however, event handlers are added to the tree, then those handlers might also make dynamic changes. An example is a mouse click on a tree row that deletes that row. For this to work, the handler must manipulate the view's data structures and call `calcViewMap()` and `rowCountChange()` just as `toggleOpenView()` does. The cleanest way to arrange that is to add extra methods to the view that do the work and then to call those methods from the handler.

Having covered the application-specific methods of the view, and the important `nsITreeView` methods, what remains of the custom view's object prototype are the unimportant `nsITreeView` methods. For another application, some of these methods might be critical, but they're not in our case. Listing 13.23 shows our implementation of them.

Listing 13.23 Less important methods of `nsITreeView`.

```
  canDropBeforeAfter: function(index, before) { return false; },
  canDropOn: function(index) { return false; },
  cycleCell: function(row, column) {},
  cycleHeader: function(col, elem) {},
  drop: function(row, orientation) { return false; },
  getCellProperties: function(row, prop) {},
  getCellValue: function(row, column) {},
  getColumnProperties: function(column, elem, prop) {},
  getImageSrc: function(row, column) {},
  getProgressMode: function(row, column) {},
  getRowProperties: function(row, column, prop) {},
  isEditable: function(row, column) { return false; },
  isSeparator: function(index) { return false; },
  isSorted: function() { return false; },
  performAction: function(action) {},
  performActionOnCell: function(action, row, column) {},
  performActionOnRow: function(action, row) {},
  selectionChanged: function() {},
  setCellText: function(row, column, value) {},
  setTree: function(tree) {}
}; // end of customTreeView.prototype
```

These methods either say "No" or do nothing. Our custom view doesn't support drag-and-drop anywhere on the tree. There are no properties on cells, rows, or columns, and no cell values, images, progress meters, editable fields, or implemented actions. We have no sorted columns or `<treeseparator>`

tags, and we don't care if the user selects rows. The `setTree()` method is run at tree initialization, and there's nothing for us to do there; although we could call `calculate()` in that method if we wanted. As it stands, we call `calculate()` when the view object is created.

After a fair amount of code, our experiment with custom views is complete. This experiment has several noteworthy results.

The first result is obvious: Using a custom view we can base a <tree> on non-XML content using JavaScript. That might be preferable to manipulating the DOM, especially for examples like ours where the original data for the tree are not hierarchical.

The second result is not so obvious: Using our custom view the resulting tree is *infinite* in size. If A is related to B, then B is related to A, and so opening a subtree on one always reveals a further subtree on the other. Such infinite trees clearly can't be created with static XUL. In the next chapter, we'll see how partially constructed (and therefore possibly infinite) trees are a common feature of the template system.

Our final result is that we've seen some of the inside mechanics of the <tree> tag. That tag is backed by several platform-supplied interfaces and structures (like a builder), and it pays to appreciate what's going on in such a powerful widget.

We've now finished the UI and the data model for the NoteTaker tool. In the following "Hands On" sessions, we'll finish it using RDF and templates. That will be the last set of changes we'll make to the way data are stored in the tool.

13.6 Debug Corner: Making <listbox> and <tree> Work

The <listbox> and <tree> tags are more complex than form elements and can be very frustrating to use if the wrong approach is used. <listbox> is still fragile and a little tricky, and that situation demands an element of caution. No matter how it may look, the <tree> tag is not fragile. It works properly, and if nothing displays inside the tree, then it must be the application code at fault. As Mozilla matures, these two tags will no doubt become more user friendly as well as more robust.

The main problem with <listbox> and <tree> stems from XUL. There are no detailed run-time syntax or layout checks that can act as a compiler, although debug features of the platform can spew out some diagnostics if required. This means that success with these widgets requires good programmer habits—in particular structured testing.

The recommended way to proceed is to always, always start with the simplest possible static widget and to work toward the desired end point using a series of simple increments. Test every change to ensure that it builds up the widget as you intend. If unexpected results occur, you can retreat one step to the last safe position. Under no circumstances should you dump a screen full

of new code into a tree or listbox. Always start with a <listbox> or <tree> that has a single, static row, even if you've made it all work before.

Many specific problems with these widgets are the result of simple oversights. Some of the more common oversights include the following:

- Lack of flex="1" attributes. The tags surrounding <tree> and <listbox>, those two tags themselves, and column specifier tags all benefit from flex.
- Using shorthand syntax for <tree>. <tree> and its content tags have no shorthand or abbreviated syntax. All tags are required and must be fully stated. <listbox> does have some shorthand.
- Missing column ids. If your <treecol> tags are missing column ids, you can't select anything in the tree.
- No height for <tree>. The <tree> tag has no default height. One must be implied by the surrounding layout, or that tag must be given a height.
- Not enough homework. It's a fact of life that the AOM, XBL, DOM, and XPIDL documentation types are all stored in separate formats, plus this book. They take some exploring before you can claim to have the landscape mapped out. Try examining a <tree> and a <listbox> with the DOM Inspector, or review the content of this chapter.
- Confusion about "item indexes." In the tree view interface, item indexes represent the series of vertically stacked rectangles in the tree that each displays a row. That set of rectangles is clipped by the tree viewing area. "Item indexes" do not represent the tags that make up the content of the tree, or even all the rows that might be displayed by the tree. These indexes only represent the currently open subtrees and their exposed rows. The same is true for listboxes, but in that case, there is a one-to-one correspondence between viewable rectangles and listbox rows.

Overall, <tree> and <listbox> can't always be used as trivially as <button> can, so take care. Until <listbox> has a long track record of robustness, beware of laying it out in the same horizontal space as other XUL tags. Layout and behavior can be unusual in that case. Always try to arrange matters so that <listbox> dictates the size of its layout area, not its sibling, parent, or other nearby tags.

13.7 SUMMARY

<listbox> and <tree> are powerful widgets in the XUL bestiary. They are like form elements on steroids. <listbox> is a little fragile, and <tree> a little complicated, but they are both flexible display systems for serious data-oriented applications.

Even so, you can only go so far displaying static data. The alternative of scripting up content changes using the DOM is bulky, slow, and awkward. For dynamic data, something new is required. Templates are that new technology, and they are discussed next.