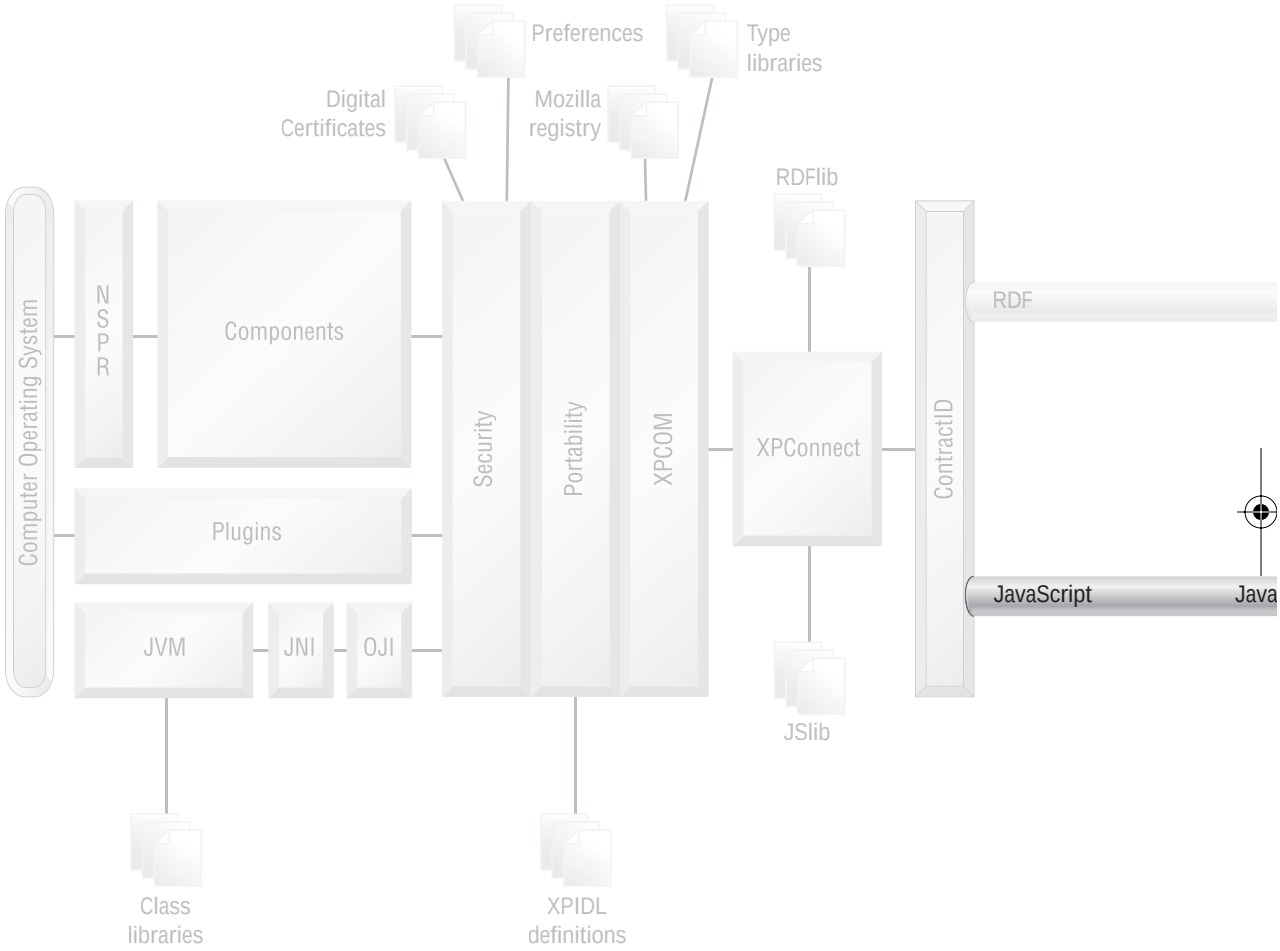
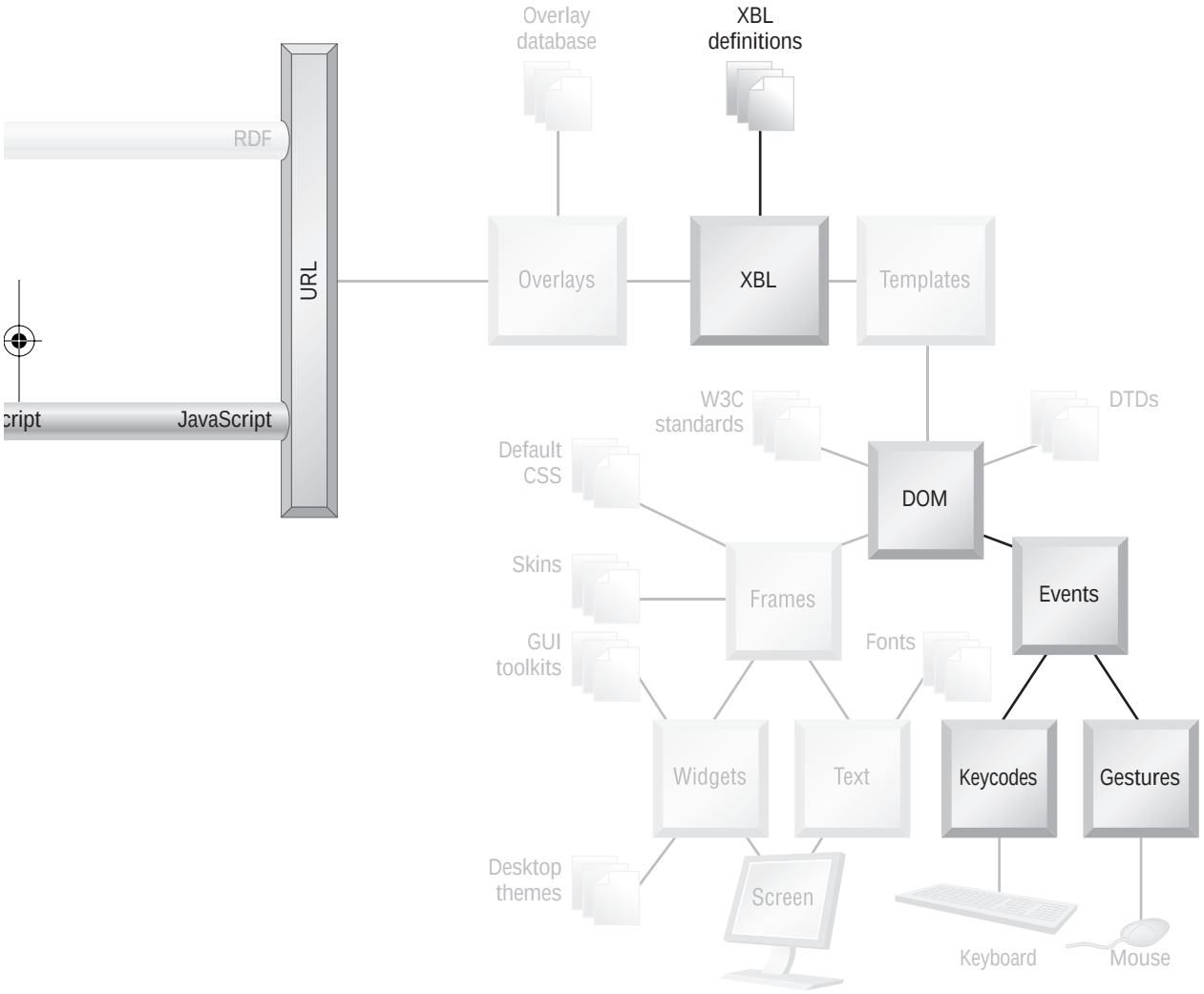

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
Plugins
JavaScript
Java
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 15

XBL Bindings



This chapter explains how to enhance the XUL language with new tags and new behavior using XBL (XML Binding Language).

XBL is an XML-based language that allows new, fully featured tags to be added to XUL, HTML, and XML. It is an efficient system for creating new GUI widgets. It is specific to Mozilla.

Plain XUL allows programmers to create user-defined tags like `<mytag>`, but such tags aren't very useful. They can be styled but are merely simple box-like tags. XBL, by comparison, allows for the creation of whole widgets with distinctive appearance and behavior. The content of an XBL widget is made up of tags drawn from XUL and HTML and from other tags based on XBL. An XBL widget is not as flexible as a Java applet or a plugin. It cannot start with a blank, rectangular canvas, and use sophisticated graphics libraries to draw on that plane. XBL can only combine existing tags.

XBL does not create new tags; it creates new bindings. A *binding* is a bundle of tags and scripts that together provide the features of a new widget-like event handler and displayable content. This bundle is a binding, and a binding is like an object. In Mozilla, a binding is matched to a user-defined tag with the Mozilla-specific CSS2 property: `-moz-binding`. After the binding is matched to the tag, they are said to be bound. A trivial binding is shown in Listing 15.1. This is a fragment of an XBL document and implements a widget that is just a smiley face (an emoticon).

Listing 15.1 Trivial example of an XBL binding.

```
<binding id="smiley">
  <content>
    <xul:image src="face.png"/>
  </content>
  <handlers>
    <handler event="click" action="alert('have a day')"/>
  </handlers>
</binding>
```

Such a binding could be attached to a user-defined tag named `<smiley/>` with a line of CSS like so:

```
smiley { -moz-binding: url("smiley.xml#smiley"); }
```

The `<smiley/>` tag will now displays a face whenever it is used in a XUL document. If the face is clicked on, it pops up an alert encouraging you to have a day. This is a very simple widget with its own trivial interactivity. Neither XUL nor HTML supplies a `<smiley>` tag.

The XBL binding system is built into the C/C++ code of the Mozilla Platform, but the bindings themselves are written using only JavaScript and XML. This fully interpreted environment makes bindings as easy to create and manage as XUL or HTML. This ease of use has been well-exploited in Mozilla. Many XBL bindings contribute to Mozilla applications such as the Classic Mail & News and the Classic Browser. Nearly all XUL tags have an XBL binding.

Throughout this book, references are made to `.xml` files in the `toolkit.jar` chrome file. These `.xml` files are XBL bindings, usually several per file. Some XUL tags documented in this book are nothing more than XBL bindings at work. Good examples are highly specialized tags like `<tabbrowser>` and `<colorpicker>`, which are defined purely in XBL. Even simple tags like `<button>` have XBL bindings.

Beyond Mozilla, a technology similar to XBL is the PostScript language. PostScript is used mostly for printing, but PostScript scripts can also be created by hand. Postscript allows chunks of displayable content to be named and reused. This makes it easy to construct large documents. List processing languages like Lisp and Tcl/Tk also provide an interpreted way of constructing GUIs, but they are more program-like than XML-based XBL. Tcl/Tk is probably XBL's biggest competitor in the Linux world. The pre-supplied components that Microsoft's .NET provides and the much simpler behaviors of Internet Explorer are XBL competitors in the world of Microsoft Windows.

An XBL binding has a clearly defined interface and a unique identity, and that makes XBL a tiny component system. Just as XPCOM is designed for 3GL components, XBL is designed for XML components. XBL's component model is much simpler than XPCOM's.

Because XBL bindings are made from XML and JavaScript, bound elements can be freely intermixed with HTML or XUL. HTML intermixing is unusual because the set of tags used in HTML pages is fixed and standard. It is far more common to see XBL used in XUL applications.

The NPA diagram at the start of this chapter shows the extent of XBL in the Mozilla Platform. Despite its status as a component system, XBL is a front-end Mozilla technology. During the loading of a XUL document, XBL bindings are identified and pulled separately into the final document's content. This inclusion process has some similarities with the overlay system. After tags bound to XBL bindings are loaded, they display their content like any XUL tag. The XBL system also has special support for capturing user input events. It uses a capture system almost the same as the XUL `<key>` tag.

This chapter continues with an overview of bindings and then gets down to the specific tags.

15.1 BINDING CONCEPTS

XBL extends XML and the traditional Web environment the way C++ extends C. It provides object-like features on top of a system that has no object-like features to start with.

15.1.1 An Example Binding

Every XBL document is just a list of bindings, each stated with a `<binding>` tag. Constructing a binding is similar to constructing an object's class. Listing

15.2 shows a simple XBL binding that is a variant of the XUL <checkbox> tag's binding.

Listing 15.2 Example XBL binding similar to <checkbox>'s binding.

```
<?xml version="1.0"?>
<bindings
  xmlns="http://www.mozilla.org/xbl"
  xmlns:xbl="http://www.mozilla.org/xbl"
     xmlns:xul="http://www.mozilla.org/keymaster/gatekeeper/
          there.is.only.xul"
>
<binding id="checkbox" extends="general.xml#basetext">

  <resources>
    <stylesheet src="chrome://global/skin/checkbox.css"/>
  </resources>

  <content>
    <xul:image class="checkbox-check" xbl:inherits="checked,disabled"/>
    <xul:hbox>
      <xul:image class="checkbox-icon" xbl:inherits="src"/>
      <xul:label xbl:inherits="xbl:text=label,accesskey,crop"/>
    </xul:hbox>
  </content>

  <implementation implements="nsIDOMXULCheckboxElement">
    <method name="check" action="this.checked = true"/>
    <property
      name="checked"
      onset="if (val) this.setAttribute('checked','true');"
      onget="return this.getAttribute('checked') == 'true';"
     />
  </implementation>

  <handlers>
    <handler event="click" button="0"
    action="if (!this.disabled) this.checked = !this.checked;"/>
    <handler event="keypress" key=" ">
      <![CDATA[
        this.checked = !this.checked;
      ]]>
    </handler>
  </handlers>

</binding>
</bindings>
```

Listing 15.2 shows many of the standard features of a binding. The <resource> and <content> sections describe data in the form of styles, images, tags, and plain XML elements. These sections are used to deliver XML content. The <implementation> and <handlers> section describe proper-

ties, methods, and event-handling hooks. These sections are installed as JavaScript and DOM content. The content and scripting aspects of a binding are like the data and code forks in a PC or Macintosh program. The pale content is XUL and JavaScript code. It is used *by* the binding, but it is not *part of* the XBL language.

It is common practice to study other people's bindings. "Reading Others' Code: Naming Conventions" in Chapter 16, XPCOM Objects, and Table 16.1 provide some reading hints for bindings created by mozilla.org.

15.1.2 Standards

XBL is an application of XML. Its XML definition can be found at *www.mozilla.org/xbl*. Mozilla does not load this URL; it is just used as an identifier that is recognized when encountered.

The standard suffix for a file containing XBL is `.xml`. The standard MIME type for an XBL document is `application/xml`.

XBL bindings are identified by a URL. There is no separate URL scheme for XBL; instead the two most common access methods are the `http:` and `chrome:` schemes. Like RDF, binding URLs include a `#id` suffix that identifies a given binding within an XBL document. Like RDF, that identified binding is considered a whole resource rather than a resource fragment. An example URL is

```
chrome://global/content/bindings/button.xml#button.
```

XBL has been submitted as a Note to the W3C. This means that a document describing XBL has been submitted to that organization. A W3C Note is a technology description lodged as a proposed solution to an issue that the W3C is considering. It is not a draft standard or "on the standards track"; it is just a proposal. The latest version of the XBL Note can be found at *www.mozilla.org/xbl/xbl.html*.

Mozilla's implementation of XBL is different from the Note submitted to the W3C. It contains additional features and does not implement all the features in the Note. The "Non-Tags and Non-Attributes" topic in this chapter discusses the differences.

Standards competing with XBL include ECMA-290 "ECMAScript Components" which is used by Microsoft in its WSH (Windows Scripting Host) technology. The WSDL standard and its related standards attempt to provide a distributed naming system for components and services, but WSDL is not as closely tied to user interaction as XBL is. Another semicompeting standard is XSL. XSL can process an XML file and replace specified tags with other content. That is the same as XBL, but XSL operates in batch mode, performing a single pass over the supplied document. XBL can be applied after a document is fully loaded. XBL is a helper system, rather than a full processing step like XSL. No other standard really addresses the niche that XBL addresses.

XBL has a broader agenda than most W3C standards. HTML, CSS, and JavaScript can only interoperate to a certain degree. Those three standards all focus on one tag, one style, and one object at a time. Although these standards allow content tags, style rules, and object properties to be grouped together, this grouping is quite narrow. There is no structural support across Web standards for programmers who need high-level object or component concepts to work with. XBL seeks to fill this gap for programmers by integrating XML, scripting, and styles into one structure—a binding.

15.1.3 Relationship with DOM Hierarchies

XBL bindings must be installed into a DOM hierarchy before they can be used.

Before a binding can be used, it must be *bound* to an existing tag in an existing XML document. That existing tag is called the *bound tag*, and the existing document is called the *target document*. If the bound tag has content of its own, that content is called the *explicit content*. If the binding contains a `<content>` tag, then the tags inside that `<content>` tag are called *anonymous content*. There is at most one set of anonymous content per binding, but there can be a different set of explicit content for each bound tag in the target document. Both types of content, plus the other details in the binding, add to the document that uses the binding.

When a binding is bound, the DOM hierarchy in the target document grows. Three changes are made:

1. The bound tag gains the properties, methods, and event handlers of the binding.
2. The bound tag has its content replaced with a mix of explicit and anonymous content.
3. The anonymous content is separately added to the DOM and is available via a special mechanism.

Steps 1 and 2 are repeated for each instance of the bound tag in the target document. This is a content generation process very similar to overlays and templates. Like overlays, content is inserted into the target document at specific points. Like templates, extra content is created by repeatedly copying a standard set of tags. The Mozilla Platform automatically performs these steps.

The full XBL definition of the binding is not available in the bound document. If necessary, it can be loaded and examined just as any XML document can.

These principles are broken down into steps in “How Bindings are Processed” later in this chapter.

15.1.4 Default Actions

The XBL system can be used to implement the default actions that HTML and XUL tags and widgets offer.

An example of a default action is given by the HTML `<input type="submit">` tag. When the button widget associated with this tag is clicked, form data are submitted by a browser to some Web server. No scripting is required to make this submission happen. The default action is part of the DOM Event processing that occurs for all XML tags. But how exactly is a form submission implemented, and how is it connected to the DOM Event processing?

The answer is that a default action can be implemented in an XBL binding. A binding provides a means to specify what DOM event and what JavaScript handler go together. When an event is generated, the XBL binding is automatically detected as the source of the default action, and the right handler is run, causing that action to occur.

A given widget can have a different default action for each kind of user input event. In the case of the `<input type="submit">` tag, pressing the Return or Enter key or clicking the displayed button produces the same effect. This does not have to be the case when a binding is created—every event can cause a different outcome.

Using a traditional even handler, the application programmer can override default actions. This, however, must be attacked correctly. If the application handler is installed as an attribute value (such as an `onclick` handler), then that handler will be used instead of the XBL handler. If the application handler is installed using `addEventListener()`, then this handler must be installed before the XBL handler is bound to the tag. This requirement puts the application handler ahead of the XBL handler on the list of event handlers. To achieve this status, see the remarks in "Scripting."

An XBL binding is not responsible for the event capture and bubbling processes. It is not responsible for maintaining the input focus or the state of the window's focus ring. Those things are done deep in the C/C++ part of the platform and have nothing to do with XBL. Mozilla's accessibility support (for the disabled) is implemented using XBL, but the input system that first captures accessibility commands is part of the core platform.

15.1.5 Object-like Features

A bound XBL binding has object-like features and can be considered an object from a JavaScript or DOM perspective.

An object interface consists of object attributes, methods, and exceptions. Binding interfaces contain object properties, methods, and handlers. Properties are effectively object attributes, and handlers are just special-purpose object methods. Exceptions can also be implemented in a binding, but XBL has no explicit syntax for them. To create an exception, use the `throw` feature of JavaScript instead.

XBL also has support for traditional object-oriented concepts. Table 15.1 is an overview of the OO support that XBL provides.

Table 15.1 XBL object support

Object concept	XBL support
Aggregation	XBL tags collect content (including bound content) and JavaScript code together into a binding, and merge explicit and anonymous content into a final content set.
Containment	The bound tag holds all content and JavaScript logic. An XBL binding can contain other bound tags, and therefore other bindings.
Encapsulation	A bound tag holds all the useful information a binding supplies, although some housekeeping information can be accessed using the JavaScript document object.
Inheritance	XBL supports the inherits attribute, for inheriting XML attributes, and the extends attribute, for inheriting whole bindings.
Information hiding	When a binding is bound, the binding specification is hidden from the bound tag. Only the interface implied by that specification is available.
Interfaces	A binding that implements an interface can be inherited by another binding by use of the extends attribute. A binding can specify what XPCOM interfaces it supports using the XBL implements attribute.
Late-binding	Bindings are loaded independently and asynchronously into a bound document. A binding can be bound or rebound using CSS2 rules at any time. Bindings must always be referred to explicitly. All bound bindings must be concrete. There is no mechanism for finding a base binding for a derived binding.
Object-based	Bound bindings look like objects from a JavaScript perspective.
Object-oriented	The XBL extends attribute supports basing XBL bindings on other XBL bindings. This support includes the concept of an inheritance chain. See following discussion.
Multiple inheritance	XBL only supports single inheritance of bindings.
Run-time type reflection	XBL has no automatic support for type reflection. A binding can implement the nsIClassInfo interface if it wishes.

Each binding can extend (inherit) one other binding. That other binding can in turn inherit a further binding, and so on. Such a set of inherited bindings form an order list called an *inheritance chain*. When a particular binding is bound, that particular binding is at the head of the chain. That binding is also called the *primary binding*. The chain is always at least one binding long. If the chain is exactly one binding long, then there is no inheritance at work.

Figure 15.1 shows an example of an inheritance chain, based on the XUL `<button>` tag.

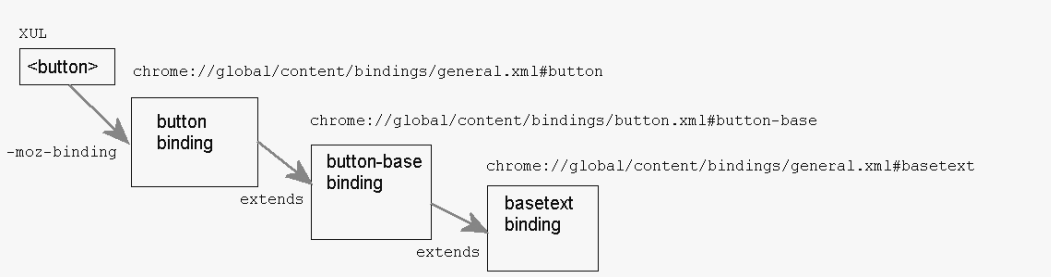


Fig. 15.1 XBL inheritance hierarchy for the XUL <button> tag.

This example shows an inheritance chain of three items. The "button" binding is bound to the <button> tag, and that binding is the head of the chain. The other two bindings are general-purpose bindings that are inherited in many places across XUL. The button-base binding is used for all (or most) button-like widgets, and the basetext binding is used for all widgets that have a textual component. button and button-base are defined in the same XBL document (button.xml), but basetext is defined elsewhere (in general.xml). All three of these bindings contribute features to the <button> tag's DOM object.

Two bindings can inherit the same base binding, so the full set of inheritance chains can be viewed as a tree (or grove of trees). This treelike structure might be an aid at design time, but there is no way to navigate it or inspect it from a script.

Table 15.2 shows how the various features of XBL bindings are inherited.

Table 15.2 Inheritance of XBL binding features

XBL feature	Inherited?
<resources>	All bindings in the inheritance chain have their resources loaded. All stylesheets have the same default weight, but rules nearest the head of the chain are applied later and override earlier rules.
<content>	No inheritance. The binding at the head of the chain is the sole binding used. If bound tags are contained inside <content>, rather than inherited, then their bindings are applied.
<field>, <property>, and <method>	Properties and methods are inherited. If the derived binding duplicates a name in a base binding, the derived version replaces the base version, and the base version is not available from logic in the derived binding.
<constructor> and <destructor>	Both are inherited. Constructers are run one at a time in order from the ultimate base binding to the final derived binding. Destructors are run one at a time in order from the final derived binding to the ultimate base binding.

Table 15.2 Inheritance of XBL binding features (Continued)

XBL feature	Inherited?
<handlers>	Handlers are inherited. Derived binding handlers override base binding handlers that define the same event.
display attribute	No inheritance. Only the attribute on the most derived binding is used.

15.1.6 XBL Component Framework

The XBL system is a very simple component framework. A component framework allows standardized components to interoperate in a convenient way. A binding is a form of component.

A component system similar to XBL is Perl's module system, which is accessed with the Perl `use` and `require` keywords. Both systems are quite primitive. XBL bindings can be inspected via the DOM just as Perl modules can be inspected via symbol tables. XBL bindings are not directly included the way Perl modules are, but in general terms they are overlaid on other code.

Component names in the XBL system are URLs of the form `Resource#Id`, where `Resource` is a document's Web address and `id` is the value of the `id` attribute for a `<binding>` tag. Like RDF, XBL treats `id`-qualified URLs as whole resources, not as offsets in an existing resource.

XBL has no component registry or name service and no way of querying the specification of a binding. Instead, each XUL or HTML document maintains a simple internal list of the currently active bindings for that document.

The XBL component system is a distributed system because bindings can be retrieved from URLs anywhere. Bindings do not communicate back to the server they are loaded from unless they contain code designed to do just that.

XBL does not have the features of complex component systems like Microsoft's COM, Mozilla's XPCOM, OMG CORBA, or Sun's JavaBeans. Its strengths are simplicity and immediacy, not expressive power or tight integration.

15.1.6.1 Bindings as XPCOM Components An XBL binding can implement one or more XPCOM interfaces. This allows the DOM object for a bound tag to act like an XPCOM component with those interfaces.

A binding cannot act as an XPCOM component unless it is bound first.

15.2 Constructing One XBL Binding

This topic describes the individual XBL tags and tag attributes.

Some XBL tags have the same names as tags used elsewhere in Mozilla. It is always a good idea to provide full `xmlns` namespace declarations when

creating XBL documents. That strategy helps avoid confusion. XBL tag names with meanings in other XML applications supported by Mozilla are

```
<bindings> <body> <children> <content> <image>
```

XBL attribute names with meaning elsewhere are

```
method action command modifiers charcode keycode key name readonly
     text
```

15.2.1 Bound Tags and `-moz-binding`

Most, but not all, XUL and HTML tags can be bound to an XBL binding. For a tag to be bound, it must have a frame—a styleable rectangular area in which it is displayed.

Any tag with this style has no frame and cannot be bound:

```
{ display: none; }
```

A tag that gains this style will lose any binding that it has. For this reason, a bound XUL tag should be collapsed rather than hidden if it needs to be visually suppressed.

Tag instances, not tag names, are bound to a binding. Each style rule that contains `-moz-binding` binds a set of tag instances. Any pattern that represents a CSS2 selector can be used as a set of instances to be bound. Listing 15.3 shows a number of examples.

Listing 15.3 CSS2 selector example for bindings.

```
box           { -moz-binding: url("binding.xml#sample"); }
#id           { -moz-binding: url("binding.xml#sample"); }
.class        { -moz-binding: url("binding.xml#sample"); }
box[X="here"] { -moz-binding: url("binding.xml#sample"); }
box,vbox,hbox { -moz-binding: url("binding.xml#sample"); }
box vbox      { -moz-binding: url("binding.xml#sample"); }
```

The first line binds all `<box>` tag instances to the same binding. Each tag instance has its own interface and its own set of state. The second line binds just one tag, regardless of its name. The third line binds any and all tags with a given class attribute. The fourth line binds only a subset of `<box>` tags, those that have the `X="here"` attribute. The fifth line binds all `<box>`, `<vbox>`, and `<hbox>` tags, and so on. Where two style rules bind the same tag instance (a very bad piece of design), the more specific of the two rules takes precedence, or the later of the two, if the two selectors are the same. That is normal CSS2 processing.

This style rule support is used in XUL for tags that have variant types, like `<button>`. Each variant is bound to a different binding. Variant bindings can be seen in `xul.css` in `toolkit.jar` in the chrome.

Bindings can be explicitly avoided as follows:

```
{ -moz-binding : none; }
```

Mozilla (to version 1.4 at least) has some issues adding and removing bindings; the advice in "Scripting" in this chapter should be followed rather than just applying and unapplying style rules.

Finally, a long-standing bug (at least to version 1.4) means that a tag with this style will not display properly if bound:

```
{ display: block; }
```

Because blocks are not part of XUL, this is only a trap for HTML. Avoid combining this style with `-moz-binding`.

15.2.2 <bindings>

The `<bindings>` tag is the outermost tag in an XBL document. It has no special meaning of its own and no special attributes. It can contain only `<binding>` tags.

A <!DOCTYPE> declaration is not needed for XBL documents. If one is added, the minimum required is

```
<!DOCTYPE bindings>
```

15.2.3 <binding>

The `<binding>` tag holds one binding specification. It can have up to four child tags. Each of the following tags should appear at most once:

```
<resources> <content> <implementation> <handlers>
```

All four are optional. It is recommended that they appear in the order shown, but that is not mandatory. A `<binding>` tag with no content is useless, except that it will remove all content and default actions from any tag bound to it. That effect is similar to the style "`-moz-binding : none`".
The `<binding>` tag has the following special attributes:

```
id display extends inheritstyle
```

The `id` attribute is unique and identifies the binding. A binding does not have a name, it has a URL identifier. If the `#id` part of the URL is left off, the first binding in the XBL document will be used.

The `extends` attribute is XBL's inheritance mechanism. It can be set to the URL of one other binding. A series of bindings specified with `extends` form an XBL inheritance chain, as discussed in "Object-like Features."

The `extends` attribute also accepts a shorthand value. That shorthand value is used for the `display` attribute as well. It consists of a `xul:` prefix and a XUL tag name. This shorthand says that the binding is based on an existing tag and should use that tag's features. In the case of `extends`, this attribute provides a base implementation for the binding to enhance. An example is

```
extends="xul:box"
```

The `display` attribute accepts only the shorthand notation used for `extends`. It states what box object to apply for the binding. The box object identified is the one that normally applies to the stated XUL tag. An example is

```
display="xul:button"
```

This causes the binding to adopt the box object that the XUL `<button>` tag has.

The `inherit` style attribute can be set to `false`. It is `true` by default. If it is set to `false`, the bound tag will not benefit from any styles that were associated with the tag before the binding was added.

15.2.4 `<resources>`

The `<resources>` tag states what other documents a given binding requires. It is similar to the `<LINK>` tag of HTML. The `<resources>` tag has no special attributes and can only contain these XBL tags:

```
<image> <stylesheet>
```

Zero or more of either tag can be stated. Order is unimportant, except that later `<stylesheet>` tags will be applied after earlier ones. The XBL system has a little intelligence about the contents of the `<resources>` tag, as described next.

15.2.4.1 `<image>` The `<image>` tag is identical to the XUL `<image>` tag, except that the only useful attribute is `src`. As for XUL, `src` specifies the URL of an image.

```
<image src="icons.gif"/>
```

XBL does nothing with this tag except tell the Mozilla Platform to retrieve the matching image, which is then cached. The `<image>` tag is a hint to the platform that the image is important and may be needed later. Later uses might be in the XUL `<image>` tag, the HTML `<img>` tag, the CSS2 list-style-image property, or in the content part of an XBL definition. Stating an `<image>` tag is just a performance optimization.

15.2.4.2 `<stylesheet>` The `<stylesheet>` tag is identical to the HTML `<style src=>` tag. It cannot contain any inline content of its own.

```
<stylesheet src="chrome://global/skin/button.css"/>
```

Stylesheets are used in XBL to contain all the presentation detail of the XBL widget. This allows the presentation details to be theme-dependent. It also allows style rules to be created that only apply to the anonymous content of the bound tag. In the previous example, all the presentation detail for the XUL `<button>` tag (which is based on an XBL binding with `id="button"`) is stored in a theme-dependent URL.

More than one <stylesheet> tag may be used per binding, but this is rare. If so, the stylesheets are applied in the order they appear.

If a binding has an inheritance chain of other bindings, and those other bindings have <stylesheet> tags, then all stylesheets will be applied. The stylesheets are applied in order from the tail to the head of the chain so that the primary binding has highest precedence.

The standard XUL tag bindings stored in toolkit.jar in the chrome have a stylesheet each. Those stylesheets are theme-dependent. A theme designer should implement all these standard stylesheets for a given theme, or else the display of XUL tags will not be consistent under that theme. The same goes for themes that hope to support Mozilla's standard applications such as Messenger. The stylesheets associated with XBL bindings used in the Messenger client should also be implemented by a theme designer if these applications are to benefit from the new theme.

15.2.5 <content>

The <content> tag holds all the anonymous content of the binding. It can hold XML tags from any XML namespace, such as HTML, XUL, or MathML. It can also hold the XBL-specific <children> tag. XUL observers and broadcasters, along with the <script> tag, do not function when used as anonymous content.

The <content> tag used to have two special attributes:

```
includes excludes
```

Both attributes are deprecated and should be avoided. Use the <children> tag in the anonymous content instead. If any other attributes appear on the <content> tag, they will be copied to the bound tag.

Preexisting attributes of the bound tag can be added to the tags inside <content> using the inherits attribute. Explicit content of the bound tag can be merged with the tags inside <content> using the <children> tag.

If a binding uses the extends attribute, so that it builds on another binding, then any <content> tag in that other binding is ignored.

If the <content> tag of the bound binding is empty, then the bound tag will have no content either.

15.2.5.1 Merging Attributes with xbl:inherits= The anonymous content of a <content> tag can be a document fragment of any type and of any size. The XBL <children> tag can appear anywhere inside that document fragment.

Non-XBL tags in the content can use one XBL tag attribute: the inherits attribute. This attribute is used to pass parameters and their values from the bound tag to the anonymous content, as though those content tags were observing the bound tag. This parameter passing is done by simply renaming and copying the bound tag attribute values to the anonymous content tags.

This mechanism gives the user of the bound tag some control over the anonymous content. For this to work, the binding creator must anticipate the user's needs and strategically place `inherits` in the binding's anonymous content.

The `inherits` attribute holds a comma- or space-separated sequence of attribute mappings. An attribute mapping states what attribute on the anonymous content tag should receive the value of what attribute on the bound tag. Each attribute mapping uses one of the following four syntaxes:

```
att1
att1=att2
xbl:text=att2
att1=xbl:text
```

`att1` and `att2` are any legal XML attribute names.

- The first syntax form says that the `att1` attribute on the bound tag, and its value, should be copied straight to the anonymous tag.
- The second syntax form says that the `att1` attribute on the anonymous tag should have the value of the `att2` attribute on the bound tag.
- The third form uses the special reserved name `xbl:text`. It says that the anonymous tag's content (a DOM 1 Text Node) should contain the value of the `att2` attribute on the bound tag. This achieves the same effect as `<textnode>` in XUL templates.
- The fourth form says that the `att1` attribute on the anonymous tag should contain the value of the bound tag's explicit content, which must be a DOM 1 text node.

Because the `inherits` attribute is an XBL attribute, it can't appear alone inside a non-XBL tag. It must be fully specified with an XML namespace. This is done by prefixing it with `xbl:` (or similar) and adding another namespace declaration to the top of the XBL document. That is why some XBL documents have two references to the XBL namespace, as shown here:

```
<bindings id="test"
   xmlns="http://www.mozilla.org/xbl"
   xmlns:xbl="http://www.mozilla.org/xbl"
   xmlns:xul="http://www.mozilla.org/keymaster/gatekeeper/
     there.is.only.xul"
>
```

The first XBL namespace is the default; the second is stated specifically for the `inherits` attribute.

When all these things are put together, a full example of inherits is

```
<xul:label dir="ltr"
     xbl:inherits="xbl:text=value,style,align=justify"/>
```

If the bound tag were defined this way:

```
<mytag value="Test" style="color:red" justify="start"/>
```

then, after substitution, the anonymous <xul:label> tag would be replaced by

```
<xul:label dir="ltr" style="color:red" align="start">
Test
</xul:label>
```

The inherits attribute can be used anywhere in the anonymous content and can map one attribute to many different anonymous content tags if required. Anonymous content tags are always free to specify attributes directly, like dir="ltr". If the special xbl:text attribute is used more than once, all assigned values will appear in the text node of the anonymous tag.

15.2.5.2 Merging Tags with <children>

The <children> tag controls the merging of the bound tag's explicit content (if any) with the binding <content> tag's anonymous content (if any). The <children> tag can appear anywhere inside the anonymous content of the binding. It represents any explicit content that the bound tag has. It supports the following special attribute:

```
includes
```

The includes attribute can be set to a comma-separated list of tag names. No namespace qualifiers are required. This attribute modifies the content merge process. What content is merged depends on what content exists. Here is an overview of the possibilities:

1. If there is no explicit content, then any <children> tags in the anonymous content are ignored. This is equivalent to deleting all start, end, and singleton <children> tags from the <content> section but leaving other tags and subtags intact. The final content is a copy of all of the remaining anonymous content.
2. If there is no anonymous content, then the bound tag will contain no content at all after all merging is done.
3. If there is both explicit and anonymous content, then count up the explicit content tags that are immediate children of the bound tag. Each of these tags should match one <children> tag. If this is achieved, then the final content will be a merged copy of all explicit and anonymous content.
4. If none of these three conditions is met, the <children> tag has been used incorrectly in the anonymous content, or unexpected explicit content is present in the bound tag. The final merged results will be unpredictable in this case.

The last two cases require further explanation. How content is merged depends entirely on the <children> tag. This tag creates the illusion that explicit content is added into the anonymous content. In reality, the final merged content is a copy of all the other content. It is convenient and harm-

less, however, to think of the explicit content as being added to the anonymous content. Here is how such additions can occur:

1. If a `<children>` tag appears without an `includes` attribute, then it stands for all the explicit content, which then replaces the `<children>` tag and its content subtree in the anonymous content. In this case, exactly one `<children>` tag should appear in the anonymous content.
2. If a `<children>` tag appears with the `includes` attribute, then it stands for some of the explicit content only. That explicit content will replace `<children>` and its content subtree in the anonymous content. All immediate children of the bound tag with tag names that match the `includes` list will be inserted. They will be inserted at that `<children>` point in the anonymous content. They will be inserted in the same order they appear in the bound tag.
3. If point 2 is repeated carefully, enough `<children includes=>` tags will exist to cover all the immediate child tags of the bound tag. Such tags provide complete coverage of explicit tags. In that case, all the explicit and anonymous content will be merged neatly.
4. If point 2 is not repeated enough, then some immediate child tags of the bound tag will not find a matching `<children includes=>` tag. That is incomplete coverage of explicit tags. All explicit tags are appended to the final content by Mozilla, and all anonymous content is thrown out.
5. If point 2 is done carelessly, then some child tags of the bound tag will match more than one `<children>` tag. That is a mess and does not work. In this case, content output will be unpredictable.

In summary, the binding designer must either (a) make no assumptions about the explicit content of the bound tag and avoid using includes; (b) fully anticipate what explicit tags may be put into a bound tag and use includes everywhere; or (c) make an educated guess and rely on the XBL system outputting something that is reasonable.

Listing 15.4 shows well-formed examples of this system at work.

Listing 15.4 Source code for XBL content merging examples.

```
<!-- tag to be bound -->
<mytag>
  <image src="face.png"/>
  <description label="Smile"/>
  <image src="face.png"/>
</mytag>

<!-- Example 1: unknown explicit content -->
<content>
  <box>
    <children/>
  </box>
</content>
```

```
<!-- Example 2: known explicit content -->
<content>
  <children includes="description"/>
  <box>
    <children includes="image"/>
  </box>
</content>

<!-- Example 3: known but optional explicit content -->
<content>
  <children includes="image|description">
    <label value="No emoticon supplied"/>
  </children>
</content>
```

In this listing, the <mytag> tag is the tag to be bound. The other fragments are from three different bindings. Example 1 adds all the explicit content, regardless of what it is, into a <box>. Example 2 carefully places each type of explicit content in a certain spot. Example 3 repeats back the explicit content, but if none exists, a <label> is reported instead. Listing 15.5 shows the final content for each of these cases.

Listing 15.5 Generated content for XBL content merging examples.

```
<!-- Example 1: unknown explicit content -->
<mytag>
  <box>
    <image src="face.png"/>
    <description label="Smile"/>
    <image src="face.png"/>
  </box>
</mytag>

<!-- Example 2: known explicit content -->
<mytag>
  <description label="Smile"/>
  <box>
    <image src="face.png"/>
    <image src="face.png"/>
  </box>
</mytag>

<!-- Example 3: known but optional explicit content -->
<mytag>
  <image src="face.png"/>
  <description label="Smile"/>
  <image src="face.png"/>
</mytag>
```

If the original contents of <mytag> are made very different, Example 1 will always work, but Examples 2 and 3 might yield unexpected results.

Listing 15.6 shows an arrangement that will never work reliably because any explicit content <image> tags will match both <children> tags in the anonymous content.

Listing 15.6 Poorly formed anonymous content for an XBL binding.

```
<content>
  <children/>
  <box>
    <children includes="image"/>
  </box>
</content>
```

15.2.5.3 Merging Bound Tags

The anonymous content inside the XBL <content> tag can include tags that are in turn bound to some XBL binding. This is just as well because most XUL tags are bound tags.

XBL automatically supports bound tags as anonymous content. Such tags are treated like any other tag. Just use the bound tag as if no binding were involved.

It is possible to create containment and inheritance cycles between one or more bindings. This should be avoided because it doesn't work. The <content> tag should never contain an instance of the tag to be bound.

15.2.6 <implementation>

The <implementation> tag is a simple container tag. It specifies the object-like interface of the binding. This interface is implemented inside the binding using JavaScript. It is then available to JavaScript scripts that run outside the bound tag. The interface created appears as properties and methods on the bound tag.

The <implementation> tag has one special attribute:

`implements`

This attribute can be set to a comma- or space-separated list of XPCOM interface names, like `nsISimpleEnumerator` or `nsIDOMEventListener`. Any interface name can be used. The Mozilla Platform treats the binding as an XPConnected (JavaScript wrapped) XPCOM component and uses these interface names to identify what the binding can do. It is up to the binding creator to make sure that the binding faithfully implements the interfaces it advertises. The XBL system automatically adds functionality equivalent to the `nsISupports` interface.

If any of these interfaces are specified and implemented, they appear as interfaces on the bound tag.

The `type` attribute is not supported by <implementation> or any of its subtags. The language used to define the content is assumed to be JavaScript. Use of <![CDATA[ ]]> XML syntax is recommended in all binding code when scripts are nontrivial in size.

The <implementation> tag can hold any number of these tags as children, in any order:

```
<field> <property> <method>
```

The <implementation> tag can also hold up to one of each of these tags:

```
<constructor> <destructor>
```

All five of these tags hold JavaScript scripts as content. Such scripts have access to several well-known JavaScript properties:

```
this window document event
```

The window and document properties are the same as those used in XUL and HTML scripts and refer to the window and document of the bound tag. The this property references the DOM Element object of the bound tag, which contains many useful DOM methods, like getAttribute(). The event property exists only for <handler> code and contains a DOM Event object.

Be aware that within an XBL <implementation> section, the this property has a special feature. All simple values assigned to properties of the this object are automatically turned to String types. Under that rule, the following line of code returns the string "1213":

```
this.dozen = 12; alert(dozen + 13);
```

This conversion shows up only when you use the + operator. For other mathematical operations, Strings will be silently converted to numbers and the behavior is hidden. A simple workaround is to store the values in an Object. This line of code reports 25:

```
this.d = {}; d.dozen = 12; alert(d.dozen + 13);
```

A binding without an <implementation> tag is still useful. It can display content and it can handle events.

Listing 15.7 is a simple example of a partial XBL object. It implements an experimental light bulb. Every time this light bulb's power output is checked, it ages slightly and its power output drops as a consequence. Eventually, trying to turn it on or off won't work at all because it becomes too inefficient.

Listing 15.7 Example object interface using XBL <implementation>.

```
<implementation>
  <field name="rating" readonly="true">60</field>
  <field name="lit">true</field>
  <field name="age">0</field>
  <property name="power"
            onget="age++; return rating-age/1000;"
            onset="age=(rating-val)*1000;"
  >
```

```
  <method name="toggle">
    <body>
      if ( this.power > 30 ) this.lit = !this.lit;
    </body>
  </method>
</implementation>
```

This object has four properties: `rating` (a constant value, intended here it is Watts), `lit` (a boolean), `age` (an integer), and `power` (a dynamically calculated value). It has one method: `toggle()`. These property names appear both as attribute values in XBL tags and as JavaScript code within the tag content. For example, the `toggle()` method uses the `power` and `lit` properties, whose names also appear as values of the `name` attribute in `<field>` tags.

The interface in Listing 15.7 is equivalent to the JavaScript object in Listing 15.8.

Listing 15.8 Equivalent JavaScript object to Listing 15.6.

```
var bulb = {
  const rating : 60,
        lit : true,
        age : 0,
    get power() { age++; return rating-age/1000; },
    set power(val) { age=(rating-val)*1000; },
        toggle : function () {
          if ( this.power > 30 ) this.lit = !this.lit;
        }
};
```

The two listings are obviously very similar. Why not just use a plain JavaScript object and avoid XBL? Such an object must be hand-attached to every tag that needs it, using a script or scripts. Using XBL, the object appears everywhere the CSS binding rule says it should, automatically.

15.2.6.1 `<field>` The `<field>` tag is the simplest part of the binding interface. It is designed to hold a simple variable. It is used to hold programmer-specified state information, for each bound tag. The `<property>` tag is a more flexible version of `<field>`.

`<field>` has two special attributes:

```
name readonly
```

- The `name` attribute holds the name of the JavaScript property that the field implements. Therefore, it must be a valid JavaScript variable name.
- The `readonly` attribute can be set to `true`, in which case a field cannot be set from a script.

The syntax for `<field>` is

```
<field>JavaScript expression</field>
```

Note that the content is an expression, not a statement. Because the expression is evaluated once only, it is usually a constant expression, but it can be a complicated calculation if required. This evaluation occurs when the binding is bound. Examples include

```
<field name="count">3+2</field>
<field name="address" readonly="true">"12 High St"</field>
<field name="phonelist">[]</field>
```

These examples look strange because there is no obvious assignment taking place. Instead, the XBL system evaluates the supplied expression and uses the result. These tags are similar to these JavaScript statements:

```
var   count     = eval('3+2');           // 5
const address   = eval('"12 High St"'); // a string
var   phonelist = eval('[]');            // an empty Array
```

Once created, fields are properties of the bound tag's object, which is the same as this, and can be used normally.

15.2.6.2 `<property>`, `<getter>`, and `<setter>` The `<property>` tag is designed to hold a simple variable, just as the `<field>` tag does. This variable is available as a JavaScript property on the bound tag's DOM object.

The difference between `<property>` and `<field>` is that the `<property>` tag's variable acts like an interface, whereas the `<field>` tag's variable is a simple value.

The `<property>` variable's value can be dynamically calculated when it is either read or written, and either operation can have side effects. This means that scripts using the property can cause other processing to occur just by setting or getting it.

Recall from Chapter 5, Scripting, that a JavaScript property can have its state defined with `__defineGetter__` and `__defineSetter__` functions. A `<property>` tag's property works the same way, except that it is defined with `<getter>` and `<setter>` tags, or with shorthand `onget` and `onset` attributes.

The `<property>` tag has four special attributes:

```
name readonly onget onset
```

- `name` is the name of the JavaScript property that this property defines. It must be a legal JavaScript name.
- `readonly` ensures that the property acts like a const JavaScript variable. It can be read but not written.
- `onget` is a shorthand notation for the `<getter>` tag and takes a script as its value. If both `onget` and `<getter>` are specified, then `onget` is used.
- `onset` is a shorthand notation for the `<setter>` tag and takes a script as its value. If both `onset` and `<setter>` are specified, then `onset` is used. `onset` is useless if `readonly="true"` is also specified.

The <property> tag can hold zero or one <getter> and <setter> tags as content. In turn, the content of these tags is the body of a JavaScript function.

- The <getter> tag and the onget attribute must contain a sequence of JavaScript statements that result in return being called.
- The <setter> tag and the onset attribute must also contain JavaScript but should return val. The special variable val contains the value passed in to the setter.

See Listing 15.7 for a simple example of onget and onset. The equivalent syntax using <getter> and <setter> is shown in Listing 15.9. The CDATA sections could be dropped in this example because the code is trivial. The keyword this is used everywhere for clarity—a recommended practice.

Listing 15.9 Example of XBL <getter> and <setter> use.

```
<property name="power">
  <getter><![CDATA[
    this.age++;
    return this.rating - this.age/1000;
  ]]>
  </getter>
  <setter><![CDATA[
    this.age = (this.rating - val) * 1000;
    return val;
  ]]>
  </setter>
</property>
```

If an attempt is made to set such a property when both <setter> and onset are missing, then an error will be reported to the JavaScript console (or an exception will be raised). If an attempt is made to get the value of such a property when both <getter> and onget are missing, then undefined will be returned.

15.2.6.3 <method>, <parameter>, and <body> The <method> tag is used to define an interface method. It contains zero or more <parameter> tags followed by exactly one <body> tag. The <method> tag supports one special attribute:

name

The name attribute specifies the name of a JavaScript property that will hold a Function object, so it must be a valid JavaScript identifier. The <method> tag does not support the action attribute and Mozilla may crash if that attribute is used.

If inheritance chains are being used, then there is no way for any extended binding <method> to run any base binding <method> that has been

overridden by an extended binding <method> with the same name. You cannot work with overridden base-binding methods.

The <parameter> tag has exactly the same syntax as the <method> tag. Its sole name attribute defines one variable name passed in to the methods. The order of <parameter> tags is the same as the order of variables passed to the method.

The <method> and <parameter> tags are not used to create a fully typed function signature. They are just used as names. There is no type checking, or even argument counting. There is no type reflection beyond what JavaScript itself provides. There is no way to specify the type of the return value. These tags are very simplistic.

It is not possible to create two methods with the same names but different numbers of parameters. Because all JavaScript functions support variable argument lists, there is also no need to create such variations.

Finally, the <body> tag contains the JavaScript statements that make up the method. This tag has no special attributes at all. The JavaScript return statement should be used in the code when the method has something to return. The arguments object, which appears in all JavaScript functions, is also available for use in the content of the <body> tag.

Listing 15.10 shows these tags working together:

Listing 15.10 Example of XBL <method>,<parameter>, and <body> tags.

```
<method name="play">
  <parameter name="boy"/>
  <parameter name="dog"/>
  <parameter name="ball"/>
  <body><![CDATA[
    if ( arguments.length != 3)
      throw Components.results.NS_ERROR_INVALID_ARG;

    if ( boy == "Tom" && dog == "Spot" )
    {
      return document.fetch(ball.type);
    }
    return null;
  ]]>
  </body>
</method>
```

This example shows that any checks on the arguments passed in must be done by hand; nothing is done for you automatically. JavaScript's throw statement may be used to terminate the method and return an exception to the calling code. The value returned in this example is one of XPCOM's official error values. By using one of these constants, the method is conforming to XPCOM component standards. The method body is free to interact with any objects that it likes in the current window—in this case, the fetch() method is a method or function stored outside the binding. This piece of XBL is equivalent to the JavaScript function shown in Listing 15.11.

Listing 15.11 JavaScript function equivalent of an XBL <method>.

```
function play(boy, dog, ball)
{
  if ( arguments.length != 3)
    throw Components.results.NS_ERROR_INVALID_ARG;

  if ( boy == "Tom" && dog == "Spot" )
  {
    return document.fetch(ball.type);
  }
  return null;
}
```

The two definitions are hardly different at all, and so it is no surprise that XBL <method> contents are converted to JavaScript soon after an XBL document is loaded and parsed.

15.2.6.4 <constructor> and <destructor> The <constructor> and <destructor> tags are event handlers that fire once only during the bound lifetime of a binding. They are used for initialization and cleanup activities, just as constructors and destructors in most object-oriented languages are. XBL constructors and destructors have standard object-oriented semantics and do not follow the prototype system of JavaScript.

The only special attribute that these two tags support is this attribute:

```
action
```

The action attribute is shorthand for the content of the <constructor> and <destructor> tags. That content is a JavaScript function body, as for the XBL <body> tag. It can be specified between start and end tags or by using the action attribute. If both are specified, the action attribute is used, but it makes no sense to specify both.

The <construction> tag runs its action each time the binding is bound to a target document tag. That could be a large number of times, and if so, it is important that the constructor's action has good performance. There is no way to control the bindings in the inheritance chain using the <constructor> tag. There is no way to control any bound tags in the <content> section from the <constructor> tag. The constructor code is passed no arguments, but the this pointer is always available. The construction code does not need to return a value.

If the current binding inherits another binding, then <constructor> tags will fire in order from the tail to the head of the inheritance chain—in other words, the <constructor> tag of the primary binding will be the last one to fire.

The <destructor> tag runs its action each time the binding is unbound from a target document tag. It is rare that this occurs without the whole target document being destroyed, but if bindings are managed by hand, or if the

state shared by multiple windows is maintained, then a <destructor> may be useful. As for the <constructor> tag, there is no special way to affect the status of other bindings that are part of the current binding. The destruction code is not passed any arguments and does not need to return any value.

If the current binding extends (inherits from) another binding, then <destructor> tags will fire in order from the head to the tail of the inheritance chain—in other words, the <destructor> tag of the primary binding will be the first one to fire.

It is common to create XPCOM objects from a constructor. If only one such object is required for a given binding, rather than one per bound tag, then a common idiom for arranging this is shown in Listing 15.12.

Listing 15.12 Creation of global components using an XBL constructor.

```
<constructor><![CDATA[
if (!document.globalPicker)
{
  var Cc = Components.classes;
  var Ci = Components.interfaces;
  var comp = Cc["@mozilla.org/filepicker;1"];
  document.globalPicker = comp.getService(Ci.nsIFilePicker);
}
]]></constructor>
```

In this code, each bound tag checks to see whether the initialization has been done yet, and if it has, the code simply ignores that step. Bindings are all bound at different times, but because Mozilla is single-threaded, it is not possible for two constructors to run at the same time.

15.2.7 <handlers>

The <handlers> tag is a simple container tag that holds all the event handlers associated with the XBL binding. It has no special attributes of its own. It holds one or more <handler> tags. Each handler tag is responsible for executing some JavaScript in response to a single DOM 2 event on the bound tag.

The XBL bindings stored in Mozilla's chrome contain over 500 individual handlers and so are responsible for a great deal of interactivity. Many of the events processed by standard applications like the Classic Browser and Classic Mail & News are the result of XBL bindings rather than specific application code.

The event handlers specified in XBL are one of four sources of handlers in Mozilla. Chapter 6, Events, describes support for on... style event handlers. It also describes the <key> tag. That tag has syntax very similar to the <handler> tag. The last set of handlers is buried deep inside the Mozilla Platform. Those handlers are written in C/C++ and are installed automatically by the platform when a window first opens.

These last handlers are the default handlers that run if nothing else is installed. Just as the Mozilla Platform precreates a special command controller for handling the focus, so too does the platform install a set of special handlers for the most fundamental types of event processing. When <key> handlers are added, they sit on top of the fundamental handlers, just as application-defined onclick handlers sit on top of the XBL bindings. When <handler> tags are added, they sit alongside the fundamental handlers, as if both were registered with addEventListener().

Although this last and most hidden set of handlers is implemented as part of the XBL system, it is not accessible to application programmers. The main thing to note is that some event handling occurs underneath XBL bindings, and that no amount of searching in the chrome will find code that is responsible for it all.

15.2.7.1 <handler> The <handler> tag is very much like the XUL <key> tag. It defines one event target and one action for one highly specific event. An action is either a piece of script or a command to execute.

The <handler> tag implements the default action for the given event when it occurs on the bound tag. It can be overridden on the bound tag with a normal DOM 2 Events event handler.

For the handler to run, the matching event must be generated on the bound tag. That means the event must be generated either by a user action, as a consequence of document changes, or via the DOM 2 Events dispatchEvent() method. There is discussion of XBL event flow in "Scripting" later in this chapter.

The <handler> tag supports the following special attributes:

```
event phase command modifiers clickcount key keycode charcode button
     action
```

- The event attribute specifies the event name for the handler, drawing from the events that Mozilla supports. See Table 6.1 for a list of event names. mousedown is an example event. event has no default value.
- The phase attribute says at what stage in the DOM 2 Event cycle the action will fire. It must be set to "capturing", "target", or "bubbling". "bubbling" is the default.
- The command attribute specifies a command (as described in Chapter 9, Commands) that will be executed when the event occurs. The controller used to find and execute the command, however, will be sought on the currently focused tag, not on the bound tag. This means that the binding must be entirely focusable; otherwise, steps must be taken to ensure that the focusable parts of the binding content have a suitable controller. This attribute is mutually exclusive with action. It has no default value.

- The `modifiers` attribute accepts a space- or comma-separated list of the values `shift`, `control`, `alt`, `meta`, `accel`, and `access`. This attribute is used to specify which modifiers apply to a key event. All stated modifiers must be pressed for the event to occur. `accel` is the generic, platform-independent accelerator key as described in Chapter 6, Events. `access` matches the case where a shortcut key (a hot key) for the bound tag has been pressed. Shortcut keys are specified with the XUL `key=` attribute. The default is no modifiers.
- The `clickcount` key is only used for mouse events. It contains a single digit that specifies how many user clicks of the mouse are required (down-up is one click). The maximum number of clicks is ultimately an operating system limit. Mozilla supports at least three clicks. The default is any number of clicks.
- The `key` attribute is used only for key events. It contains a single printable key, such as '`A`'. Either `key` or `keycode` should be used for key events, but not both. The default is any key press.
- The `keycode` attribute is used only for key events. It contains a single `VK_` key mnemonic from the list in Table 6.3. Either `keycode` or `key` should be used for key events, but not both. The default value is any key.
- The `charcode` attribute is an older, deprecated name for the `key` attribute. Use `key` instead.
- The `button` attribute is used only for mouse events. It contains a single digit that specifies which mouse button is pressed. The number of mouse buttons is ultimately an operating system limit; Mozilla supports at least three buttons. The first button is button 0. The default button is any button. On a two- or three-button mouse, the right button is button 2.
- The `action` attribute contains the script that should run when the event occurs. This attribute is mutually exclusive with the `command` attribute. It has no default value.

If neither the `command` nor the `action` attribute is specified, then a JavaScript event handler for the event can be put between start and end `<handler>` tags. An example handler is

```
<handler event="click" phase="target" clickcount="1" button="0">
  this.do_custom_click(); // a <method> of the binding
</handler>
```

This is the default action for a left-button single-click click event. The `do_custom_click()` method is specified elsewhere in the binding so that it can also be called directly on the bound tag's DOM object.

15.2.8 Non-tags and Non-attributes

The XBL system has its share of tags that don't exist, but appear to exist. The following summary applies to Mozilla versions to 1.4.

The W3C Note for XBL is a proposal and does not exactly match Mozilla's implementation. Mozilla's version of XBL does not support the following aspects of that Note:

- The `type` attribute, used on `<bindings>`, `<method>`, and other tags.
- The `applyauthorsheets` attribute on the `<content>` tag.
- The `applybindingsheets` attribute on the `<children>` tag.
- The `<element>` tag and all references to it.
- The following events are not supported: `contentgenerated`, `contentdestroyed`, `bindingattached`, `bindingdetached`.

These two attributes come from the early development of XBL and should be avoided: the `includes` attribute when used on the `<content>` tag and the `charcode` attribute when used on the `<handler>` tag.

The `<script>` tag is not part of XBL and won't work at all inside a pure XBL document.

The following tag attributes do nothing at all:

```
attachto applyauthorstyles styleexplicitcontent
```

15.3 COMBINING MULTIPLE BINDINGS

The XBL inheritance system is given an overview in "Object-like Features" earlier in this chapter. This topic is concerned with uses of the inheritance system. Only a binding creator can implement inherited bindings.

15.3.1 Simple Inheritance with `extends=`

Listing 15.13 show two bindings related by inheritance.

Listing 15.13 Trivial example of XBL binding inheritance.

```
<!-- both bindings contained in example.xml -->

<binding id="smileyA">
  <content>
    <xul:image src="faceA.png"/>
  </content>
  <implementation>
    <method name="methodA">
      <body> return true; </body>
    </method>
  </implementation>
</binding>

<binding id="smileyB" extends="example.xml#smileyA">
  <content>
    <xul:image src="faceB.png"/>
```

```
  </content>
  <implementation>
    <method name="methodB">
      <body> return false; </body>
    </method>
  </implementation>
</binding>
```

Table 15.2 describes how different parts of a binding are inherited, so following that table yields these results.

If binding `smileyA` is bound to a XUL tag, then that tag will have a single `<image src="faceA.png">` tag as content, and one method called `methodA()`. In this case, the bound tag's XBL inheritance chain is of length one and contains only binding `smileyA`. This case has no inheritance, and `smileyA` is the primary binding.

If binding `smileyB` is bound to a XUL tag, then that tag will have a single `<image src="faceB.png">` tag as content, and two methods named `methodA()` and `methodB()`. In this case, the bound tag's XBL inheritance chain is of length two and contains binding `smileyB` at the head and binding `smileyA` at the tail. This is the inherited case, and `smileyB` is the primary binding.

Both of these bindings can be applied to different tags at the same time.

It is impossible to tell whether a binding is extended because its URL might be used anywhere in the world. If a binding resides in the chrome, then an audit of all the chrome files can reveal all uses of a given binding. Even so, secure applications installed outside the chrome might still extend a chrome binding undetectably.

15.3.2 Zero Content Bindings

A very popular XBL technique used in Mozilla is to extract common logic from a set of bindings and build a zero content binding to hold that logic. Such a binding has no `<resources>` or `<content>` section and is rarely (or never) bound directly to a tag. It contains mostly programming logic and state information.

If another binding uses this binding as its base, then the methods and handlers of the zero content binding are added back into the extended binding at run time. The zero content binding can be extended a number of times for different purposes. In object-oriented terms, the zero content binding might be called a "virtual content base class."

An example of such a zero content binding is the `button-base` binding in `button.xml` in `toolkit.jar` in the chrome. It contains:

- properties: `accessible type dlgType group open checked checkState autoCheck`
- handlers: `event="command"`

These properties and events are responsible for managing a number of states used by button-like widgets. For example, the `checked` property has `<setter>` and `<getter>` JavaScript logic that allows a bound tag to have `<checkbox>` or `<radio>`-like states. This logic does housekeeping tasks on the bound tag, setting and unsetting XML attributes and coordinating the checked state against other attributes that might be present.

The logic in this one binding is inherited by many other bindings. Those bindings in turn are used for many XUL tags: all variants of `<button>`, all variants of `<toolbarbutton>`, `<thumb>`, `<dropmarker>`, `<radio>`, `<menu>`, the buttons in `<wizard>`, `<dialog>`-based windows, and so on. Clearly the `button-base` binding is highly reusable.

A zero content base tag saves unnecessary duplication of code. When such a binding is used, the binding extending it should add a `<content>` section.

15.3.3 Input Façade Bindings

It is possible to make an XBL binding that concentrates on user input. Such a binding has a large `<handlers>` section and not much else, although it might also have supporting methods and properties.

Such a binding is called an input façade binding for two reasons. First, it adds a layer of user input and event servicing between the user and the widget, in the form of a set of handlers. That is like the front counter of a shop. Second, it relies on some other binding to supply it with the widget-specific processing power it needs. It is therefore a "thin" binding, and façades are thin. A trivial example of an input façade binding is

```
<binding id="click-facade" extends="bindings.xml#base-widget">
  <handlers>
    <handler event="click"> this.click(); </handler>
  </handlers>
</binding>
```

This simple binding erects an input façade between single-click input and the widget. It does not implement the `click()` method—that is left up to some other binding. That method might be generic (in which case the `base-widget` binding would supply it), or it might be highly specialized (in which case a binding that extends `click-facade` will supply it).

The handler code in this binding could be more complex; it might be required to maintain state that enables and disables certain inputs depending on what the last input was. An input façade is another place to put macro-processing functionality for user input.

Like a zero content binding, an input façade binding is rarely bound to a tag. Its main purpose is to extract from a set of bindings common user input semantics and collect those semantics together into one place. Its lack of `<content>` means that it should generally be extended before use.

There are no XBL input façades in the Mozilla Platform (to version 1.4), but there is room for them. If you examine the XBL bindings for <tree>, <listbox>, <tabbox>, and HTML's <select> (start with xul.css in toolkit.jar in the chrome), then you will see a common set of keypad navigation keys in all the bindings for those widgets. This subset could be extracted into an input façade binding that would then be reused across widgets, providing a uniform navigation experience.

Mozilla does currently implement one kind of input façade. It consists of the XUL documents that hold sets of <key> bindings. These documents are included in most Mozilla application windows via the use of overlays. Where an XBL input façade is specific to an XBL binding, a XUL input façade in the form of a set of <key> bindings is specific to a whole XUL document.

15.3.4 Inner Content Bindings

This chapter's title implies that bindings are used to create whole widgets, and that can indeed be done. It is also possible for a binding to supply the insides of a widget only. Such a binding is called an inner content binding.

Inner content bindings are simple to create. Their <content> section makes an assumption about the parent tag of the bound tag. The expectation is that the bound tag will deliver content to its parent tag, which is some kind of container tag. That container tag defines the borders of a widget. The inner content binding relies on that parent tag being correctly stated. Obvious containers in XUL are tags like <box>, <button>, <menupopup>, <scrollbar>, and <toolbar>.

Inner content bindings are dependent on the bound tag's parent and are closely coupled with it. Such a binding might provide methods and properties that the parent expects, or it might reach up into the parent to extract information that it needs. This close coupling means that the bound tag is usually an ugly failure when stated by itself. When stated inside the right container tag, it provides an abstraction that expresses the content of that tag neatly.

Inside Mozilla's XUL, several tags follow the inner content philosophy. The tags used inside <scrollbar> are examples: <thumb>, <slider>, and <scrollbarbutton>. These tags do not all have XBL bindings, but they do rely on their parent tags for correct functioning. Examples of tags with XBL bindings are the <dropmarker> tag used in some <button> tags and the <tabs> tag used inside <tabbox>.

15.4 How Bindings Are Processed

Here are the steps a binding goes through when it is bound to a tag in a target document:

1. When loading the target document, the Mozilla layout engine detects the need for a binding when it sees a -moz-binding style on a given loaded tag.

2. The document's list of things to finish is incremented by one. This list delays the firing of an `onload` event for the target document.
3. If a complete, compiled copy of the binding does not yet exist for the target document, steps 4-7 are done. If it exists, processing continues from step 8.
4. The XBL document for the binding is fetched and put in the Mozilla cache. This is done in parallel with the loading of the target document.
5. The fetched XBL document is parsed and compiled into an internal data structure.
6. If an `extends` attribute exists, steps 2–7 are repeated (in parallel) for each binding in the inheritance chain. So another binding could reach step 7 before the current binding.
7. The compiled binding is added to the document's binding manager. The binding manager is an XPCOM component that holds a list of all bindings used by the document.
8. The full inheritance chain for the binding is now available in loaded and compiled form. The target document's tag is now bound to the binding—ultimately this is just a pointer assignment inside Mozilla. Before this step, a script inspecting the bound tag would see no binding. After this step, a script cannot inspect the bound tag until all the remaining steps are complete. This is because the remaining steps occupy Mozilla's sole processing thread until they are finished.
9. Attributes of the `<content>` tag are copied to the tag to be bound.
10. Explicit and anonymous content are merged, and the result is copied to the bound tag using DOM operations.
11. `<handler>` handlers are installed on the bound tag using the binding manager.
12. `<property>` and `<method>` properties are added to the bound tag's DOM object.
13. Any `<constructor>` code in the inheritance chain is run.
14. If there are any stylesheet changes resulting from all these steps, they are applied as a final step.
15. The list of things to finish before an `onload` event can fire is decremented by one.

15.5 SCRIPTING

A bound tag looks like any other DOM object to JavaScript. Scripts originating from the bound side of the tag can act on the bound tag's object in a number of standard ways:

- Properties can be invoked as methods, or assigned to, or read.

- Properties can be added or set to `null`.
- Properties that hold methods (`Function` objects) can be changed to user-defined methods.
- Event handlers lodged with `on...` style attributes, or via the DOM 2 `addEventListener()` method, can override handlers in the binding.
- DOM 1 interfaces can set and unset attributes on the bound tag.
- DOM 1 interfaces can modify the final merged content of the bound tag.

None of these actions affects the XBL binding specification for a bound binding.

The most significant issue for a user of XBL bindings is the issue of load time. There is no general way to tell whether a single bound tag has finished being bound. The only way to be sure is to use an `onload` handler in the bound tag's document and to avoid lazy-loading templates. If the bindings you use are all custom made, then it is good design for those bindings to provide some hint that they are loaded.

Another aspect of scripting bound tags has to do with event handling. There is the matter of ordinary DOM events and the special case of input focus.

Ordinary DOM events travel into the bound tag's final merged content as if it were normal XUL content. From an application programmer's point of view, a bound tag is a single indivisible item and the last step in the capture phase of the event's life. It is only from the binding creator's point of view that the content of a bound tag has any active role. In that case, the `event` continues into the binding where it can be accessed with the event object. The `event.originalTarget` property of that object is of most use in the binding; the `event.target` property is not yet reliable at version 1.4.

If, however, the bound tag is not part of the window's focus ring (such as a XUL `<box>`) tag, then parts of the merged content can still receive events. In particular, if the merged content contains tags that are focus ring candidates (like `<checkbox>` or `<button>`), then those tags can still be tabbed into and out of by the user. The following principle is at work:

If a tag ought to be able to receive the focus, and its parent tag can't receive the focus, then let it receive the focus.

The XBL binding system does not have a "refresh" or "rebuild" interface that can be used to bring a bound tag up to date. A bound tag can only have a whole binding added or removed. There are two ways to do this.

The first way to change a bound tag's binding is to modify the `-moz-binding` style property with a script. The most straightforward thing to do is to set this property to or away from the special value `none`. Unfortunately, a Mozilla bug makes this solution unreliable. The recommended alternative is to create two style classes:

```
.binding-installed { -moz-binding : url("test.xml#Test"); }
.binding-removed   { -moz-binding : none; }
```

Rather than set the `-moz-binding` property, set the class attribute of the bound tag to one or the other of these two class names. This works properly.

The second way to change a bound tag's binding is to use the sole XPCOM interface available for the purpose. This interface is

```
nsIDOMDocumentXBL
```

It is automatically available on the document object. Table 15.3 lists the methods of this interface.

Table 15.3 `nsIDOMDocumentXBL` interface

Returns	Method signature
void	addBinding(nsIDOMElement element, String URL)
void	removeBinding(nsIDOMElement element, String URL)
nsIDocument	loadBindingDocument(String URL)
nsIDomElement	getBindingParent(nsIDOMNode node)
nsIDOMNodeList	getAnonymousNodes(nsIDOMElement element)
nsIDOMElement	getAnonymousElementByAttribute(nsIDOMElement element, String attribute, String value)

The `addBinding()` and `removeBinding()` methods add and remove a primary binding to the tag matching the DOM object supplied. If the primary binding implies an inheritance chain, then that is loaded as well. These two methods operate synchronously, so they don't return until the binding change is complete.

The `loadBindingDocument()` method synchronously loads an XBL document with the specified URL into a DOM tree of its own, which is returned. This method can be used to inspect the structure of a binding specification. This method is useful for reflecting the specification of a binding into a data structure, which can then be used to build schema-based tools like designers and inspectors. It also provides a way to change a binding synchronously—that is, to chance a binding so that script execution suspends until the new binding is fully in place.

The `getBindingParent()` method returns the bound tag that has the supplied object as a content tag or text node. It is used to walk up the hierarchy of content from a piece of content is whose origin is uncertain, until a tag that has a binding is found.

The remaining two methods are XBL variations on the `document.getElementById()` method. Instead of returning tag objects from the document DOM tree, they return tag objects from an XBL binding's anonymous content. This is the only part of an XBL binding's specification that can

be accessed from a script. In both methods, the DOM Element passed in is the DOM object for a bound tag.

`getAnonymousNodes()` returns an object containing all the immediate children of the `<content>` tag in a matching binding. The number of children is specified by a length property on the returned object, and each child is available using the `item()` method on the returned object, which takes a child index number.

`getAnonymousElementByAttribute()` returns a single tag from the set of tags inside the `<content>` tag of a binding. The tag returned has the attribute name and attribute value specified in the method call.

This last method solves a problem created by the XBL merging of anonymous and explicit content in a bound tag. Before the merge, anonymous content tags are in a certain order. After the merge, the relative positions of those tags may be different, owing to the addition of explicit content. This method allows an anonymous content tag to be found, regardless of its position. The tag must have a unique attribute-value pair that can be searched for. The attribute named `anonid` is a convention used for this purpose. The DOM Element supplied should be that of the bound tag.

A second problem solved by this method has to do with themes. A theme may contain more than style information; specific skins may also include scripts. Those scripts might customize the content of a given widget binding so that it has a look consistent with the theme. `getAnonymousElementByAttribute()` provides a way for a script-enabled skin to poke around inside an existing binding, or its own replacement binding, and make changes.

The XPCOM system has two components that are related to XBL:

```
@mozilla.org/xbl;1 @mozilla.org/xbl/binding-manager;1
```

Neither of these have any interfaces that are significant for application scripts.

15.6 Style Options

The `-moz-binding` CSS2 style extension is the only style option specific to XBL. See the topic “Bound Tags and `-moz-binding`” in this chapter.

Each binding created should have its own stylesheet so that the designed widget can support multiple themes or skins. Such a stylesheet is specified in the `<resources>` section of the binding specification.

15.7 Hands On: The `<noteplacer>` Tag

This “Hands On” session is about developing general-purpose code using XBL. In particular, we will create a new XBL binding and matching user-defined XUL tag that might contribute to the content and function of the NoteTaker dialog box.

XUL content should not be converted to XBL bindings just because it is possible. Only code that is used more than once in a given page or that could be reused in another application is a good candidate for XBL. XBL bindings wrap up the content they implement in extra structure. For mostly harmless content, that extra structure is not always desirable from a code maintenance point of view. Only when the binding is used frequently are there complexity savings.

In the NoteTaker tool, there are no repeated parts in the content of the toolbar or dialog box: therefore, there is little need for XBL use. We do, however, need an XBL example to illustrate the technology. The positioning information (top, left, height, width) of the Edit pane in the dialog box is worthy of experiment. Those four numbers are very similar to the positioning styles in the CSS2 standard, and a need for them may crop up in other applications. Furthermore, the existing implementation is a little cumbersome: The user must make an educated guess when entering values, and there is no user feedback on those values until the dialog box is closed, which is too late.

We can improve on that existing positioning system with a new widget. We will create an XBL binding that the user can manipulate graphically to specify the location of a note. This binding will be a special-purpose widget that is a bit like the <colorpicker> tag and a bit like a Print Preview window.

This positioning widget won't be integrated into the final NoteTaker code; we'll just illustrate the possibilities.

15.7.1 Designing Widget Interfaces

We need to find visual, XML, JavaScript, and user input interfaces for the new binding and implement those interfaces in XBL code. The job of the widget is to cue the user to provide positioning information and to make available that information to the rest of the XUL document.

We first look at the visual interface. The visual part of the binding is the inspiration for the widget. That visual must come from creative insight and invention because every widget attempts to provide something unique and different. This is the most challenging part of widget design because each widget must be original. If it's not original, we might as well just copy and enhance an existing widget like <button>.

There is one way to avoid being original. An XBL widget can aggregate other, simpler widgets into a useful group. Such an aggregated system is often called a console, just as an old-fashioned console consisted of a set of switches and knobs. That kind of use of XBL is straightforward.

Because all XUL tags and tag combinations take up a rectangle of screen area, we know that we must start with a rectangle. It's possible to create rather weird widgets that break this rule, but it's rare to go that far. Figure 15.2 shows a simple mockup of our new widget.

This idea came from looking at other software that provides visual configuration options, like print subsystems and image-processing programs, and then reflecting on our own need. The essential thing that is new in this widget

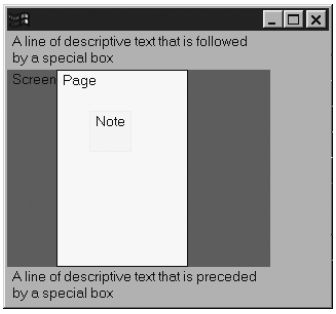


Fig. 15.2 Mockup of an XBL binding that lets the user position a note.

is that it is a fixed-sized model of the whole desktop. In the limited space we have here, we'll just do a simple-minded job of creating this widget. A more extensive solution would use technology similar to the Print Preview feature of the Mozilla Browsers.

In Figure 15.2, the main rectangle will be our new widget. It contains three parts: a backdrop <box> that represents the total screen area of the user's desktop, an optional white <box> that represents a browser window or page, and a yellow <box> that represents the note to be positioned. Each box is labeled in case it's not obvious to the user what they are. The whole rectangle will be sized in a way that reflects the dimensions of the desktop (the display resolution).

This main rectangle accepts user input. If the user clicks with the primary mouse button, the top-left corner of the note is located under the mouse. If the user clicks again, the bottom-right corner is set. This series of clicks can be repeated as long as is necessary. If the user clicks with the alternate mouse button (right-click or apple-click), then the nearest leading diagonal corner of the note is moved to the mouse location.

When the note is positioned, its coordinates are calculated. If the white page is not present, those coordinates are relative to the whole desktop. This case is useful if the user browses with windows maximized. If the white page is present, the coordinates are relative to the edge of the page, which is the edge of a window smaller than the whole screen. The white page is present for psychological purposes: human beings prefer windows that are displayed in the golden ratio, where width and height are in the approximate proportion 1:1.618. It is common for browser windows to end up in that ratio. The white page reminds the user what that ideal size looks like.

The second interface is user input. The widget will support button 0 single click (the primary or left mouse button) and button 2 single click (the secondary, apple-click or right-click button). Both buttons set the position of either the top-left corner or the bottom-right corner of the note. Button 0 clicks alternate between the two corners. Button 2 clicks move the nearest corner. After a quick glance at the visual interface, we should be able to create a widget that is nei-

ther a member of the focus ring or of the accessibility system, so there is no user input planning required there. Both possibilities are disqualified because none of the XUL tags in the binding's content is focusable or accessible.

The third interface required for this new widget is an XML interface. In our case, that means a tag name and a set of attributes that are known to the binding and layout behavior. A binding can be bound to any tag, but we'll select `<noteplacer>` as the tag of choice for the binding. The following attributes will have special meaning:

```
scale screenx screeny pageless
```

- `scale` accepts an integer and specifies the size of the noteplacer widget relative to the actual desktop size. For example, set to 4 (the default is 2), the noteplacer widget will be one-quarter the dimensions of the full desktop.
- `screenx` states the width of the desktop in pixels. The default is the value given by the `window.screen.width` property.
- `screeny` states the height of the desktop in pixels. The default is the value given by `window.screen.height` property.
- `pageless` can be set to `true` or `false`. If set to `true`, the white page will not appear in the widget. The default is `false`.

There is also the question of layout. Can the widget flex? What orientation does it have? The list of questions goes on. Because the noteplacer widget mimics the shape of the desktop, it will be a fixed size and won't flex. It will always be oriented horizontally and will have no particular alignment or direction.

The last interface will be the JavaScript XBL interface for the bound tag. We'll support the following properties:

```
top left width height scale screenx screeny pageless
     setFromEvent(e,primary)
```

- `top`, `left`, `width`, and `height` match the fields in the Edit dialog box.
- `scale`, `screenx`, `screeny`, and `pageless` match the XML attributes for the bound tag.
- `setFromEvent()` positions one of the corners of the note based on an event object. If the second argument is `true`, the top-left corner is set; otherwise, the bottom-right corner is set.

Offhand, we can't think of any existing platform interfaces (e.g., `nsIAccessible`) that we'd like to support. The bound tag will automatically gain the standard DOM interfaces for a DOM 2 Element object; we don't have to do anything to get those.

Together, these four types of interface fully specify our new widget. The final question we should ask is: Can we exploit any existing binding? It doesn't appear likely in this case because the noteplacer widget is so simple.

15.7.2 Adding XBL Content

To create the binding itself, we start with a simple skeleton. In this step, we'll also consider the content parts of the binding. The skeleton is shown in Listing 15.14.

Listing 15.14 Trivial example of XBL binding inheritance.

```
<?xml version="1.0"?>
<bindings id="notetaker"
   xmlns="http://www.mozilla.org/xbl"
   xmlns:xul="http://www.mozilla.org/keymaster/gatekeeper/
          there.is.only.xul"
   xmlns:xbl="http://www.mozilla.org/xbl">

  <binding id="noteplacer">
    <resources></resources>
    <content></content>
    <implementation>
      <constructor></constructor>
      <destructor></destructor>
    </implementation>
    <handlers></handlers>
  </binding>
</bindings>
```

The set of bindings specific to NoteTaker we've called "notetaker" and the binding specifically for the <noteplacer> tag we've called "noteplacer". Because there's no binding inheritance, there's no extends attribute on the <binding> tag. Because the new widget is based on simple XUL boxes, there's no need for a display attribute on that tag either. All we need to do is fill in the four sections of the binding.

We start with the <content> and <resources> sections. We mock up some content (the basis of Figure 15.3), and then consider how it might best be modified to suit the XBL system. The mocked-up content appears in Listing 15.15.

Listing 15.15 Mocked-up XUL content for the noteplacer XBL binding.

```
<stack minwidth="320" minheight="240">
  <description value="Screen"/>
  <box class="page" top="0" left="86" minwidth="148" minheight="240">
    <description value="Page"/>
  </box>
  <box class="note" top="20" left="106" minwidth="40" minheight="40">
    <description value="Note"/>
  </box>
</stack>
```

This content is of a fixed size and has many attributes hard-coded in place. Most widgets should rely on the platform rather than fixed pixel size to

lay out the content neatly. The unique thing about our widget is fixed dimensions, so that's the rule we break to add something new to XUL's widget set.

Before this code will be suitable for the <content> section of the binding, it needs some work. We see the following problems:

- The page part of the widget always appears; we want it to be invisible if the user uses full-screen windows.
- The page is very boring and plain white. We might want to put something more suggestive in there, like a browser menu bar and toolbar.
- The attributes quoting pixel sizes for most tags are useless. The real values will be computed based on <noteplacer> tag attributes and the current screen size.
- Font sizes are misleading. If the widget is supposed to represent a scaled-down desktop, then the font sizes should be scaled-down as well.

There are simple solutions to all of these problems. Page disappearance is the easiest. We'll let the <box> tag holding the page inherit the pageless attribute from the <notepicker> tag:

```
<box class="page" xbl:inherits="collapsed=pageless" ...>
```

To stop the page from being boring, we let any explicit content of the <noteplacer> tag dictate the contents of the <box class="page"> tag. By carefully reviewing the merging rules for explicit and anonymous content, we conclude that

```
<description value="page"/>
```

should be replaced with

```
<children>
  <description value="page"/>
</children>
```

If there is explicit content, it will all be added in one place; if not, then the <description> tag stands as the default.

We next remove the fixed pixel sizes. They were based on a widget that is the size of a 640×480 desktop, scaled down by a factor of two. Instead, values will be calculated at run time in the binding constructor. We can't inherit from the attributes in the <noteplacer> tag because the needed values are a mathematical combination of those attribute values, not a direct copy. We'll see shortly how that is done.

Finally, font sizes will be addressed in the constructor as well. We'll add an inline style to any anonymous content that is a <description> or <label> tag. That style will scale down the fonts to match the size of the widget. In some cases, the text will be unreadable, but it's only intended as a point of reference for the user—the widget is mostly graphical.

For these last two points, we'll add an anonid attribute to the anonymous content that we'll want to modify later. Looking ahead a bit, that is the

<stack> tag and two <box> tags. The final <content> part of the binding will be as shown in Listing 15.16. Note the xul: namespace prefix on every XUL tag.

Listing 15.16 <content> part of the noteplacer XBL binding.

```
<content>
  <xul:stack anonid="desktop">
    <xul:description value="Screen"/>
    <xul:box xbl:inherits="collapsed=pageless" class="page" anonid="page">
      <xul:children>
        <xul:description value="Page"/>
      </xul:children>
    </xul:box>
    <xul:box class="note" anonid="note">
      <xul:description value="Note"/>
    </xul:box>
  </xul:stack>
</content>
```

If we test the binding at this stage (a good idea in principle), the results will look odd. They look odd because we have stripped out quite a lot of the final layout information. The binding is working; it's just that we haven't finished it yet.

Next, we can create a simple stylesheet with default values for the anonymous content, as shown in Listing 15.17.

Listing 15.17 CSS2 stylesheet for the noteplacer XBL binding.

```
stack {
  background-color : background;
  font-family : -moz-fixed;
}
box.page {
  background-color : white;
  border : solid thin;
  border-color : black;
}
box.note {
  background-color : lightyellow;
  border : solid thin;
  border-color : yellow;
}
```

The background color ensures that the desktop part of the widget will have the same color as the user's desktop. The -moz-fixed font (the system font) has the important property that it can be rendered at all sizes. 7px is 14px scaled down by a factor of two. Note that the default namespace for an XBL-included stylesheet is the XUL namespace. This means that tags in the stylesheet do *not* need to be prefixed with a CSS2 namespace. Such namespaces look like this:

```
@namespace url("http://www.mozilla.org/keymaster/gatekeeper/
     there.is.only.xul");
xul|stack { ... styles ... };
```

The stylesheet will go in the <resources> section of the XBL specification, using a relative URL. It has a relative URL because the noteplacer binding is specific to the notetaker package and in the same directory as the binding. It is not a global binding in the global package. In general, one CSS file should be used for all bindings for the notetaker package, and that file should be called notetaker.css. We're using a different name just to emphasize that the file contains only XBL style information and to avoid confusion with the skins created earlier. The finished piece of XBL reads

```
<resources>
  <stylesheet src="noteplacer.css"/>
</resources>
```

Having created the content and stylesheets, we've finished half the binding and have completely specified the visual and XML interface for the widget.

15.7.3 Adding XBL Functionality

The remainder of the noteplacer binding is the <implementation> and <handlers> sections. These two sections provide the JavaScript and user-input interfaces to the new widget. Let's look at the JavaScript side first, based on the bulleted list of wanted features in the earlier part of this "Hands On" session.

It's very common to see XML attributes reflected directly as JavaScript properties on the bound object, and we could do that. Adding such reflected features is trivial, as this example from the button-base binding shows.

```
<property name="type"
  onget="return this.getAttribute('type');"
  onset="this.setAttribute('type', val); return val;"/>
```

We don't use this approach because we don't want to store the state of our widget in XML attributes. That is a design decision that weighs the advantages and disadvantages of XML attributes. The advantages of storing state in attributes follow:

- Very common default cases require no attributes at all, which speeds up widget processing.
- JavaScript properties and XML attributes are automatically coordinated because one always depends on the other.
- The state is accessible from CSS stylesheets.
- The state is more easily accessible from C/C++ platform code.
- Bound tags can inherit values into those attributes if they're used in the <content> of another binding.

The disadvantages of storing state in XML attributes follow:

- Access to attributes is slow compared to ordinary JavaScript variables.
- Any attempt at information hiding or simplification is pointless because state is deliberately exposed.
- Scripts inside the binding (and outside) are required to do additional checks for the special null case when a given attribute is missing altogether.

If you are building a general-purpose toolkit of widgets, or building a frequently used widget, then the advantages may outweigh the disadvantages, and using XML attributes should be considered.

In our case, the widget is clearly application-specific, rather than general-purpose. We have no popular default cases. We won't be adding platform code to process our widget or using complex style tricks. The only null case might be the `pageless` attribute, where absence could be equated to `false`. For our widget, we prefer a reliable interface that never yields up a `null` or an empty value. The rest of the application can then use the widget as a true black box. This decision means that we will use JavaScript to store the widget's state.

Overall, we need to create `<field>`, `<property>`, `<method>`, `<constructor>`, and `<destructor>` content for the widget.

For the `<field>` attribute, we'll expose the golden ratio to the user. This is not ideal because the value can't be usefully passed to the XBL object, and so we might question how useful it is. At least it illustrates the syntax:

```
<field name="ratio">(1+Math.sqrt(5))/2</field>
```

This mathematical expression is one of several ways to calculate the golden ratio.

For the `<constructor>`, we observe that our widget tends to be quite large, and that it shouldn't stretch and shrink as a result of surrounding layout changes. This fixed size is unusual, and shouldn't always be a design choice, but it suits our purpose, which is to represent the desktop accurately. It also means that the `<constructor>` can size the widget once, and we won't have to worry about its flexing later. The constructor also must store the state of the widget in JavaScript. The logic for both state and layout is shown in Listing 15.18.

Listing 15.18 `<constructor>` code for the `noteplacer` XBL binding.

```
this.getAEBA = function (x,y,z) {
  return document.getAnonymousElementByAttribute(x,y,z);
}

this.desktop = getAEBA(this,"anonid","desktop");
this.page    = getAEBA(this,"anonid","page");
this.note    = getAEBA(this,"anonid","note");
```

```
this.d = {};      // protect numbers from string-ification

this.d.top     = 40;
this.d.left    = 40;
this.d.width   = 100;
this.d.height  = 100;

this.d.scale    = att2var("scale", 2);
this.d.screenx  = att2var("screenx", window.screen.width);
this.d.screeny  = att2var("screeny", window.screen.height);
this.d.pageless = att2var("pageless", false);

this.d.page_offset = 0;
if ( !d.pageless )
  d.page_offset = (d.screenx - d.screeny/ratio)/d.scale/2;

/* layout the content once */

this.setAttribute("style","font-size:"+14/d.scale+"px;");
placeDesktop();
placeNote();
if ( ! d.pageless ) placePage();
```

The `getAEBA` property is just a shorthand reference to the mouthful that is `document.getAnonymousElementByAttribute()`. We remember the important DOM elements of the widget for our future convenience. The `d` (for data) property saves our numeric state data away from XBL's tendency to force simple values into strings. All the properties of `this.d` are hidden from the user of the widget (the application programmer who places `<noteplacer>` tags). After these properties are calculated, the `<content>` part of the widget has its attributes adjusted, including scaling the displayed text. The `att2var()`, `placeDesktop()`, `placeNote()`, and `placePage()` functions are all methods of the `this` object that are declared at the start of the constructor code. They are shown in Listing 15.19.

Listing 15.19 Private `<constructor>` methods for the `noteplacer` XBL binding.

```
this.att2var = function (name, dvalue) {
  var val = this.getAttribute(name);
  if ( val == "" ) return dvalue;
  if ( isNaN(val) ) return dvalue;
  return val - 0;
}

this.arg2var = function (arg) {
  var err = "Bad argument passed to noteplacer binding";
  if ( arg == "" ) throw err;
  if ( isNaN(arg) ) throw err;
  return arg;
}
```

```
this.placeNote = function () {
  note.setAttribute("top", d.top / d.scale);
  note.setAttribute("left", d.page_offset + d.left / d.scale);
  note.setAttribute("minwidth", d.width / d.scale);
  note.setAttribute("minheight",d.height / d.scale);
}

this.placeDesktop = function () {
  desktop.setAttribute("minwidth", d.screenx/d.scale);
  desktop.setAttribute("minheight",d.screeny/d.scale);
}

this.placePage = function () {
  page.setAttribute("minheight", d.screeny/d.scale);
  page.setAttribute("minwidth",  d.screeny/d.scale/ratio);
  page.setAttribute("top", "0");
  page.setAttribute("left", d.page_offset);
}
```

The `att2var()` method calculates the value for a state variable based on the default value and an optional XML attribute that overrides that default if it is present. The `arg2var()` method checks that a passed-in argument is a legal number. We'll use it shortly. The other methods adjust the positions of the widget content based on the state information and a little mathematics. All these methods, plus the `getAEBA()` method, are private to the widget. They don't appear as properties on the `<noteplacer>` tag's DOM object.

We haven't held any XPCOM components or other big objects during the life of the widget, so there's nothing to let go of when the widget is destroyed. As a consequence, there is no `<destructor>` required for our widget.

We've already made the effort to capture the state information of the widget, and that makes implementing the `<property>` tags very easy. Listing 15.20 shows these tags.

Listing 15.20 `<property>` tags for the `noteplacer` XBL binding.

```
<property name="top"
  onget="return this.d.top;"
  onset="this.d.top = arg2var(val); placeNote();"/>
<property name="left"
  onget="return this.d.left;"
  onset="this.d.left = arg2var(val); placeNote();"/>
<property name="width"
  onget="return this.d.width;"
  onset="this.d.width = arg2var(val); placeNote();"/>
<property name="height"
  onget="return this.d.height;"
  onset="this.d.height = arg2var(val); placeNote();"/>

<property name="scale" readonly="true"
  onget="return this.d.scale;"/>
```

```
<property name="screenx" readonly="true"
  onget="return this.d.screenx;"/>
<property name="screeny" readonly="true"
  onget="return this.d.screeny;"/>
<property name="pageless" readonly="true"
  onget="return this.d.pageless;"/>
```

These properties could hardly be simpler to implement. Note how the setters throw an exception via `arg2var()` if the value passed by the widget user contains rubbish. If our widget were more dynamic, the `placeNote()` function call in each `onget` attribute would be more extensive and might have to do extensive calculations or processing.

Let's now turn to the sole `<method>` tag required.

The widget coordinates used in both XML attributes and JavaScript properties are relative to the pane that the note is placed in. When DOM events occur, however, the DOM Event object generated has coordinates that are relative to the whole window. That is a different coordinate system. The convenience method `setFromEvent()` allows a user of the widget to set a corner of the note using that Event object. The widget handles the required coordinate transformation for the user. This method is shown in Listing 15.21.

Listing 15.21 `<method>` tag for the `noteplacer` XBL binding.

```
this.getCoords = function(x,y) {
  return {
    x: (x - this.boxObject.x - d.page_offset) * d.scale,
    y: (y - this.boxObject.y) * d.scale
  };
}

<method name="setFromEvent">
  <parameter name="evt"/>
  <parameter name="cornerFlag"/>
  <body><![CDATA[
    var coords = getCoords(evt.clientX , evt.clientY);
    if (cornerFlag) {
      d.width  += d.left - coords.x;
      d.height += d.top - coords.y;
      d.top = coords.y;
      d.left = coords.x;
    }
    else {
      d.width  = coords.x - d.left;
      d.height = coords.y - d.top;
    }
    placeNote();
    ]]>
  </body>
</method>
```

The `getCoords()` function converts the supplied coordinates. It is added to the `<constructor>` contents so that it resides with the other utility functions—it is not exposed to users of the widget. The `setFromEvent()` method is exposed to the user. After that method has local coordinates from `getCoords()`, it recalculates the size and top corner of the note based on the location provided and the existing note dimensions. That done, the note is repositioned to match. This method could use some additional sanity checks (e.g., width and height should never be negative), but it does the job for our purposes.

The last part of the binding specification is the `<handlers>` section. We have two handlers, both for the DOM 2 click event, but for different mouse buttons. For the primary button, we introduce another private variable, this time called `this.d.topclick`. If this variable is true, then a primary click will set the location of the top-left corner of the note. If it is false, the bottom-right corner will be set. We initialize this variable to `true` in the constructor. For the secondary button, we rely on the Pythagorean theorem ("the sum of the squares of the sides of a right triangle equals the square of the hypotenuse") to detect the leading diagonal corner of the note nearest the click point. That corner is then updated. The resulting code is shown in Listing 15.22.

Listing 15.22 `<handler>` tags for the `noteplacer` XBL binding.

```
<handlers>
  <handler event="click" button="0"><![CDATA[
    this.setFromEvent(event,d.topclick);
    this.d.topclick = !this.d.topclick;
    ]]>
  </handler>

  <handler event="click" button="2"><![CDATA[
    var coords = this.getCoords(event.clientX, event.clientY);
    var dist1, dist2;
    with (Math) {
      dist1 = sqrt(pow(coords.x-d.left,2) + pow(coords.y-d.top,2));
      dist2 = sqrt(  pow(coords.x - (d.left + d.width), 2)
                   + pow(coords.y - (d.top + d.height), 2)
                   );
    }
    this.setFromEvent(event, (dist1 < dist2));
    ]]>
  </handler>
</handlers>
```

Using CDATA sections avoids a great deal of time otherwise wasted in debugging. The handler functions reuse both the exposed `setFromEvent()` method and the private `getCoords()` method, and so their logic is very simple.

With the `<handlers>` content finished, the `noteplacer` binding is complete, at about 150 lines of code all told. With no more binding changes required, all that remains is to try it out.

15.7.4 Integrating the Binding

Having created the noteplacer binding, we next want to try it out. In fact, the next few steps are a useful way to create a binding test harness. Such a harness is as useful during binding development as it is for final testing.

To connect the binding to a tag, we use a single style rule:

```
noteplacer {
  -moz-binding : url("noteplacer.xml#noteplacer");
}
```

We put this in a file named xulextras.css for now. This file lives in the same directory as the rest of the noteplacer content. This stylesheet enhances the standard set of XUL tags with the new <noteplacer> tag.

In a test of the new widget, we want to see the following things:

- The widget fits properly inside surrounding content.
- The widget reacts properly to user input, including window resizing.
- Attributes set on the widget have the desired effect.
- Explicit content provided to the widget has the desired effect.

To that end, we construct the test XUL document of Listing 15.23.

Listing 15.23 Test page for the noteplacer XBL binding.

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
<?xml-stylesheet href="xulextras.css" type="text/css"?>
<!DOCTYPE window>
<window xmlns="http://www.mozilla.org/keymaster/gatekeeper/
          there.is.only.xul"
  onload="install()">
<script>
  var np;

  function install()
  {
    np = document.getElementById("test");
    np.addEventListener("click", report, false);
  }

  function report()
  {
    var str = "";
    str += "Top: "     + np.top + "\n";
    str += "Left: "    + np.left + "\n";
    str += "Width: "   + np.width + "\n";
    str += "Height: "  + np.height + "\n";
    str += "ScreenX: " + np.screenx + "\n";
    str += "ScreenY: " + np.screeny + "\n";
    str += "scale: "   + np.scale + "\n";
```

```
    alert(str);
  }
</script>
<description>Before Test </description>
  <hbox>
    <description value="Left of Test"/>
    <noteplacer screenx="1024" screeny="768" scale="4">
      <toolbox flex="1">
        <toolbar grippyhidden="true">
          <description value="File"/>
          <description value="Edit"/>
          <description value="View"/>
          <description value="Go"/>
          <spacer flex="1"/>
        </toolbar>
        <spacer style="background-color:white" flex="1"/>
      </toolbox>
    </noteplacer>
    <description value="Right of Test"/>
  </hbox>
  <description>After Test </description>
</window>
  </handler>
</handlers>
```

This document surrounds the `<noteplacer>` tag with other content and provides a fake toolbar as a content hint for the page. It also lodges an event handler on the tag. That event handler will be applied to the `click` handlers specified in the binding. When this document is displayed, it appears as in Figure 15.3.

Obviously, the supplied toolbar is a primitive example of a browser window, but then an indication to the user is all that is required. Perhaps one day the screen capture technology inside the DOM Inspector's XPCOM screenshot components will be complete. Then an actual, scaled-down screenshot can be put where the explicit content currently goes.

Fig. 15.3 Test XUL document showing customized `<noteplacer>` tag.

If the click handlers receive input, then an alert box appears that reports the current position of the note. That alert box is shown in Figure 15.4.

A little mathematics and the values displayed can be confirmed as probably accurate.

It is easy to see how the <noteplacer> tag could be integrated into the NoteTaker application using JavaScript. From the Edit dialog box, it could be launched in a separate second dialog box or embedded somewhere in the existing <tabbox>. When the dialog box closes, an onclose event handler reads the state of the widget and copies the values into the matching fields in the Edit pane, or into the matching fields in the toolbar note object.

A single widget by itself gives the user few clues how it should be used. Like many XUL tags, <noteplacer> is functional rather than obvious. This tag should be surrounded by other content when served up to the user. Figure 15.5 shows one possible scenario.

That ends the "Hands On" session for this chapter.

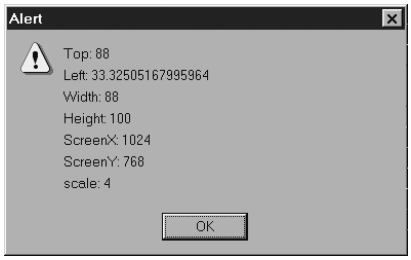


Fig. 15.4 Diagnostic alert box confirming the state of a <noteplacer> widget.

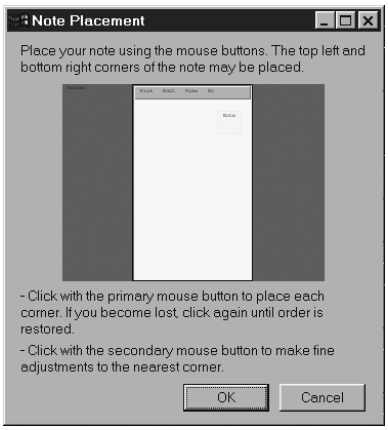


Fig. 15.5 Application use of the <noteplacer> tag.

15.8 DEBUG CORNER: XBL DIAGNOSIS

There is nothing mysterious about XBL bindings; the window and document objects are both available to all property, method, handler, and constructor/destructor code. All the debugging techniques available to normal scripts are also available to scripts used in XBL.

XBL is not as mysterious to work with as RDF either. Syntax errors in an XBL document are reported to the JavaScript console, and the messages supplied are readable.

The two most frequent problems when creating bindings are the result of XML namespaces and internal JavaScript properties. Note the following regarding namespaces:

- The XBL `inherit` property must be prefixed with a namespace identifier like `xbl:` because the default namespace for the tag that `inherit` resides in will always be either XUL or HTML.
- XBL must have a separate namespace because it shares some tag names with other XML applications such as XUL, RDF, and HTML.
- If the standard set of namespaces in which the default namespace is XBL is used, then every XUL or HTML tag in the `<content>` section must have a namespace prefix. If not, nothing will appear.

On internal JavaScript properties of XBL bindings, two simple omissions are:

- Forgetting to prefix binding variables with `this`.
- Forgetting that immediate properties of this store simple values as String types.

The most frustrating problem with bindings is the way they load asynchronously. You can't assume when bound methods and properties will be available. There are several trivial solutions to this problem.

The simplest solution is to use a document `onload` handler for any initialization code that is required. This will work only if all the bindings used are bound once directly in stylesheets and never changed.

The next simplest solution is to add this field to every binding created:

```
<field name="loaded" readonly="true">true</field>
```

Before scripting the bound tag, check that this property is set; if it is, then the binding is available.

A more systematic solution might use the `<constructor>` part of a binding's specification. This piece of code is a simple registration system:

```
with (window.document)
  bindTotal = (bindTotal) ? ++bindTotal : 1;
```

If a XUL application window consists of only a few very complex XBL widgets, then it is easy to keep track of how many have finished loading. A timer can be used to consult the in-progress total until all widgets are loaded. If a destructor also decrements this total, then the total can be a running total that keeps track of loaded widgets when their bindings are dynamically changed. For highly specialized XBL widgets, this small amount of coupling between the widget and the containing document can be a practical solution to loading delays. In short, this is the same logic as that used to coordinate frames in a Web document.

A last solution to this asynchronicity is to use the `loadBindingDocument()` method, which prevents the problem entirely.

Finally, XBL widgets have some security restrictions. If the documents represented by two URLs are to have full access to each other, they must pass Mozilla's "Same Origin" test. That test demands that both URLs come from the same server. XBL widgets must also pass this test or be installed where security restrictions are dropped. A XUL application delivered by one server cannot ordinarily use an XBL binding defined at another server. Rather than go through a complex certification and code signing exercise, it is generally simpler just to install the bindings in the chrome.

In all cases, if a binding is to take advantage of XPCOM components, the document that contains the bound tag of interest should be installed in the chrome. Bindings have the same security as the documents to which they are bound.

15.9 SUMMARY

Mozilla's XBL technology adds extensibility and componentization to user interface development. It follows the general philosophy of visual markup languages like HTML: Keep it simple so that it is quick to use. But XBL is also a software engineering tool—it allows common pieces of functionality to be isolated, specified, and reused.

The XBL part of the Mozilla Platform helps glue interactive, visual widgets to the rest of the platform by giving them highly customized object-like features. XBL bindings back nearly every XUL tag, and many HTML tags as well. The default actions supplied by these bindings contribute to the ease of use that those other markup languages enjoy.

In the next chapter, we'll see the objects to which XBL bindings usually connect. These are the many objects supplied by the XPCOM system. Many XPCOM objects have been explored already, but there are always objects complementary to other technologies like XUL and RDF. Many XPCOM components are functional in their own right, and these components dramatically extend a Mozilla-based application.