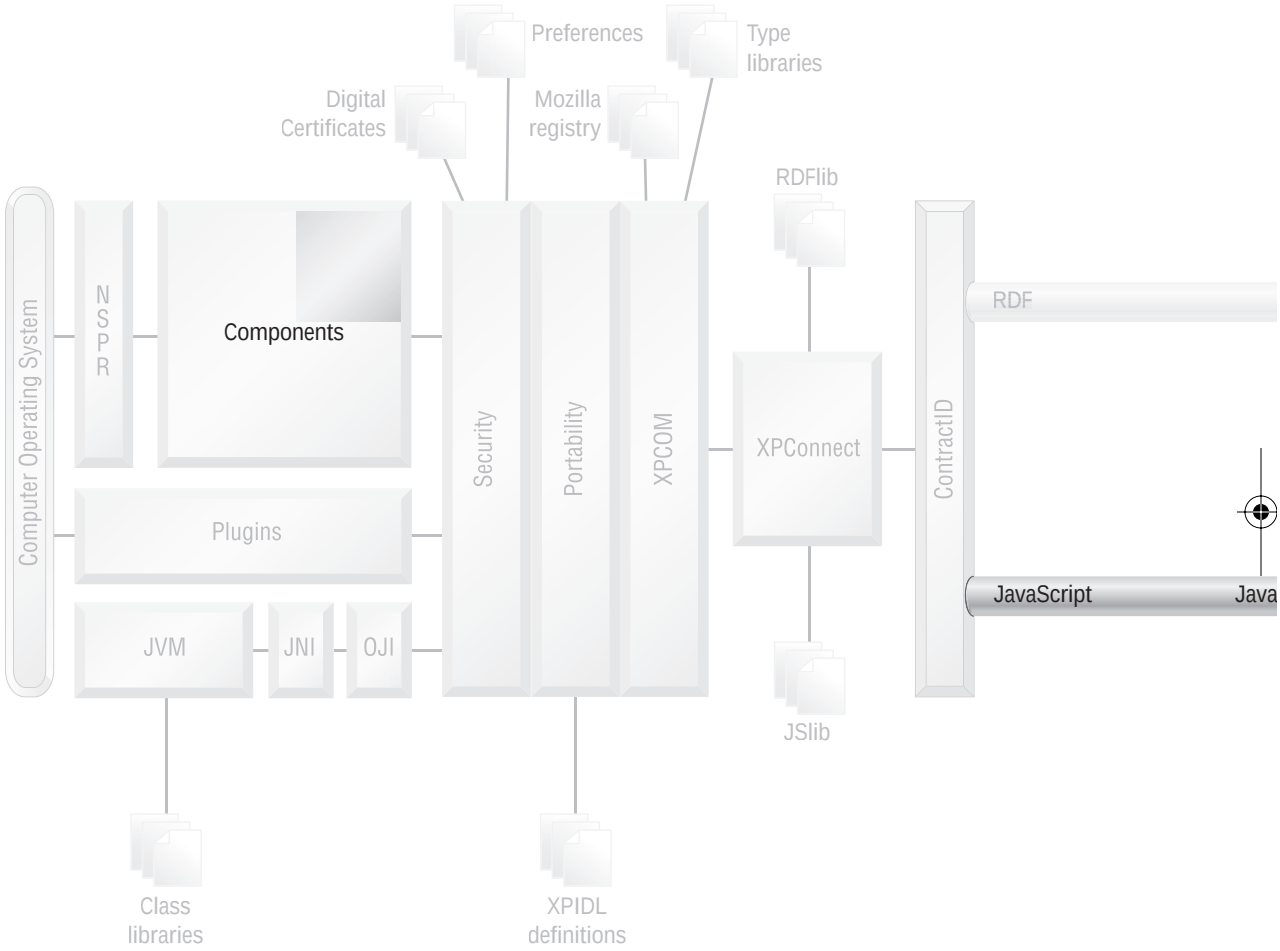
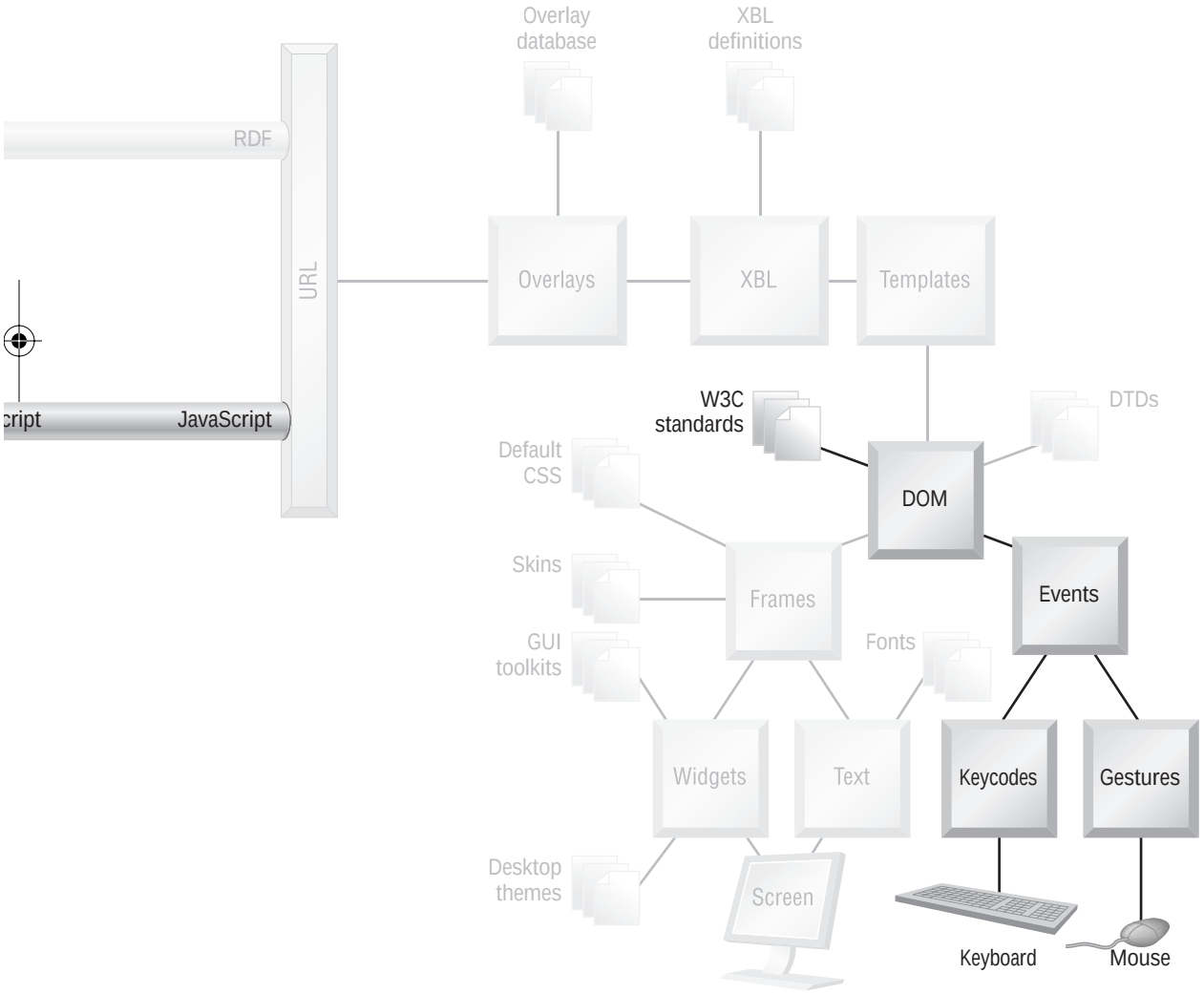

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
Plugins
JavaScript
Java
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 6

Events



The Mozilla Platform is designed first and foremost for interactive applications. This chapter explains the basic features of Mozilla that build a bridge between a real live human and a software system. Such features are quite varied. They range from device drivers to single keystrokes to forms, menus, and toolbars. Underneath most of these possibilities is an event-driven input-processing system. This chapter covers that underlying raw, event-driven input.

In order to act, a user's raw input must first be boiled down into something more useful. This chapter also describes Mozilla's low-level collection systems for user input from the mouse, keyboard, and other sources. User input is deeply linked with Mozilla's event management systems, and the basics of these systems are carefully explored as well.

Traditional computer-user interfaces like keyboards, mice, and monitors are not very standard at all. These devices are connected to an operating system and a desktop, and those systems attempt to impose their own unique standards in order to make sense of it all. Such standards make the construction of cross-platform applications a challenge. Even the simplest standard makes portability a challenge: one mouse button for the Macintosh, two for Microsoft Windows, and three for UNIX.

On the other hand, most operating systems use Control-X, Control-C, and Control-V for cut, copy, and paste operations, respectively, except when keyboards don't have X, C, or V. The thing about standards is this: Not only are there many to choose from, but there are also many to be compatible with. Every standard is a little tyrant.

The Mozilla Platform must try to handle all these constraints gracefully, as well as find a way to set its own conventions. Users want their interaction with Mozilla to be familiar, obvious, simple, and reliable. Most user actions are repetitive, and users want to be able to perform those actions by reacting through habit, not by thinking.

The NPA diagram that precedes this chapter illustrates the areas of Mozilla that involve user input. The diagram is exactly as you would expect—most of the user input activity is on the user side of the diagram. The box labeled "Events" is of central concern to this chapter; it represents a system described in the DOM 2 Events and DOM 3 Events standards.

Now that JavaScript, XPConnect, and XPCOM have been introduced, a few relevant components from the back part of Mozilla are explored as well. These components, and the XUL tags that happen to complement them, provide several ways to express where an event should go after it appears.

Before proceeding, note this word of caution. Many of the concepts in this chapter are trivial to grasp. Some, however, are a little too subtle for readers with only HTML experience. If you are a beginner programmer, it's highly recommended that you experiment with the event concepts described, to gain experience. These event concepts recur throughout Mozilla and are an important step on the road to mastering the platform. The "Hands On" session in this chapter is a good starting point.

6.1 How Mozilla Handles Events

Figure 6.1 is a conceptual overview of the event-driven systems inside Mozilla.

All these approaches are briefly discussed here; most are extensively discussed here, but commands are discussed in Chapter 9, Commands, and content processing is a topic addressed in Chapter 16, XPCOM Objects.

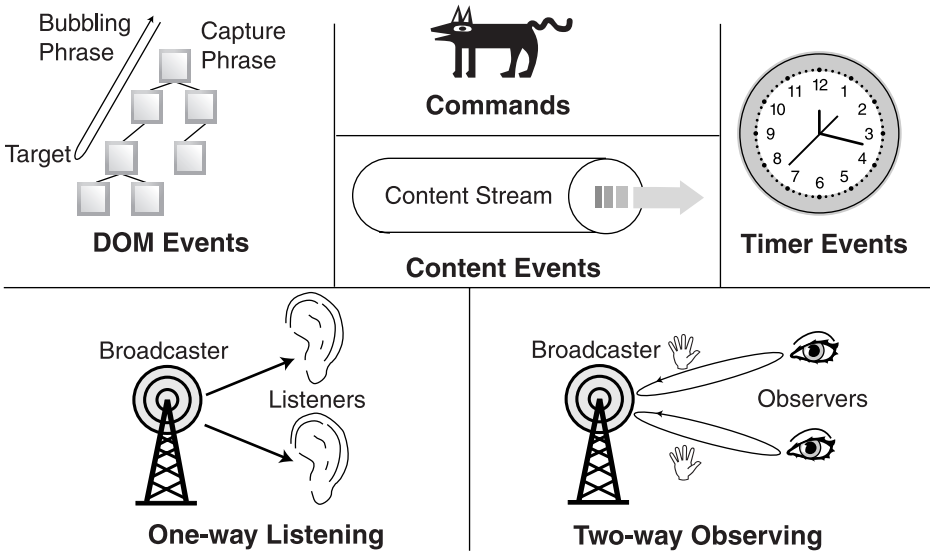


Fig. 6.1 Overview of event processing inside Mozilla.

6.1.1 Scheduling Input and Output

All programs need some form of input and output, and the Mozilla Platform is no exception. It can receive input from devices driven by human beings, from network and Internet connections, from the operating system, from other applications, and from itself. It can send output to most of those places. All together, that's a complicated set of possibilities to juggle. What if input arrives from several places at once? Won't something be missed? Mozilla's solution involves a simple event-driven scheduling system.

The problem with scheduling systems is that few programming languages have direct support for them, and simple-minded programs rarely use them. Some programmers never come into contact with such systems. A beginner's introduction is provided here. Experienced programmers note that this system is equivalent to a multithreaded environment and the `select()` kernel call.

"hello, world" might be the first program everyone attempts, but it only produces output. The second program attempted is likely to do both input and

output. Listing 6.1 shows an example of such a second program, using JavaScript syntax. You can't run this program in a browser—it is just an imaginary example.

Listing 6.1 Example of a first input-output program.

```
var info;
while (true)
{
  read info;
  print info;
}
```

According to the rules of 3GL programming languages, this program has one statement block containing two statements, which are executed one after the other, over and over. This program never ends; it just reports back whatever is entered. It could be written in a slightly more structured way, as Listing 6.2 shows.

Listing 6.2 Example of a first structured input-output program.

```
var info;
function read_data()  { read info; }
function print_data() { print info; }

function run()
{
  while (true)
  {
    read_data();
    print_data();
  }
}

run();
```

Even though this second program performs only the single statement `run()`, it's clear that this is the same step-by-step approach used in Listing 6.1. Software based on scheduled systems doesn't work this way at all. The same program expressed in a scheduling system might appear as in Listing 6.3.

Listing 6.3 Example of a first scheduled input-output program.

```
var info;

function read_data() {
  if (!info)
    read info;
}
```

```
function print_data() {
  if (info) {
    print info;
    info = null;
  }
}

schedule(read_data, 500);
schedule(print_data, 1000);

run();
```

The `schedule()` function in this example takes two arguments: a function to run and a time delay in milliseconds. `schedule()` says that the supplied function should be run regularly whenever time equal to the millisecond delay has passed. `run()` tells the system to start looking for things to run. At time 0.5 seconds (500 milliseconds), `read_data()` will be run. At time 1.0 seconds (1,000 milliseconds, which is also 2 * 500 milliseconds), both functions will be run. At time 1.5 seconds (1,500 milliseconds = 3 * 500 milliseconds), `read_data()` will run for a third time, and so on. The `run()` function never finishes.

Such an arrangement can seem quite foreign, especially since the `schedule()` and `run()` functions aren't defined anywhere in the program. You just have to trust that they work as advertised. They are services provided by some existing scheduling system. Worse, `read_data()` and `print_data()` don't execute in any particular order. In fact, `read_data()` runs twice as frequently (every half second, compared to once a second for `print_data()`).

In order to work together, the two functions rely on a shared piece of data (the `info` variable). `read_data()` will not read anything until the last item in info is used by `print_data()`. `print_data()` will not print anything until `read_info()` puts something into `info`. The two functions are coordinated with each other via shared state information, even though they are otherwise independent.

In Mozilla, you can either create such a scheduler yourself or use an existing one and add your own scheduled items, but neither strategy is common practice. The Mozilla Platform takes care of all this for you. It has a built-in scheduler and scheduled functions that check for all possible forms of user input and output. A keypress is one kind of input, a JavaScript script to run is another, and a chunk of HTML content received from a Web server is a third. Scheduled items can be small (a mouse click) or very large (please redisplay this whole HTML document). Everything in Mozilla is a scheduled item, even if this fact is buried deeply underneath everyday features.

Occasionally this scheduled system is revealed to the programmer. Listing 6.4 closely matches Listing 6.3; however, it is a legal and runnable JavaScript script. Include it in any HTML or XML page, fill in the regularly

appearing popup, and watch the text in the window's title to see the scheduling system at work. The example goes slowly so that you have time to shut the windows down with the mouse when you've had enough.

Listing 6.4 Example of function scheduling using `setInterval()`.

```
function read_data() {
  if (!window.userdata)
    window.userdata = prompt("Enter new title");
}

function print_data() {
  if (window.userdata) {
    window.title = window.userdata;
    window.userdata = null;
  }
}

window.setInterval(read_data, 5000);
window.setInterval(print_data, 100);
```

Even though the two functions run at very different frequencies, the system works. `print_data()` will do frequent checks for something to do; `read_data()` will less frequently ask the user for information.

This example could be simplified. The `read_data()` function could be made to write the title into the window as soon as it is read. If that were done, `print_data()` could be done away with entirely. Such a variation shows how bigger functions mean fewer scheduled items. The reverse is true as well. The number of items scheduled is just a programmer's choice at design time.

Listing 6.4 can be viewed from a different perspective. Rather than be amazed that two functions can run in a coordinated but independent manner, it is possible to think about the roles of those two functions inside the running environment. `read_data()` acquires new information and makes it available. From the perspective of the run-time environment, it is a producer of information. `print_data()` takes available information and puts it somewhere else, using it up in the process. It is a consumer of information.

Producers and consumers are very important design ideas, and they appear throughout Mozilla. The idea of a *producer-consumer pair* working together is a very common one, and the pair is tied together by a common purpose: the desire to process one kind of information. Sometimes the pair sits and does nothing. When some new stimulus occurs, the pair goes into action.

A further modification to Listing 6.4 might be to change the `setInterval()` calls to `setTimeout()` calls. `setTimeout()` schedules its function argument to run just once, not repeatedly. After that sole run, the scheduled item is removed from the schedule system, never to appear again. Items that appear just once in a schedule system are a very important special case.

Suppose that a schedule system consists of nothing but one-off items. Further, suppose that these items all have a time delay of zero. Lastly, suppose

that these items are all producers—they will each produce one item of data when they run. In such circumstances, the schedule system is a simple event queue. When a producer in the queue runs, an event is said to be underway. In casual conversation, the producer in the queue is also called an event, but the data that the producer creates is called an event too. Since all the producers in an event queue have a zero delay, Mozilla will do its best to run them right away, which means they leave the queue almost immediately. Such a system is useful for reporting keystrokes, for example.

In Listing 6.4, the application programmer is responsible for creating both consumers and producers. An event queue can be arranged differently. The Mozilla Platform can be made responsible for adding the events to the queue. It can also be made responsible for finding a suitable consumer when a producer event occurs. The application programmer can be responsible for writing consumers that consume the created event data and for telling the Mozilla Platform that these consumers are interested in certain kinds of event data.

The application programmer calls these consumers event handlers, and Mozilla calls them listeners. A listener is a programming design pattern—a well-established idea that is a useful thinking point for design. Code designed to be a listener awaits information from other code and acts on it when it occurs. All JavaScript scripts are listeners that run in response to some event, even though no event appears to be at work.

This brief introduction has gone from a plain 3GL example to an event-driven system. This is how much of Mozilla works. In particular, this is how user input, for example, is processed. We have touched on the concepts of consumers, producers, scheduling, listeners, and timing along the way, and these concepts are all useful in their own right. When the moment comes to get dirty using XPCOM components, these concepts will become essential. For now, all you need to do is accept that events drive all processing in Mozilla, even if the thing to process is just one big script.

From Chapter 5, Scripting, you know that much of the Mozilla Platform is available as XPCOM interfaces and components, and that includes event queues. The two components

```
@mozilla.org/event-queue;1
@mozilla.org/event-queue-service;1
```

and the interface

```
nsIEventQueue
```

allow an event queue to be created, started, and stopped. These components are only useful for applications that deeply customize Mozilla.

Because the Mozilla event queue is so low-level, it is rarely used directly. Instead, Mozilla builds several higher-level systems on top that are easier to work with. These are the features that application programmers depend on heavily. They are still “below the surface” in the sense that they don’t include much direct user input. They are like the middle layer of a cake, not the icing.

This overview of how Mozilla processes events is continued with a look at no less than five of these middle systems.

6.1.2 The DOM Event Model

The event handlers and events that Web developers use on a daily basis come from the W3C's DOM Events standards. Many of these events apply to any XML document, including XUL, and experience with these handlers can be directly applied to Mozilla. The DOM Event system sits on top of Mozilla's basic event queue. DOM Events are restricted to a single XML document.

Because the DOM Events standard has only recently been finalized with DOM level 3, Web browser support has some catching up to do. Cross-browser compatibility issues, nonstandard event behavior, and gaps have made the use of events in scripts something of a vexing issue. Generally, the "flat" event model of version 3.0 Web browsers, in which events do not propogate much, is all Web developers risk using for cross-platform HTML.

With the release of Mozilla, many of these problems have gone away. For application developers and for XUL documents, there are no cross-platform issues. Mozilla supports the full DOM 2 and DOM 3 Event flow for many common events. Both event capture and bubbling phases are supported.

Mozilla also supports the use of more than one event handler per event target, as per the standards. It is strongly recommended that you read sections 1.1 and 1.2 of the DOM 3 Events standard, if you haven't yet. That's a mere four pages of reading.

To briefly summarize those sections of the standard, events in an XML document can be handled with JavaScript scripts. A piece of script or a single function is an event handler, which the programmer installs against an event target. An event target is just a tag or the tag's DOM 1 `Element` object, plus an event type. When an event occurs as a result of user input or some other reason, an `Event` object is created. That object travels down the DOM hierarchy of tags, starting with the `Document` object and ending with the event target tag. This is the capture phase, and any tag along the way can process the event if it has a suitable event handler. Such an interposed handler can stop the event or allow it to continue to the event target tag.

After the `Event` object reaches the event target tag, the intended event handler is executed. This handler can stop the `Event` object from traveling and prevent the default action from occurring. The default action is what would happen if the event target tag had no installed handler at all. After the handler has finished running, the default action takes place (if it hasn't been stopped), and then the event enters the bubble phase. In the bubble phase, the `Event` object returns up the DOM hierarchy to the `Document` object, and again can be intercepted by one or more other handlers on the way. When it reaches the `Document` object, the event is over.

Events travel down and up like this to support construction of two types of interactive documents.

In one design, tags within a document are considered dumb objects that need to be managed. In this case, events need to be captured at the most abstract level (at the document level) so that high-level processing of the document's contents can occur. Examples of this type of design include window managers that receive input on behalf of the windows, desktops that receive input on behalf of icons, and rubber-banding (area-selection) operations. The capture phase supports this kind of design.

In another design, tags within a document are considered smart objects that can think for themselves. Input goes straight to the most specific object, which does its own processing. Examples of this type of design are ordinary application windows, forms, and graphical tools. This is the more common use of XUL. The bubbling phase supports this kind of design.

Table 6.1 shows events that Mozilla recognizes. This list is derived from the Mozilla source file `content/events/src/nsDOMEvent.cpp` and is organized according to the sections of the DOM 3 Events standard, plus two columns for Mozilla custom events. There are many XUL tags that support only a subset of these events.

Mozilla-specific events are discussed in the appropriate topic in this chapter. Here is a brief summary of these events. `dblclick` occurs on double-click of a mouse button. `contextmenu` occurs if the mouse or keyboard makes a context menu appear (e.g., right-click in Microsoft Windows). Events

Table 6.1 Implemented events in Mozilla

DOM mouse	DOM text	DOM HTML	DOM mutation	Mozilla	Mozilla
click	keydown	load	DOMNodeInserted	dblclick	popupshowing
mousedown	keyup	unload	DOMNodeRemoved	contextmenu	popupshown
mouseup		abort	DOMAttrModified		popuphiding
mouseover		error	DOMCharacterData-Modified	keypress	popuphidden
mousemove		select		text	
mouseout		change		command	dragenter
		submit		commandupdate	dragover
		reset		input	dragexit
		focus			dragdrop
		blur		paint	draggesture
		resize		overflow	
		scroll		underflow	broadcast
				overflowchanged	close

grouped with `keypress` are for keyboard actions or for actions that are typically initiated by the user. Events grouped with `paint` are for changes to document layout. `popup...` events are for menu popups and tooltips. `drag...` events are for drag-and-drop gestures with the mouse. `broadcast` supports the broadcaster-observer system that is a Mozilla-specific technology. `close` occurs when a user attempts to close a window.

These events all generate an `Event` object, as per the DOM Events standard. Mozilla's `nsIDOMEvent` interface is equivalent to the `Event` interface in the standard, and other interface names convert similarly.

Table 6.2 shows events that Mozilla does not recognize as of version 1.4. These events are also drawn from the DOM 3 Events standard.

Table 6.2 Unimplemented standard event names in Mozilla 1.2.1

DOM user interface events	DOM mutation name events	DOM text events
DOMFocusIn	DOMElementNameChanged	textInput (use keypress or text)
DOMFocusOut	DOMAttributeNameChanged	
DOMActivate	DOMSubtreeModified	
	DOMNodeInsertedIntoDocument	
	DOMNodeRemovedFromDocument	

The common way to specify an event handler is to use an XML attribute. Mozilla is case-sensitive, and all such attributes should be lowercase, unless uppercase is explicitly stated in these tables. A handler attribute has the same name as the event with a prefix of "on". Less popular, but more powerful, is the DOM system for setting event handlers. Listing 6.5 compares the syntax for these two systems.

Listing 6.5 Event handlers registration techniques.

```
<!-- the XML way -->
<button id="test" onclick="myhandler(event);">
  <label value="Press Me"/>
</button>

// The DOM way
  var obj = getElementById("test");
  obj.addEventListener("click", myhandler, false);
```

Traditional HTML would supply the handler function a `this` argument. In Mozilla, the current `Event` object is always available as an event property. That `Event` object's `currentTarget` property is equivalent to this, so use either name.

There are minor differences in the execution of these two variants. The scripted DOM approach is slightly more flexible for several reasons:

- More than one handler can be added for the same object using `addEventListener()`.
- `addEventListener()`'s third argument set to `true` allows a handler to run in the capture phase. That is not possible from XML.
- Handlers can be removed using `removeEventListener()`. In XUL, an inline event listener can only be replaced, not removed.
- The DOM method allows all event handlers to be factored out of the XML content. Handlers can therefore be installed from a separate `.js` file.

The XML approach has one advantage: It supports a script fragment smaller than a whole function. This capability might be useful for HTML, but for XUL, where structured programming is good practice, it is not much of a benefit. If your XML is likely to have a long life, it's recommended that you keep event handlers out of XML attributes.

6.1.2.1 XUL Support for Event Handlers According to Table 5.4, Mozilla does not support the `HTMLEvents` and `TextEvents` interfaces that are part of the DOM 3 Events standard. That is not the full story.

XUL is not HTML, so lack of support for `HTMLEvents` is not a surprise. The default actions for some HTML tags are substantial (e.g., form submission or link navigation), but default actions for XUL tags are less common. XUL tags, on the other hand, are frequently backed by XBL bindings and sophisticated styles. These bindings and styles add effects and handlers that might as well be counted as default actions. If a XUL tag has no programmer-installed event handlers, but something is still happening when the event occurs, the best place to start looking is in `xul.css` in the `toolkit.jar` chrome file. That is where default styles and bindings for tags are located. Chapter 15, XBL Bindings, describes how to add event support using XBL bindings.

TextEvents is a DOM standard term for events in which a character is sent to an XML document. In plain English, this means `keypresses`, but `TextEvents` could in theory originate from some other source. The `TextEvents` section of the Events standard is based on Mozilla's `nsIKeyEvents` interface, but it is slightly different in detail. In practice, Mozilla has almost the same functionality as the standard.

`TextEvents` do have one shortcoming in Mozilla. These events can only be received by the `Document` or `Window` object. There is no capturing or bubbling phase at all.

There is no exhaustive cross-reference yet that documents event support for every XUL tag. The easiest way to proceed is still to try out an event handler and note the results for yourself. The individual topics of this chapter provide some guidance on specific handlers.

Many XUL tags have XBL bindings attached to them, and XBL has a `<handler>` tag. This means that the default action for a given tag might be specified in XBL, rather than inside Mozilla's C/C++. This opens up some XUL default actions to study.

6.1.3 Timed Events

A second event system inside Mozilla is a simple mechanism for timed events. Listing 6.4 and its discussion use this system. Because Mozilla's event queue is such a fundamental feature of the platform, it is both easy and sensible to make a user-friendly version available to programmers. The four calls that make up this little system are shown in Listing 6.6.

Listing 6.6 Timed events API examples.

```
timer = window.setTimeout(code, milliseconds);
timer = window.setInterval(code, milliseconds);

window.clearTimeout(timer);
window.clearInterval(timer);

// examples of timed code
timer = setTimeout(myfunction, 100, arg1, arg2);
timer = setTimeout("myfunction();", 100);
timer = setTimeout("window.title='Go'; go();", 100);
```

The argument named `code` can be either a function object or a string containing JavaScript. If it is a string, it is evaluated with `eval()` when run. The timer return value is a numeric identifier for the scheduled item. Its only purpose is to allow removal of the item. The identifier has no meaning within JavaScript, other than being unique.

Timers have some restrictions: The timed event does not have the JavaScript scope chain of the code that called it; the window object is the first scope chain member; and the number of milliseconds must be at least 10.

Perhaps the biggest restriction is the single-threaded nature of both JavaScript and XPCOM. No timed event will occur until the current script is finished running. Only one timed event can be in progress at a given time. If a given piece of code takes a long time to complete, then all the due events will bank up and be executed late. Finally, some input and output operations must wait for an event handler to finish before they can take place. For example, style layout changes made by a `setTimeout()` event won't occur until the event is over. This is a major constraint on animated systems. In general, the JavaScript interpreter must release the CPU before other activities in the Mozilla Platform can proceed.

Mozilla's XPCOM system can also provide timed events. It has an API entirely separate from `setTimeout()`. The XPCOM pair required is

```
@mozilla.org/timer;1 nsITimer.
```

Such an object works with a second object that supports the `nsIObserves` interface. To use `nsITimer`, you must implement this second object yourself. Listing 6.7 illustrates.

Listing 6.7 Single `setTimeout()` call implemented using XPCOM components.

```
var observer = {
  // Components.interfaces.nsIObserver
  observe: function(aSubject, aTopic, aData)
  {
    alert("From: " + aSubject + " saw: " + aTopic + " data: " + aData);
  },

  // Components.interfaces.nsISupports
  QueryInterface : function(iid)
  {
    if ( iid.equals(Components.interfaces.nsIObserver) ||
 iid.equals(Components.interfaces.nsISupportsWeakReference)
 || iid.equals(Components.interfaces.nsISupports)
       )
       return this;
    throw Components.results.NS_NOINTERFACE;
  }
};

with(netscape.security.PrivilegeManager)
{  enablePrivilege("UniversalXPConnect");  }

var c, timer;

c = Components.classes["@mozilla.org/timer;1"];
timer = c.createInstance(Components.interfaces.nsITimer);

timer.init(observer, 5000, timer.TYPE_ONE_SHOT);
```

Most of this code simply creates an object that implements the `nsIObserver` interface. How exactly this works isn't too important yet; it is just worth noting that the `observe()` method has the same name and arguments as the one defined in the `nsIObserver` interface, which you can find in the `nsIObserver.idl` interface file. Compare this object with the contents of `nsIObserver.idl` and `nsISupports.idl` in the source code, if you like. That created object will receive the timed event when it occurs.

The last few lines of Listing 6.7 create the `Timer` object, and set up the event. `init()` tells the timer what object should be used when the event comes due, when the event should be, and whether the event should run once or repeatedly. Just like `setTimeout()`, this script then ends. The event fires later on, causing `observer.observe()` to run. It's clear that this code achieves the same effect as a `setTimeout()` call that runs `alert()`.

The line of security code in the middle of this example is required if the script is not stored inside the chrome. The user must confirm that the script

has permission to use XPCOM, for security purposes. It is shown here just to illustrate what is required for nonchrome XUL. Most XUL is stored in the chrome and doesn't require this line.

If you include this code in any HTML or XUL page, you'll see that a single alert box appears 5 seconds or so after the page loads. For everyday purposes, it's much easier just to use `setTimeout()` and `setInterval()`. Later on, this more formal system will seem much more convenient for some tasks.

6.1.4 Broadcasters and Observers

The third event system in Mozilla is based on the `Observer` design pattern. Recall that design patterns are high-level design ideas that, when implemented carefully, create powerful, flexible, and sometimes simple software systems. Mozilla's XPCOM components, which together look like an object library, make extensive use of the `Observer` design pattern. Any nontrivial use of XPCOM means learning this pattern. This pattern is used so much that it also appears directly in XUL tags.

Earlier in this chapter the concepts of event queues, producers and consumers, and listeners were outlined. All of these things are designed to bring some structure to input or, more generally, to notify that something, somewhere has changed. Recall that listeners are a kind of consumer that is notified when an event occurs. Observers are another kind of consumer that is notified when an event occurs.

The difference between listeners and observers is that listeners don't generally respond. A listener notes the information or event that it is advised of, perhaps does something with it, but usually ignores the system that supplied the event. An observer also notes the information or event that it is advised of and perhaps does something with it, but an observer is also free to interact with the provider of the event. Observers are somewhat interactive and, in less common cases, can even take on a supervisory role.

Observers work with a broadcaster. The broadcaster's role is to notify the observer when an event occurs. The broadcaster has the ability to notify any number of observers when one event occurs, hence its name. There is a one-to-many relationship between a broadcaster and its observers. Just as one DOM 2 Event can be handled by several event handlers on the same event target, one broadcast event can be handled by several observers.

An observer and a broadcaster are like a consumer-producer pair, except the broadcaster is a producer for many observer consumers. The pair's relationship doesn't end there, though. Initially, the observer does nothing but wait until the broadcaster acts, sending an event. After the event is sent, the roles are reversed. The broadcaster does nothing, and the observer can act on the broadcaster, if it so wishes. Both are free to act on other software that they are part of.

All that is a bit abstract, so here is some concrete technology.

6.1.4.1 XUL Broadcasters and Observers

XUL supports the <broadcasterset>, <broadcaster>, and <observes> tags. It also supports the observes and onbroadcast attributes, which apply to all XUL tags. A related tag is the <command> tag. <command> is explored in Chapter 9, Commands.

These tags allow XML attribute value changes in one tag to appear in another tag. In other words, two tags are linked by an attribute. These changes are used as a notification system inside XUL. Syntax for these tags is shown in Listing 6.8.

Listing 6.8 Broadcasters and observers implemented in XUL.

```
<broadcasterset>
  <broadcaster id="producer1" att1="A" att2="B" ... />
  <broadcaster id="producer2" att1="C" ... />
</broadcasterset>

<observes element="producer1"
          attribute="att1"
          onbroadcast="alert('test1')"/>

<observes element="producer1"
          attribute="att1 att2"
          onbroadcast="alert('test2')"/>

<box observes="producer2"/>
```

att1 and att2 stand for any legal XML attribute names, except for id, ref, and persist. None of these tags have any visual appearance except possibly <box>. The last line could have used any XUL tag; there is no special reason for choosing <box>. An overview of these tags follows.

Recall that the <stringbundleset> tag was introduced in Chapter 3, Static Content. It is a nondisplayed tag with no special behavior. It is just used as a convenient container for <stringbundle> tags. <broadcasterset> has exactly the same role for <broadcaster> tags. It does nothing except look tidy in the source code; however, you are encouraged to follow its use convention.

The <broadcaster> tag has a unique id and a set of attributes. These attributes are just plain data and have no special meaning. The tag can have as many of these attributes as necessary. If any of these attributes change, the broadcaster will look for places to send an event.

The <observes> tag uses the element attribute to register itself with a <broadcaster> that has a matching id. The <observes> tag must also have an attribute attribute. This determines which of the <broadcaster>'s attributes it is observing. The second example in Listing 6.8 shows an <observes> tag observing two attributes. The value for this attribute can also be "*", meaning observe all attributes.

The onbroadcast attribute applies only to the <observes> tag. It is optional. Broadcast events will occur even if such a handler is missing. If a handler is present, it will be fired before the attribute change takes place. In that way, the old value can be read from the tag, and the new value, from the event. The onbroadcaster handler is buggy in some Mozilla versions. It can fire twice, or not at all, which are both wrong. It works reliably if the attribute attribute has a value that is a single attribute name.

Finally for this example, the observes attribute allows any XUL tag to observe all the changes from a given broadcaster. This is the role that the <box> tag has. When observes is used, there is no way to restrict the observations to specific attributes. The tag with observes receives all attribute changes that occur on the broadcaster.

Listing 6.9 shows all this at work as a series of code fragments.

Listing 6.9 Processing order for a broadcast event.

```
<button label="Press Me" oncommand="produce('bar');"/>

function produce(str)
{
  getElementById("b").setAttribute("foo",str);
}

function consume1(obj)
{
  if (obj.getAttribute("foo") == "bar")
    getElementById("x1").setAttribute("style","color:red");
}

<broadcaster id="b" foo="default"/>

<observes id="o1"
          element="b"
          attribute="foo"
          onbroadcast="consume1(this)"/>

<box id="x1"><label value="content"></box>
```

In this example, the broadcast event starts and ends with the XUL tags <button> and <box>, respectively. This practice is common but not absolutely necessary. The two JavaScript functions produce() and consume1() are the real start and end points for the event.

produce() starts the event by changing a foo attribute on the broadcaster tag. The broadcaster then informs any observers of the change. In this case, there is only one observer. The observer's foo attribute then changes. The event could finish at this point, in which case some other script would have to come back and examine the observer's attribute. In this example, the <observes> tag finishes the event with a call to the consume1() function. It has some end-effect, in this case setting a style on the <box> tag. If there were

more observers, consume2() and consume3() functions might exist as well. The consume1() function manipulates the observer tag, not the broadcaster tag. The foo attribute it manipulates is copied to the observer tag when it is changed on the broadcast tag—that is the origin of the broadcast event.

In this example, the button's handler could set the required style directly, saving a lot of tags and code. Even if there were more than one observer, each with a different effect, the button handler could still implement all these effects directly. Listing 6.9 is more sophisticated than a single button handler because the existing button handler has no idea what tags are observing the change it creates. This means that the <button>'s action is not rigidly connected to specific tags. The <button> tag provides a service, and any interested parties can benefit from it. This is particularly useful when overlays are used—they are described in Chapter 12, Overlays and Chrome. Overlays create an environment where even the application programmer is not sure what content might currently be displayed. In such cases, the observer pattern is a perfect solution: Just broadcast your event to the crowd, and those wanting it will pick it up.

There is a further use of <observes>. Any XUL tag may have an <observes> tag as one of its content tags. The parent tag of <observes> will then receive on broadcast whatever attributes of the broadcaster the <observes> tag specifies. The <observes> tag itself does not change in this case. If the broadcast event is just designed to change tag attributes, then this nested tag arrangement makes the general case in Listing 6.8 much simpler. The consume1() function can be done away with, and if attribute names are coordinated throughout the code, then Listing 6.10 shows how the <box> tag's attribute can be set without any scripting:

Listing 6.10 Processing order for a broadcast event.

```
<button label="Press Me" oncommand="produce('color:red');"/>

function produce(str)
{
  getElementById("b").setAttribute("style",str);
}

<broadcaster id="b" style=""/>

<box id="x1">
  <observes id="o1" element="b" attribute="style"/>
  <label value="content">
</box>
```

This example could be further shortened by using the observes attribute. To do this, remove the <observes> tag and add observes="b" to the <box> tag. The <box> tag now receives all broadcaster changes, not just those on attribute style, but less code is required.

Finally, a <broadcaster> tag also sends an event when its XUL page is first loaded. In this case, no broadcast-related event handlers fire.

Having covered the mechanics of <broadcaster> and <observes>, it's easy to wonder what it is all used for. The answer is that the application programmer adds a layer of meaning (semantics) on top of the basic system to achieve an end purpose. In Mozilla, informal but common naming conventions for the id attribute of the broadcaster are used to indicate different semantic purposes. Three example purposes follow:

1. **id="cmd_mycommand"**. The observer pattern can be used to send a command (like cmd_mycommand) to multiple destinations. The command might be a function attached to the broadcaster. When an event occurs, the observers all retrieve it and execute it. An example of this attribute at work is the closing of multiple windows in one operation.
2. **id="Panel:CurrentFlags"**. The observer pattern can be used to manage a resource or an object. This resource may or may not be exactly one XUL tag. In the simplest case, the <broadcaster> tag is effectively a record for the resource, holding many informational attributes. When those attributes of the resource change, the broadcaster sends updates to the observers, keeping them advised.
3. **id="data.db.currentRow"**. The observer pattern can be used as a data replication system. The id might exactly match a real JavaScript object or any other data item. When the data changes, the broadcaster tells all the observers. These observers can then grab the JavaScript object and update whatever subsystems asked them to observe the data.

These examples show that the observer design pattern is widely useful as a design tool.

When the observer concept was introduced, it was said that observers often interact with their broadcasters. This is less common for the XUL-based observer system, although there is nothing stopping an onbroadcast handler from digging into the matching broadcaster. It is far more common in Mozilla's XPCOM broadcaster system, discussed shortly.

6.1.4.2 JavaScript Broadcasters and Observers

Event handlers can be installed using an XML attribute or using a JavaScript method based on DOM standards. Both techniques were illustrated in Listing 6.8. The same is true of Mozilla's broadcasters and observers. The analogous methods apply only to XUL, however. Listing 6.11 compares the XUL and JavaScript DOM techniques for broadcast-observer pairs.

Listing 6.11 Linking broadcasters and observers using XUL JavaScript.

```
<!-- the XML way -->
<broadcaster id="bc"/>
<box id="x1" observes="bc" onbroadcast="execute(this)"/>
```

```
// the AOM way
<broadcaster id="bc"/>
<box id="x1"/>

var bc = getElementById("bc");
var x1 = getElementById("x1");
addBroadcastListenerFor(bc, x1, "foo");
addEventListener("broadcast", execute, false);

// also removeBroadcastListenerFor(bc, x1, "foo");
```

Both of these examples set up a broadcaster-observer relationship between the two tags. The new method `addBroadcasterListenerFor()` of the XUL window object sets up the connection in the JavaScript case. Note that the observer's `onbroadcast` listener needs to be installed as well.

6.1.4.3 XPCOM Broadcasters and Observers Like timers, broadcasters and observers can be created the quick, Web development way using XUL, or more slowly, but more flexibly using XPCOM components. Listing 6.7 illustrates a partial example of XPCOM component use. Recall that the observer object clearly follows the observer pattern, and that the `nsITimer` object is a broadcaster of sorts, although it is limited to accepting one observer for a given delayed event.

Listing 6.7 can be studied further. Figure 6.2 shows output from the single `alert()` call run in that example.

The three arguments passed to the `observe()` method are displayed in this dialog box. We didn't consider these arguments earlier, but they are now ripe for examination.

The third argument is data associated with the event. `nsITimer` supplies no data for this argument. Boring.

The second argument is equivalent to the `attribute` attribute in XUL's `<observes>` tag. It specifies what event is occurring. The value of "`timer-callback`" is the special value associated with an XPCOM `nsITimer` event. This value is specified in the C/C++ code of Mozilla, but it is not a standard DOM Event.

The first argument is the `nsITimer` object itself. In the alert box, it has been changed to a JavaScript String type. The only way to do this is to call the `toString()` method on that object. So the observer in this example is executing a method on the broadcaster for the trivial purpose of getting a string

Fig. 6.2 `nsIObserver observe()` arguments from `nsITimer`.

back. This is an example of objects and broadcasters actively working together. The observer can in fact examine or use any of the properties that the broadcaster has, not just `toString()`.

Mozilla has more general support for broadcaster-observer systems than `nsITimer`. The simplest observer is

```
@mozilla.org/xpcom/observer;1 interface nsIObserver
```

This is a "do nothing" observer, a bit like `/dev/null`. It's `observe()` method just returns success. Mozilla has many specialist observer components, or you can make your own, as is done in Listing 6.7. Mozilla's XPCOM broadcaster is more useful. It is based on this XPCOM pair:

```
@mozilla.org/observer-service;1 interface nsIObserverService
```

This singleton object is a general-purpose broadcaster. It will broadcast to any observer object registered with it, regardless of the observer's window of origin. Listing 6.12 shows a typical use.

Listing 6.12 Example of an XPCOM broadcaster-observer pair.

```
var observer  = { ... as in Listing 6-6 ... };
var observer2 = { ... another observer ... );

var CC = Components.classes, CI = Components.interfaces;
var cls, caster;

cls = CC["@mozilla.org/observer-service;1"];
caster = cls.getService(CI.nsIObserverService);

caster.addObserver(observer,  "my-event", true);
caster.addObserver(observer2, "my-event", true);

caster.notifyObservers(caster, "my-event", "anydata");
```

The `caster` object is the single broadcaster. When observers are added, using `addObserver()`, the first argument is the observer object; the second is the event string to observer; and the third argument is a flag that indicates whether the observer is written in JavaScript. If it is, `true` must be used. If it is a C/C++ XPCOM component, `false`. `notifyObservers()` generates an event of the given type on the broadcaster; the third argument is a piece of arbitrary data in which arguments or parameters for the event can be placed. It will be received by the observer object.

That concludes this introduction to Mozilla's broadcasters and observers.

6.1.5 Commands

The fourth event system noted in this chapter relates to commands. In Mozilla, a command is an action that an application might perform. HTML's

link navigation and form submission are roughly equivalent to the tasks that Mozilla commands implement.

Although the simplest way to think of a command is as a single JavaScript function, that is far from the only way. A command can also be a message sent from one place to another. This aspect of Mozilla commands is very event-like. Another set of event-like behaviors occurs on top of the basic command system. Change notifications are sent when commands change their state.

Overall, Mozilla's command system is quite complex. Chapter 9, Commands, not only examines this system in the context of a whole XUL window but also covers its nuts and bolts. For now, just note that Mozilla's command system is an extension of, or a complement to, the DOM 3 Events processing model.

6.1.6 Content Delivery Events

The final Mozilla event input system considered in this chapter is the content delivery system. This is the system responsible for accepting a URL and returning a document or for sending an email. It is a general system with several different uses.

DOM events, timed events, and observer events all deal with tiny pieces of information. The event that occurs may represent some larger processing elsewhere in the Mozilla Platform, but the event data itself is usually small. You might call these events lightweight events.

Content delivery events, by comparison, are typically big. They range from email download to a file delivered via FTP to a newsgroup update. Such events, furthermore, are likely broken into subevents: single emails, a partial chunk of an FTPed file, or individual newgroup headers. During the passing of these subevents, the producer and consumer have a long-term relationship. That relationship doesn't end until the last bit of the main job is complete.

Such a heavyweight event system has its own language. Producers are called content sources or data sources; consumers are called content sinks or just sinks. Detailed discussion of sources and sinks is left until Chapter 16, XPCOM Objects.

Having now covered Mozilla's internal event processing, it's time to see how those events can be created by the user.

6.2 How Keystrokes Work

Mozilla's XUL supports the `<keyset>`, `<key>`, `<commandset>`, and `<command>` tags. These tags are used to associate keys with GUI elements and to process keystrokes when they occur. These tags allow key assignments to be changed on a per-document or per-application basis. XUL also supports the `accesskey` attribute.

A key is a small bit of plastic on a keyboard, with some printing on it. Keys can be divided into two rough groups: those with an equivalent glyph and those without. A glyph is a visual representation, like A. The Unicode standard handles keys that have glyphs; for nonglyph keys, like Control-C, there is almost no standards support.

Mozilla's support for keypresses predates the DOM 3 Events standard. That standard defines only keys without glyphs. Mozilla's support includes additional keys that do have glyphs. Mozilla and the standard use different numbering systems, so values for keycodes are different in each. As noted under "XUL Support for Event Handlers" earlier, Mozilla only supports generic `keypress`, `keyup`, and `keydown` events on the `Document` object. These events are not supported on every tag.

Table 6.3 shows the keycode differences between the two definitions. VK stands for Virtual Key.

Table 6.3 Differences between Mozilla and DOM 3 event keycodes

Keycodes specific to Mozilla	Keycodes specific to Mozilla	Keycodes specific to DOM 3 Events
DOM_VK_CANCEL	DOM_VK_SEMICOLON	DOM_VK_UNDEFINED
DOM_VK_HELP	DOM_VK_EQUALS	
DOM_VK_CLEAR	DOM_VK_QUOTE	DOM_VK_RIGHT_ALT
	DOM_VK_MULTIPLY	DOM_VK_LEFT_ALT
DOM_VK_ALT	DOM_VK_ADD	DOM_VK_RIGHT_CONTROL
DOM_VK_CONTROL	DOM_VK_SEPARATOR	DOM_VK_LEFT_CONTROL
DOM_VK_SHIFT	DOM_VK_SUBTRACT	DOM_VK_RIGHT_SHIFT
DOM_VK_META	DOM_VK_DECIMAL	DOM_VK_LEFT_SHIFT
	DOM_VK_DIVIDE	DOM_VK_RIGHT_META
DOM_VK_BACK_SPACE	DOM_VK_COMMA	DOM_VK_LEFT_META
DOM_VK_TAB	DOM_VK_PERIOD	
DOM_VK_RETURN	DOM_VK_SLASH	
	DOM_VK_BACK_QUOTE	
DOM_VK_0 to 9	DOM_VK_OPEN_BRACKET	
DOM_VK_A to Z	DOM_VK_BACK_SLASH	
	DOM_VK_CLOSE_BRACKET	
DOM_VK_NUMPAD0 to 9		

Two further differences between the standard and Mozilla are worth noting:

- The DOM 3 Event `keyval` property matches the Mozilla `nsIDOMKeyEvent` interface's `charCode` property.
- The DOM 3 Event `virtKeyVal` property matches the Mozilla `nsIDOMKeyEvent` interface's `keyCode` property.

These differences are little more than syntax differences.

6.2.1 Where Key Events Come From

A key stroke is translated several times before it becomes a key event. The "Debug Corner" in this chapter explains how to diagnose that translation when it goes wrong. Here we just illustrate the process of collecting a key event.

Keypresses start at the keyboard. Each physical key in a keyboard has its own key number. This number does not match ASCII or anything else. This is the first number generated when a key is pressed.

A keyboard is not as dumb as you might think. It has two-way communication with its computer. Inside the keyboard firmware, a keypress, key release, or key repetition is converted into a scan code. A scan code is a single- or multibyte value. There is no absolute standard for scan codes, just well-known implementations like the IBM PC AT 101 keyboard. Scan codes don't match ASCII or anything else, and they are sent in both directions.

Some scan codes go directly to the hardware, like Delete when used to start BIOS firmware on PC hardware, or sometimes Pause or Control-Alt-Delete. Others are interpreted by the operating system of the computer. An attempt is made to turn interpreted scan codes into a character code—either an ASCII code, a Unicode code, or an internal representation. The Keyboard icon in the Control Panel comes into play in Microsoft Windows. On Linux, the operating system does this using a driver.

Some application software is quite sophisticated. Examples include X-Windows and East Asian (e.g., China, Korea, and Japan) word processors. If the application is sophisticated, almost-raw scan codes might be sent directly to it. In X-Windows, the `xmodmap` utility is responsible for mapping names associated with scan codes to X11's own internal character system. In countries where the character set is larger than the keyboard, a special system called an input method is used. This is a small window that appears when the user presses a "Character compose" key sequence. The window gives visual feedback as the character is created using multiple keystrokes. Non-Western versions of Mozilla have such an input method. An example is Japanese Hiragana, which has thousands of characters, all composed by a keyboard with less than a hundred keys.

6.2.2 What Keys Are Available

After conversion has been attempted, a standards-based character code (ASCII, Unicode, X11, or operating-system internal) holds the key. From that point on, the key itself is less interesting than so-called keyboard maps. These are use statements for individual character codes. Such maps can be added by the operating system, desktop, or individual application.

Some character codes are always interpreted by the operating system, desktop, or window manager keyboard maps. These keys never reach an application. An example is the (Windows) key, used to pop up the Start menu on Microsoft Windows, or Alt-Tab. Pressing such a key is of no use to an application, so application programmers cannot readily implement these keys.

Of the keys that are sent to an application, some will be tied to application-specific functions, like "open addressbook window," and some will be tied to desktop or application keyboard map standards, like "save document." If the application programmer seeks maximum compatibility with desktop conventions, then the first group of keys is available for special tasks. The second group of keys must match existing desktop keyboard maps or else not be used at all.

Keyboard maps often use a modifier key like Control or Alt and another key. Such a modifier key is usually set by convention. This modifier key is not the same on all platforms. Mozilla solves this problem with an accelerator key. That key is bound to the modifier that best suits the current platform. The accelerator key and one or two others like it can be set as user preferences. See the URL *www.mozilla.org/unix/customizing.html* for details.

In the Mozilla Platform, character codes arrive via the native GUI toolkit and are converted into a `DOM_VK` keycode. These application keys are bound to tasks that a given Mozilla window performs. This can be arranged in one of several ways: by using a general keyboard mapping, by assigning keys directly, and by matching keys to an observer. Each of these techniques is described next.

At the highest level, what key does what action is a matter of design. A list of the current key allocation policy for Mozilla can be seen at *www.mozilla.org/projects/ui/accessibility/mozkeyplan.html*.
Click on any key in the top half of that page to get a report of uses for that key in the bottom half.

6.2.2.1 XBL Keyboard Maps

Mozilla's generic keyboard mappings are done in XBL files. The existing examples are not in the chrome: they are platform resources stored under the `res/builtin` directory. There are two files: the general `htmlBindings.xml` and the specific `platformHTMLBindings.xml`. These files apply to the XUL and HTML in existing Mozilla windows and so are good starting points for any application bindings. XBL is described in Chapter 15, XBL Bindings, but a brief example of a keyboard binding is shown in Listing 6.13.

Listing 6.13 Implementing the Redo key command with XBL.

```
<bindings id="myAppKeys">
  <binding id="CmdHistoryKeys">
    <handler event="keypress"
             keycode="DOM_VK_Z"
             command="cmd_redo"
             modifiers="accel,shift"
    />
  </binding>
</bindings>
```

This fragment of XBL creates a group of binding collections called `myAppKeys`, presumable for a specific application. Each collection within the group sets keys for some part of the application. Collecting keys together into several collections encourages reuse. In this case, the `CmdHistoryKeys` collection specifies all the keys required for command history, that is, Undo and Redo operations. Only one such key is shown.

The handler tag is specifically an `onkeypress` event handler in this case. The command that the key is tied to is `cmd_redo`. For now, just note that this is a function implemented somewhere in the platform. The key and modifiers attributes together specify the keypresses required to make Redo occur just once. In this case, it is implemented with three keys: the platform accelerator, any Shift key, and the Z key. In Microsoft Windows notation, this combination is Control-Shift-Z, since the Control key is the accelerator key for Mozilla on Windows.

If the command attribute is left off, then plain JavaScript can be put between open and closing `<handler>` tags. In that case, the script is directly invoked for that keypress. `keycode="DOM_VK_Z"` can be replaced with just `key="z"` as well, since `z` has a character equivalent. If this is done without a modifiers attribute, both `z` and `Z` mean lowercase z.

6.2.3 `<keybinding>`, `<keyset>`, and `<key>`

The `<keybinding>`, `<keyset>`, and `<key>` tags are used to specify how a single XUL document will capture keystrokes. These tags are used more specifically than XBL bindings. The `<key>` tag is the important tag and supports the following attributes:

```
disabled keytext key keycode modifiers oncommand command
```

Except for the `oncommand` attribute, the `<key>` tag acts like a simple data record and has no visual appearance. It is not a user-defined tag because the Mozilla Platform specially processes its contents when it is encountered. Every `<key>` tag should have an `id` attribute, since that is how other tags reference the key.

The `disabled` attribute set to `true` stops the `<key>` tag from doing anything. `keytext` is used only by `<menuitem>` tags and contains a readable

string that describes the key specified. The key attribute contains a single printable keystroke; if it is not specified, then the keycode character is examined for a nonprintable keystroke that has a VK keycode. *In XUL, such a key code starts with VK_, not with DOM_VK_*. key or keycode specify the keyboard input for which the <key> tag stands.

The modifiers attribute can be a space- or comma- separated list of the standard key modifiers and the cross-platform accelerator keys:

```
shift alt meta control accel access
```

Use of accel is recommended if the application is to be cross platform. The access modifier indicates that the user had pressed the shortcut key that exists for the tag in question.

The oncommand attribute is a place to put a JavaScript handler that will fire for the given keystroke. When used for the <key> tag, its meaning is different from the use in Chapter 9, Commands. For <key>, oncommand fires when the keystroke occurs, as you'd expect.

<keyset> is a container tag like <stringbundleset> and <broadcasterset>. It has no special properties of its own. Use it as a tidy container for a set of <key> tags.

<keybinding> is not a true XUL tag at all. Inside Mozilla's chrome, some key definitions are stored as separate .xul files. Rather than have <window> as the outermost tag in such files, <keybinding> is used instead. This is just a naming convention with no special meaning. These keybinding files are added to other XUL files and never display a window on their own.

The <key> tag is most often associated with formlike documents, and this shows. Once a <key> tag is in place, striking that key will fire the oncommand handler for that key, no matter where the mouse cursor is in the document window. There is, however, one restriction. There must be at least one form control in the document, and one of the form controls must have the current focus. This means that at least one of these tags must be present and focused before keys will work:

```
<button> <radio> <checkbox> <textbox> <menulist>
```

It is possible to bend this restriction a little by hiding a single, focused tag with styles so that it takes up zero pixels of space.

There are other XUL tags and attributes that relate to <key>. The accesskey attribute, described in Chapter 7, Forms and Menus, specifies a per-tag key for accessibility use. The <key> tag also works closely with the <menuitem> tag used in drop-down menus.

The following attribute names are used to lodge data with <key> tags, but they have no special meaning any more—if they ever did. They are listed here because they occasionally appear in Mozilla's own chrome. Don't be confused by them—ignore them:

```
shift cancel xulkey charcode
```

6.2.4 The XUL `accesskey` Attribute

XUL provides a feature that supports disabled access. The `accesskey` attribute can be added to many form- or menu-like tags, including `<button>` and `<toolbarbutton>`. It has the same syntax as the `keytext` attribute; in other words, it is specified with a printable character:

```
accesskey="K"
```

`accesskey` keystrokes are collected only if the access system is turned on. This access system is turned on either by custom accessibility equipment connected to the computer, like special input devices, or by pressing the Alt key. This system is described briefly in Chapter 8, Navigation, but is not a large theme of this book.

Individual XUL tags throughout this book note whether they support `accesskey`. Some tags provide a visual hint of the specified `accesskey`; others do not.

6.2.5 Find As You Type (Typeahead Find)

Find As You Type, formerly called Typeahead Find, is a Classic Browser feature in version 1.21 and later that makes extensive use of keystrokes. Its goal is to speed up Web navigation and to assist with accessibility goals. It works only in HTML pages.

Find As You Type is activated with the / key. From that point onward, any printable keys typed are gathered into a string. That string is matched against the text that appears in hypertext links on the page. Pressing Return/Enter causes the currently matching link to be navigated to. This is similar to the search syntax used in UNIX tools like `more`, `less`, and `vi`.

Find As You Type affects state information in the document such as the current focus. This can have unexpected effects if navigation in your document is heavily scripted.

This feature picks up keystrokes during the bubbling phase of DOM Events processing. To stop it from running, call `preventDefault()` on the event after your own key handling code is finished with it.

Find As You Type is complemented by Find Links As You Type. This feature searches only the content of hypertext links. It runs if no / key precedes the search characters, or if the ? key is used instead of the / key.

6.3 How Mouse Gestures Work

Mouse gestures are button clicks, mouse movements, and scroll wheel rolls. Mozilla supports them all. Mozilla also supports graphic tablets and other devices that pretend to be mice.

Mouse gestures can be simple or complex. Just as keystrokes can be put together into a search word with the Find As You Type feature, so too can mouse clicks and movements be put together into a larger action that has its own special meaning. Such a larger action is a true mouse gesture. A simple click by itself is a trivial gesture. Nontrivial examples of gestures include drag-and-drop and content selection.

Trivial gestures are handled in Mozilla using DOM 3 Events. These events are the simplest (atomic) operations out of which larger gestures are built and are listed in Table 6.2 earlier in this chapter. Use them as for any DOM Event. More complex gesture support is examined in the following subtopics.

Why would you want complex gesture support? Users are well educated about several basic gestures, and some applications are expected to support them. The most common example is an image manipulation or drawing program with its bounding boxes and freestyle scribble. Novel or experimental uses of gestures must be handled carefully because there is a risk that the user will become confused.

In most cases, complex mouse gestures must be implemented in JavaScript. Although Mozilla can catch and process a stream of `mousemove` events in JavaScript impressively fast, such processing is also very CPU-intensive. Poorly implemented (or even well-implemented) JavaScript gesture support may be too CPU-intensive to run on older PC hardware.

6.3.1 Content Selection

Content selection is a mouse gesture that picks out a portion of a document, from a starting point to an ending point. Visual feedback, in which content is "blacked" or "highlighted" to indicate the extent of the current selection, is usually provided. Selections can be quite small, comprising a single menu or list item, or even just a few characters. Mozilla supports several types of content selection.

The simplest type is selection of whole data items. This applies to XUL widgets such as menus, listboxes and trees, and HTML equivalents, where they exist. Display any menu, and as you draw the mouse pointer down it, one item at a time is selected. This kind of selection is seen as part of the underlying widget's normal behavior, and the start and end points of the selected content are not generally user-definable. Selecting an item from a menu is something we do without thinking.

More significant content selection occurs where there is semistructured content and the user defines the start and end points of the selection with the mouse. Word processors and text editors are the places where this kind of selection is most common. In such systems, the selected content is separate from the gesture that selects it. A typical gesture for selecting content goes like this:

1. Left-click-press on the starting point.
2. Mouse move to the ending point.
3. Left-click-release on the ending point.

This is not an absolute rule, though. A different gesture with the same effect might be:

1. Left-click down and up on the starting point.
2. Mouse move to the ending point.
3. Right-click down and up on the ending point.

There are many such variations. For example, shift-click is also commonly used to extend a selection from an existing start point to a current point. Mozilla implements two kinds of content selection using one mouse gesture—the press-move-release gesture style. The first of these two selection types occurs inside textboxes. The second kind of selection occurs across all the content of a displayed document.

Textbox selection works inside HTML's `<input type="text">` and `<textarea>` tags and XUL's `<textbox>` tag. The DOM 1 Element for these tags contains properties and methods to examine and set this selected text:

- **value**. The whole content of the textbox.
- **selectionStart**. The character offset of the selection start point.
- **selectionEnd**. The character offset of the selection end point.
- **setSelectionRange(start,end)**. Set the offsets of the start and end points.

The CSS3 draft property `user-select` is also relevant.

General content selection outside widgets is only implemented for the user in HTML. A starting point for examining HTML's support is to look at the `document.getSelection()` method, which returns an object that includes an array of `Range` objects. `Range` objects are defined in the DOM 2 Traversal and Ranges standard.

By default, it is not possible for the user to select XUL content with a mouse gesture. Such a gesture-based feature can be added, though. A starting point is to examine the `createRange()` method of the XUL document object. This creates a DOM `Range` object, which can be manipulated to match any contiguous part of an XUL document tree. This object can be dynamically updated as the mouse moves, and styles can be applied to the subtree the range represents so that the content selected is visually highlighted.

In both HTML and XUL, this form of selection is restricted to visually selecting whole, contiguous lines of content (in other words line boxes; see Chapter 2, XUL Layout), except for the first and last lines, which may be partially selected. The exception to this rule is text that displays vertically, like Hebrew and Chinese, and HTML table columns (see "Multiple Selection").

If you decide to experiment with the DOM Range object, beware that although it is fully implemented, it is used only a little in Mozilla and has not had extensive testing.

6.3.1.1 Rubber Band Selection

Rubber band selection is best known in desktop windows that display icons. By clicking on the background of the window and dragging, a small box with a dotted or dashed border appears. Any icon that falls within that expanding box is considered to be selected. Rubber-banding is a form of content selection, but it is not restricted to selecting whole contiguous lines. It can select any content in a given rectangular area.

Mozilla has no support for rubber band selection at all, but the platform has enough features to implement it. This can be done in HTML or XUL. The XUL case is discussed here.

Since XUL doesn't support layers or CSS2 `z-index` styles, the rubber band box itself is problematic to create. The solution is to ensure that all content inside the XUL document is contained in a single card of a `<stack>` tag (see Chapter 2, XUL Layout). The rubber band box is a `<box>` tag with no contents and a border style. It comprises the sole other card of the `<stack>`. In normal circumstances, this second card has style `visibility:none`.

When the user begins the mouse gesture, scripts makes the bordered box visible. As the drag part of the rubber band gesture proceeds, the width and height properties of the box are changed to match. At each `mousemove` update, a diagnostic routine walks the DOM tree and changes the styling of any tag that falls within the current edges of the rubber band box. Simple.

6.3.1.2 Multiple Selection

Multiple selection is most familiar when it appears inside desktop windows that display icons. On Microsoft Windows or GNOME, for example, several icons can be selected by Control-left-clicking on each one in turn. This is different from rubber-banding, which is a form of single selection.

Mozilla supports multiple selection in `<listbox>` and `<tree>` tags. See Chapter 13, Listboxes and Trees, for a description of those tags at work. Multiple selection for those tags uses Control-left-click.

Mozilla also supports multiple selection in HTML for vertically oriented text. Support for this text, called BiDi text (bi-directional text) does not exist in XUL yet. The gesture for this form of multiple selection is the same as for normal content selection, but the data processing is different. When using the `getSelection()` factory method of the HTML document object, an array of `Range` objects, instead of just one object, is created. These ranges cover one vertical swath (one column of selected content) each. The `getRangeAt()` index method returns any one such range.

Desktop-style Control-left-clicking can easily be implemented in Mozilla. `keydown` and `keyup` events can be used to define the start and end of the gesture, which occur when the Control key is pressed and later released. Click events can be used to identify the items selected while the gesture is active.

Just because the mouse is completely released between clicks doesn't mean that the gesture ends. A gesture ends when the programmer says it ends.

6.3.2 Drag and Drop

Drag and drop is a gesture where a visual element is chosen and moves with the mouse cursor until it is released. An optional aspect of drag and drop is the use of target sites. A target site is a spot in the window where the drag-and-drop operation could successfully end. If target sites exist, they should be highlighted when the dragged object hovers over them. The classic example of a target site is the Trash icon on the Macintosh. The trash goes dark when a document's icon is dragged over it. When target sites are used, the dragged object usually disappears from view when the drop part of the drag-and-drop gesture occurs.

Drag-and-drop gestures can occur within an application window, between application windows, or between windows of different applications. Mozilla's support for drag-and-drop is designed for gestures that stay within one Mozilla window. This support can in theory be extended to gestures between Mozilla windows, but there is only basic support for dragging to or from the desktop or other applications. It is possible to detect when a desktop drag enters or leaves a Mozilla window and collect or send the resulting dragged data.

The major limitation of drag-and-drop gestures in Mozilla is that the item dragged does not follow the cursor. During the drag, Mozilla only provides alternate mouse cursors and occasional style information that hints when the dragged item is over a drop target. This limitation can be worked around using a stack as described in "Rubber Band Selection." Instead of the rubber band occupying a second card of the stack, that card contains a copy of the dragged item. This item can be animated to follow the mouse cursor using the techniques used in Dynamic HTML.

Mozilla's drag-and-drop support is a puzzle of several pieces.

The first piece of the puzzle is events and event handlers. Three events are required for simple drag and drop: `draggesture` (the start of the drag), `dragover` (equivalent to `mouseover`), and `dragdrop` (the end of the drag). Two additional events cover the more complex case where a desktop drag-and-drop operation enters or leaves a Mozilla window. Those two events are `dragenter` and `dragexit`.

The second piece of the puzzle is XPCOM objects. The most important component pair is

```
@mozilla.org/widget/dragservice;1 nsIDragService
```

This is the bit of Mozilla responsible for managing the drag-and-drop gesture while it is in progress.

The `invokeDragSession()` method of the `nsIDragService` interface starts the drag session. For special cases, it can accept an `nsIScriptableRegion` object. This object is a collection of rectangles stated in pixel coordinates.

It is used for the special case where the area dragged over includes a complex widget built entirely with a low-level GUI toolkit. The set of rectangles identify a set of "hotspots" (possible drop targets) on the widget that the widget itself should handle if the gesture tracks the cursor over any of them. This system allows the user to get drag feedback from a widget that can't be scripted, or from a widget where no scripts are present. This is mostly useful for embedded purposes and doesn't do anything in the Mozilla Platform unless a `<tree>` tag is involved. In most cases, use `null` for this `nsIScriptableRegion` object.

The `getCurrentSession()` method returns an `nsIDragSession` object. This object is a simple set of states about the current drag gesture and contains no methods of its own. At most, one drag operation can be in progress per Mozilla window.

The third piece of the puzzle is JavaScript support. Although management of the drag-and-drop gesture is implemented, the consequences of the gesture (the command implied) must be hand-coded. In the simple case, this means setting up the `nsIDragService` and `nsIDragSession` interfaces when the gesture starts, detecting the start and end points of the gesture, providing styles or other feedback while the gesture is in progress, and performing the command implied by the gesture at the end.

In the more complex case, the gesture might result in dragged data being imported to or exported from the desktop so that the scripting that backs the gesture must also copy data to or from an `nsITransferable` interface. This interface works with the desktop's clipboard (the copy-and-paste buffer).

In general, the scripting work for drag and drop is nontrivial. Two JavaScript objects are available to simplify the process.

The chrome file `nsDragAndDrop.js` in `toolkit.jar` implements the `nsDragAndDrop` object, which is installed as a property of the window object. This object provides methods that do most of the housekeeping described so far, including special support for the XUL `<tree>` tag. There is no XPIDL interface definition for this file—just what appears in the `.js` file. It has a method to match each of the drag-and-drop events.

The second object must be created by the application programmer. It is a plain JavaScript object and also has no published XPIDL interface. It must be created with the right methods. This object is a bundle of event handlers, one for each event. Regardless of what pattern it might or might not follow, an object of this kind is created and assigned to a variable, usually with a name like `myAppDNDObserver`.

Together, these two objects reduce each required drag-and-drop handler to a single line of script. The event first passes through the generic code of `nsDragAndDrop` and then into any specific code supplied by `myAppDNDObserver`. To examine this system at work, look at the simple example in Mozilla's address book. Look at uses of `abResultsPaneObserver` (which could be called `abResPaneDNDObserver`) in `messenger.jar` in the chrome.

Table 6.4 shows the equivalences between these handy methods and the basic events.

Table 6.4 Equivalences between Drag events and Script methods

Event	nsDragAndDrop method	-DNDObserver method
Draggesture	startDrag()	onDragStart()
Dragover	dragOver()	onDragOver()
Dragdrop	drop()	onDrop()
dragexit	dragExit()	onDragExit()
dragenter	dragEnter()	onDragEnter()

6.3.3 Window Resizing

Window resizing is a consequence of a mouse gesture that is built into the C/C++ code of Mozilla. See the discussion on the `<resizer>` tag in Chapter 4, First Widgets and Themes.

6.3.4 Advanced Gesture Support

Mouse gestures can be less formal than the examples discussed so far. Like an orchestra conductor waving a baton, specific movements and clicks with the mouse might cause the Classic Browser to execute menu and key commands such as Bookmark This, Go Back, or Save. Support for these kinds of gestures is being considered for the Mozilla Browser as this is written.

In the simplest case, such gesture support consists of chopping mouse movements up into a number of straight strokes. The strokes are identified by dividing the window into an imaginary rectangular grid, with each cell some pixels wide and high. A stroke is considered complete when `mousemove` events indicate that the mouse cursor has left one cell in the grid and entered another. Such strokes form a set of simple instructions that together make a distinctive pattern that is the gesture. After the instruction set is complete, the gesture is identified, and the matching command is executed.

The Optimoz project is such a gesture system and is implemented entirely in JavaScript. It can be examined at *www.mozdev.org*. The code for this extension is small and not difficult to understand. Like Find As You Type, this gesture system grabs events during the bubbling phase, thus avoiding competition with other consumers of the same events.

6.4 Style Options

There are no Mozilla-specific styles that affect event handling. Some of the CSS2 pseudo-styles, like `:active`, are activated by changes to the currently focused and currently selected element.

6.5 HANDS ON: NOTETAKER USER INPUT

In this session, we'll add key support and a few event handlers to the NoteTaker dialog box. There is further discussion on event handlers in "Hands On" in Chapter 13, Listboxes and Trees. There, a systematic approach is considered. This session contains simple introductory code only.

The key support we want comes in two parts: We want the user to have a hint at what keys can be pressed and we want the actual keystrokes to do something.

To add key hints, we use the `accesskey` attribute. This attribute serves as the basis for disabled use of the application, but we're not designing for disabled users at this point. Instead, we're just exploiting the fact that this attribute underlines a useful character of text for us.

To do that, we change all the buttons in the dialog box, as Listing 6.14 shows.

Listing 6.14 Addition of `accesskeys` to NoteTaker.

```
// old
<toolbarbutton label="Edit" onclick="action('edit');"/>
<toolbarbutton label="Keywords" onclick="action('keywords');"/>
<button label="Cancel"/>
<button label="Save"/>

// new
<toolbarbutton label="Edit" accesskey="E" onclick="action('edit');"/>
<toolbarbutton label="Keywords" accesskey="K"
          onclick="action('keywords');"/>
<button label="Cancel" accesskey="C"/>
<button label="Save" accesskey="S"/>
```

We can go further than that one set of changes. We can also highlight keys in the body of the edit area using the `<label>` tag. This means changing content that reads

```
<description>Summary</description>
```

to

```
<label value="Summary" accesskey="u"/>
```

In this case, the key for `Summary` can't be `S` because we've used that already for `"Save"`. When this change is made to all four headings, the resulting window looks like that in Figure 6.3.

After these changes, pressing the underlined keys still has no effect. To change that, we need to add some `<key>` tags, as shown in Listing 6.15.

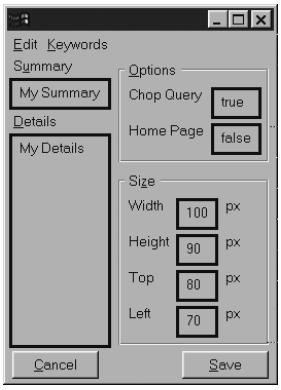


Fig. 6.3 NoteTaker dialog box with available keys highlighted.

Listing 6.15 Tying NoteTaker keys to actions.

```
<keyset>
  <key key="e" oncommand="action('edit')"/>
  <key key="k" oncommand="action('keywords')"/>
  <key key="c" oncommand="action('cancel')"/>
  <key key="s" oncommand="action('save')"/>
  <key key="u" oncommand="action('summary')"/>
  <key key="d" oncommand="action('details')"/>
  <key key="o" oncommand="action('options')"/>
  <key key="z" oncommand="action('size')"/>
</keyset>
```

The `key` attribute is not case-sensitive; if we explicitly want a capital S, then we would add a `modifiers="shift"` attribute. If we wanted to use F1, then we'd use the `keycode` attribute and a `VK` symbol instead of `key` and a printable character. We're able to reuse some code from previous chapters because the function `action()` accepts a single instruction as argument. If `action()` is modified to include a line like this

```
alert("Action: " + task);
```

then it is immediately obvious whether pressing a given key has worked. We'll defer the cancel and save instructions to a later chapter. To practice event handling in a bit more depth, we'll experiment with the other four keys. In later chapters, this experimentation will become far easier.

It would be convenient if the user didn't need to consider every detail of this dialog box. Perhaps there are some defaults at work, or perhaps some of the information is not always relevant. Either way, it would be nice if the user could de-emphasize some of the content, with a keystroke or mouse click. That is what we'll implement.

Our strategy is to deemphasize content using style information. When we pick up the user input, we'll change the class attribute of the content so that new style rules are used. When the user repeats the instruction, we'll toggle back to the previous appearance. We intend to apply this rule to the boxed content only. The deemphasizing style rule is

```
.disabled {
  border : solid; padding : 2px; margin : 2px;
  border-color : darkgrey; color : darkgrey
}
```

There are four areas on the edit panel (Summary, Details, Options, and Size) and a total of eight displayed boxes, each of which might need an update. We can group the eight back into four using broadcasters and observers. We add the broadcasters in Listing 6.16, which specify nothing to start with.

Listing 6.16 Broadcasters for disabling NoteTaker dialog subpanels.

```
<broadcasterset>
  <broadcaster id="toggle-summary"/>
  <broadcaster id="toggle-details"/>
  <broadcaster id="toggle-options"/>
  <broadcaster id="toggle-size"/>
</broadcasterset>
```

We add `observes=` attributes on all eight boxes, tying each one to one of the four broadcasters, so

```
<box id="dialog.top" class="temporary">
```

becomes

```
<box id="dialog.top" class="temporary" observes="toggle-size"/>
```

If any broadcaster gains a `class="disabled"` attribute, that attribute will be passed to the boxes observing it, and those boxes will change style. We can test this system by temporarily hand-setting the `class="disabled"` attribute on any broadcaster. If that is done for the last broadcaster (`"toggle-size"`), then the dialog box displays as in Figure 6.4.

Now that this code is working, we can hook it up to the user input. The keystrokes are first. For those, we update the `action()` method. Listing 6.17 shows the new code.

Listing 6.17 Code to toggle gray-out state of NoteTaker dialog subpanels.

```
function action(task)
{
  if ( task == "edit" || task == "keywords" )
  {
    var card = document.getElementById("dialog." + task);
    var deck = card.parentNode;
```

```
    if ( task == "edit" )    deck.selectedIndex = 0;
    if ( task == "keywords") deck.selectedIndex = 1;
  }

  if ( task == "summary" || task == "details" || task == "options" || task
          == "size" )
  {
    var bc = document.getElementById("toggle-" + task);
    var style = bc.getAttribute("class");

    if ( style == "" || style == "temporary" )
      bc.setAttribute("class","disabled");
    else
      bc.setAttribute("class","temporary");
  }
}
```

This code modifies the broadcaster tags only, and the platform takes care of the rest. If necessary, we can test this code by directly calling the `action()` method, perhaps from the `onload` handler of the `<window>` tag. It's just as easy to test by pressing the correct key.

Supporting the mouse is a little trickier. Exactly where in the dialog box is a user click supposed to have an effect? We could install an event handler on every tag, but that isn't very clever. Instead we'll use the XUL layout model and the DOM Event model. We'll make sure that each of the subpanels is contained in a single XUL tag—that's the layout part. And, at the top of the document tree, we'll install an event handler that will catch any `onclick` events and figure out which of the subpanels applies—that is the DOM Event part.

First, we will look at the XUL layout part. The Summary and Details subpanels aren't contained in a single box, so we'll just wrap another `<vbox>` around each `<label>` plus `<box>` pair. That's unnecessary from a display

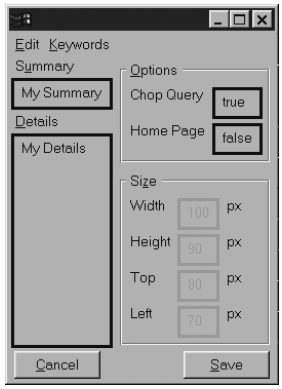


Fig. 6.4 NoteTaker dialog box with content grayed out.

point of view, but it's convenient for our code. Now we have two subpanels contained within <vbox> tags and two within <groupbox> tags. We'll add a subpanel attribute with the name of the subpanel to each of these four tags:

```
<groupbox subpanel="size"/>
```

This attribute has no special meaning to XUL; we just made it up. It's a conveniently placed piece of constant data.

Second, we'll write an event-handling function to catch the mouse click. We want to catch it right at the start, which means during the capture phase. The XUL `onclick` attribute is good only for the bubbling phase, so we'll have to use `addEventListener()` from JavaScript. The handler function and handler installation code is shown in Listing 6.18.

Listing 6.18 Code to capture click events and deduce the action to run.

```
function handle_click(e)
{
  var task = "";
  var tag = e.target;
  while ( (task = tag.getAttribute("subpanel")) == ""
          && tag.tagName != "window" )
    tag = tag.parentNode;

  if ( task != "")
    action(task);
}

document.addEventListener("click", handle_click, true);
```

Since this handler is installed on the `Document` object (the <window> tag) and set for the capture phase, it will receive `onclick` events before anything else. It looks at the passed-in DOM 2 Event object and determines what the target of the event is. That target is the most specific tag that lies under the user's click. The code then walks up the DOM tree from that target, looking for a tag with a subpanel attribute. If it finds one, it runs the action in that attribute. If it doesn't find one, nothing happens. The event then continues its normal progress through the tree.

Because the actions are the same for each of the subparts of the dialog box, the resulting code is very general. Instead of four, eight, or more event handlers, we've succeeded with only one. Again, savings have resulted from the passing of command names into a general routine. We can't expect these savings all the time, but good design should always yield some. We've used four tag names, three attribute names, and two functions to achieve all of this.

That concludes the "Hands On" session. We won't keep all the experiments we've made here, but in general terms, they are valuable practice.

6.6 DEBUG CORNER: DETECTING EVENTS

Detecting events is a trivial process in Mozilla.

First, ensure that you have the preference `browser.dom.window.dump.enabled` set to `true`. Start the platform with the `-console` option. Second, write a one-line diagnostic function:

```
function edump(e) { dump(e.type+":"+e.target.tagName); }
```

Third, install this function as a handler on the window object for every event that interests you. Install it via `addEventListener()` if you want to see the events during the capture phase as well. Finally, start the browser from the command line, and watch the flood of events appear in that command-line window as you click and type.

To diagnose broadcast events, install handlers for the `oncommand` event, or just add diagnostic observers to the `nsIObserverService` object. Currently, there is no way to observe all the topic names that are broadcast from this global broadcasters in the standard platform.

In a debug version of the platform, it is possible to create a log file of all broadcasts using these environment variables (see the source file `prlog.h` for more details):

```
set NSPR_LOG_MODULES=ObserverService:5
set NSPR_LOGFILE=output.log
```

6.6.1 Diagnosing Key Problems

Nothing is more frustrating than a key that doesn't work. The application, whether Mozilla or something written on top of it, often receives the blame because it is the intended destination of the keystroke. Some problems, however, have nothing to do with Mozilla.

At the lowest level, a physical key can be damaged or its contact ruined. Test the key in some other application (e.g., in `vi` enter insert mode (`'i'`), press Control-V, and then type the key desired).

The software drivers in the keyboard firmware and in the operating system can be updated and, on rare occasions, may be incorrect. Read the Linux Keyboard HOWTO document for more details on keyboard technology.

To test if the operating system is receiving the key correctly, start Windows in DOS-only mode or login to Linux from the console without starting X11. On Windows, the Edit character-mode editor can be used to test non-ASCII keys.

To test foreign language keyboard issues, it is best to ask a native speaker for help. The `soc.culture` USENET newsgroups are good places to put a polite request. Some languages receive less support than others, and users of those languages are occasionally keen to help out. The Mozilla bugZilla bug database (*http://bugzilla.mozilla.org*) has captured many discussions on the topics of internationalization, compatibility, and localization.

To test if a desktop application is receiving a key, test the key using a terminal emulator (e.g., a DOS box, `xterm`, or `gnome-terminal`) in "raw" mode. Raw mode exists when an editor like Edit or `vi` runs in that terminal. That terminal emulator relies on the desktop (or at least the windowing system) for input.

To test if Mozilla is receiving a key, just add an event listener for that key to the document object of a chrome window.

6.7 SUMMARY

Programmers can't get access to user-typed data by admiring a widget; they need some kind of internal structure to work with. The Mozilla Platform is built on an event-oriented architecture from the very beginning and provides several different event models for the programmer to work with.

The most powerful and obvious of these systems is the DOM 3 Event model and associated HTML events, most of which are applicable to Mozilla's XUL. This event system is sufficient for the programmer to capture raw user keypresses and mouse gestures.

Mozilla also supports less obvious event processing. From the world of software design patterns, there are several models that allow the programmer to handle input and change notifications. Generally speaking, these systems have at their cores the producer-consumer concept for data processing. The most noteworthy of these systems is the observer-based system that allows several tags (or programming entities) to be advised of changes that occur at one central place. This system lays the foundation for understanding the more sophisticated command system that is at the heart of Mozilla applications.

Raw processing of events is fairly primitive. Most events occur in association with a useful widget. In the next chapter, we look at Mozilla's most useful widgets—forms and menus.