Extracted from:

Prototype and script.aculo.us

You Never Knew JavaScript Could Do This!

Chapter 7

Playing with the DOM Is *Finally* Fun!

When writing modern web applications, you find yourself doing a *lot* of DOM manipulation. Traversing the DOM, fetching elements, showing and hiding them, replacing fragments of the document with new (X)HTML contents, fiddling with CSS class names... this is what we web developers *do*.

Unfortunately, if we stick with the raw standards (such as DOM Level 2), we find ourselves tragically underequiped. The tools of the trade were judged and found wanting. It feels like building a skyscraper with cardboard and string.

But you're using Prototype now.

True to its aim, Prototype comes with plenty of nifty tools you can use to tweak the DOM. At the heart of it is the notion of *DOM extension*. The idea is simple: one way or another, you can get "extended" versions of the original DOM nodes, and these versions are *way* easier to play with than their bare-bones counterparts. At the time of this writing, there are 45+ extension-provided methods in there.

7.1 Extending DOM Elements

Let's first focus on the net result for the web developer: fetching an element through the $() function (which we saw in Section 3.2, *Quick Fetching of Smart Elements with $*, on page 44) *guarantees* that what you get is the extended version of the original DOM element.

An extended element is a DOM element that also features all the methods we will see in the next section (plus extra ones if it's a form or form field element, as we'll see in Chapter 8, *Form Management*, on page 175). It is not a fresh object, distinct from the original DOM node. It's the same node but augmented with Prototype's extensions.

I could discourse for pages about Prototype's extension mechanism, but this would be slightly beyond the scope of this book. So instead of entering into the nitty-gritty details of stuff like Element.Methods, Element.Methods.Simulated, or Element._attributeTranslations, let me answer the most common questions first.

Speed Cost

On browsers providing DOM element types with a prototype, the cost is close to zero. Prototype automatically extends the relevant prototypes at loading time, which is blazing fast. This is, most notably, the case of all Gecko-based browsers (and hence Firefox), Opera (at least from version 9), Konqueror, and Safari (although specific versions of Safari may handle this in a specific way, the particulars are addressed by Prototype, and the speed cost is roughly identical).

For browsers with no such support (for example, Internet Explorer), the element is extended on the fly the first time it is requested as an extended element (either through the $() function or through a direct call to Element.extend(), which your own code should never need to do). Such an extension request can very well happen inside Prototype's code, because numerous methods in Prototype return extended elements. The element is then marked as extended, and there will be no further cost associated with requesting it as an extended element.

However, on-the-fly element extension is not a trivial cost in itself, and when applied over a large number of elements (depending on your environment, this can be anywhere between 100 and 1,000 elements), the speed hit can be noticeable. So, you should refrain from needlessly relying on $() (or other methods that guarantee extended results) when working with very large sets of elements. All extended methods can be called indirectly on "raw" elements (but because they will use $() over the element internally, what may have been raw before is now extended anyway. . .).

What If These Methods Exist Natively?

Simple as Sunday: they're left as is. Prototype's method extensions apply only when there is no native version present in order to maximize execution speed.

7.2 Element, Your New Best Friend

Your gateway to DOM extension is the Element namespace. It contains the DOM extension machinery and the repositories for the extra methods (mostly Element.Methods).

Calling the Methods

All those methods can be used in two ways:

- As vanilla functions, which can be passed any DOM element (including, most important, nonextended ones) as their first argument. The easiest way is to call them through the Element namespace, like this:

```
Element.remove(elt);
Element.next(elt, 'li');
```

- As methods over extended elements, which certainly feels more like object-oriented programming:

```
$(elt).remove();
$(elt).next('li');
```

All *mutative* methods return their original element extended. A mutative method alters the element in some way. Methods returning elements (for example, fetching the descendant elements of the one passed as first argument) return extended elements, too. This makes method chaining easy:

```
$(elt).next('li').remove();
```

Building a Staff Manager

To get familiar with most methods provided through DOM extension, we'll build a complete example that heavily relies on it. The idea is to have a simple web page that lets us describe people and groups of people. Groups can be nested to an arbitrary depth.

Our page lets us see the whole staff using a tree representation, on the left, and lets us create new groups and people, as well as rename existing ones, through a small editor zone next to the tree. Naturally, all

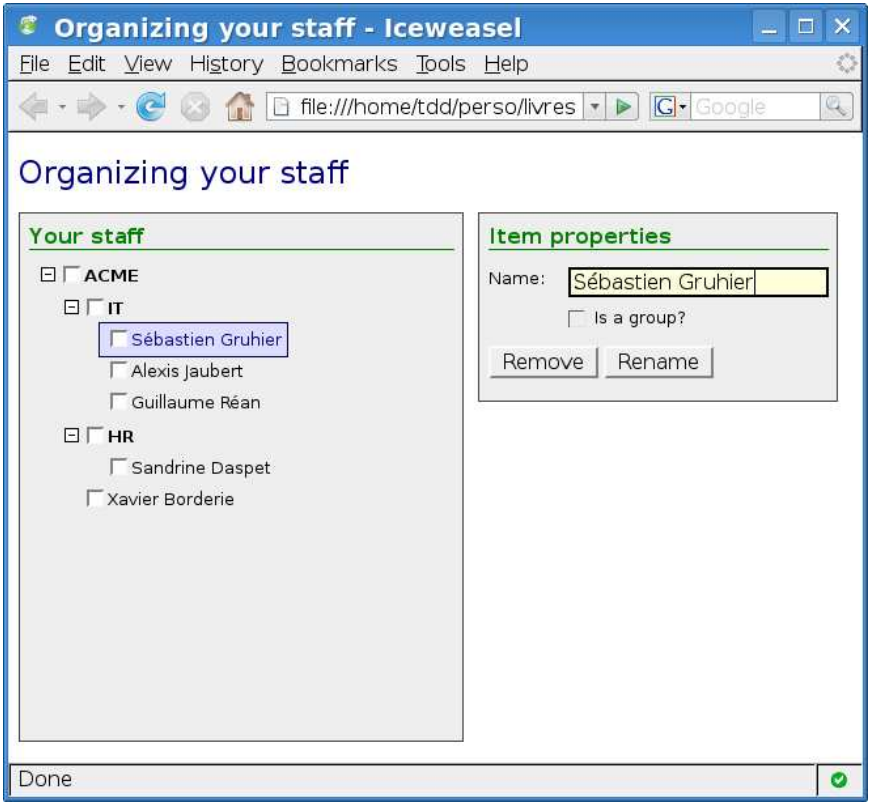

Figure 7.1: Our finished screen

groups in the tree can be collapsed and expanded. All nodes can also be checked using a plain checkbox. This opens the door to further use of the data (for example, we could use this to select to whom to send an e-mail).

The completed screen looks like Figure 7.1. Note that we provide a highlighted representation of the selected node. To build this tool, we will need to follow several steps:

1. Create the HTML file for our screen.
2. Create a JavaScript representation of our data tree.
3. Write a function that takes the JavaScript representation of a person or group and inserts the corresponding DOM fragment in the document.

4. Handle clicks anywhere in the tree to deal with group togglers (those little +/- signs that let us expand or collapse groups) but also select (or deselect) nodes.
5. Maintain editor state depending on the currently selected node (buttons may be disabled or enabled, information needs to be pre-filled when a node gets selected).
6. Handle uses of the form on the right in order to deal with node creation, renaming, or removal.

Of course, we'll do all this with the proper double take of polish, making sure the user experience is as smooth as possible and trying to leverage Prototype's features as much as possible.

Because the primary goal of this chapter is to acquaint you with Prototype's DOM extensions, we will not add an extra layer of complexity by using Ajax to deal with server-side data. However, in a later chapter, we will come back to this example and turn it into an actual client/server application, using Ajax for a snappy user experience. Data will not reside only as JavaScript objects on the client side but be stored on the server side. This will let us tinker away with form serialization methods and most Ajax-related utilities.

Laying the Groundwork: Our HTML Page

The markup for our screen is fairly simple: a title, proper definition of charset, binding on the style sheets and scripts, and two zones (the tree and the editor form). Here you go:

Download prototype/dom/people.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>Organizing your staff</title>
  <link rel="stylesheet" type="text/css" href="people.css" />
  <script type="text/javascript" src="../../prototype.js"></script>
  <script type="text/javascript" src="people.js"></script>
</head>
<body>
  <h1>Organizing your staff</h1>

  <div id="tree">
    <h2>Your staff</h2>
    <form id="staff">
      <ul></ul>
    </form>
  </div>
```

```
  <div id="props">
    <h2>Item properties</h2>
    <form id="editor">
      <p>
        <label for="edtName" accesskey="N">Name:</label>
        <input type="text" id="edtName" />
      </p>
      <p>
        <input type="checkbox" id="chkIsGroup" />
        <label for="chkIsGroup" id="lblIsGroup"
         accesskey="G">Is a group?</label>
      </p>
      <p>
        <input type="button" id="btnRemove" value="Remove"
         accesskey="R" />
        <input type="button" id="btnAddChild" value="Add as child"
         accesskey="C" />
        <input type="submit" id="btnSubmit" value="Create" />
      </p>
    </form>
  </div>
</body>
</html>
```

We use a <*form*> element in the tree because we're going to put checkboxes in there and strict HTML mandates that form fields be located in forms (which rather makes sense). We also nest a <*ul*> element because in our tree, all group-like levels (be it the root level, like here, or a regular group level) use a <*ul*> to contain their children.

This is because we're going to represent our tree with proper semantic markup: using nested lists. Since we have no specific ordering requirements, we use <*ul*> instead of <*ol*>. Each item in such a list is a <*li*>, inside which all item contents (including sublists) are located.

The markup for our editor form is fairly short as well: a text field, a checkbox, and three buttons (two regular ones that need to be specifically activated, which will trigger the removal of the currently selected element and the creation of a new node below this same element, respectively, and a submission button, which is activated whenever the user hits the `Return` key in the text field or the checkbox in addition to plain old clicking. . .). That submission button either creates an element at root level (when no element is selected) or renames the currently selected element. This makes for faster batch-oriented operation.

So far, with no styling, the page is a mess. Let's add some CSS magic:

Download prototype/dom/people.css

```
Line 1  body { font-family: sans-serif; font-size: small; }
-       h1 { color: navy; font-size: x-large; font-weight: normal; }
-       h2 {
-         color: green; font-size: larger;
5       border-bottom: 1px solid green; margin: 0 0 0.5em;
-       }
-       img { border: 0; }
-
-       #tree {
10        width: 25em; height: 30em; overflow: auto; float: left;
-         border: 1px solid #444; background: #eee; padding: 0.5em;
-         cursor: default;
-       }
-
15      #props {
-         width: 25em; height: 10em; margin-left: 27em;
-         border: 1px solid #444; background: #eee; padding: 0.5em;
-       }
-
20      #tree ul {
-         list-style-type: none;
-         margin: 0; padding: 0;
-       }
-       #tree ul ul { padding-left: 1.3em; }
25      #tree li { padding-left: 0.1em; margin: 0.4em 0; }
-       #tree span { padding: 5px; }
-
-       span.group { font-weight: bold; }
-
30      #tree span.person { font-weight: normal; margin-left: 16px; }
-
-       #tree span.selected {
-         border: 1px solid #004; padding: 4px; background: #ddf;
-         color: navy;
35      }
-
-       #editor p { position: relative; height: 1.3em; }
-       #edtName, #chkIsGroup { position: absolute; left: 4em; margin-left: 0; }
-       #edtName { padding: 0 0.1em; right: 0; }
40      #edtName:focus, #edtName:active { border: 2px solid black;
-         background: #ffd; }
-       #lblIsGroup { position: absolute; left: 6.3em; }
```

Some of this is not immediately useful, because it relates to elements that will be created dynamically by script to represent tree nodes (those are the lines 24 to 35). The rest is styling as usual. Our page is now ready for life to be breathed into it, thanks to scripting.

Representing the Staff: Our Staff Object

We'll put most of the functionality of staff management into a custom object, which we'll call, quite simply, Staff. Inside it, we'll put many methods, plus the actual data structure, tucked neatly in a nodes field. It is an array of "tree nodes," each of which is a simple object with at least two properties: id and name.

The id property matches the id= attribute of the <*li*> elements representing the tree node in the screen and is of the form *itemXXX*, where *XXX* is an incrementally generated integer. The name property holds the tree node's name, its visible label.

If a tree node is actually a group, it also features a children property, which is an array. Such an array holds tree node objects for anything inside the group, and so on and so forth, recursively.

Let us start by defining a default tree with data for the staff of an imaginary company, ACME.[1] This goes like this:

Download prototype/dom/fragments/people_1.js

```
var Staff = {
  nodes: [
    { id: 'item1', name: 'ACME',
      children: [
        { id: 'item11', name: 'IT',
          children: [
            { id: 'item111', name: 'Sébastien Gruhier' },
            { id: 'item112', name: 'Alexis Jaubert' },
            { id: 'item113', name: 'Guillaume Réan' }
          ] },
        { id: 'item12', name: 'HR',
          children: [
            { id: 'item121', name: 'Sandrine Daspet' }
          ] },
        { id: 'item13', name: 'Xavier Borderie' }
      ] },
  ]
}; // Staff
```

Here we are: our staff is represented in Staff.nodes. The next step is to turn this data into actual tree nodes on the screen. . . .

1. Boy, that's *groundbreaking*.

Walking Around: Moving Across the DOM

```
down([selector = '*'] [, index = 0]) → HTMLElement
firstDescendant() → HTMLElement
next([selector = '*'] [, index = 0]) → HTMLElement
previous([selector = '*'] [, index = 0]) → HTMLElement
up([selector = '*'] [, index = 0]) → HTMLElement
```

To build and manipulate DOM fragments based on this JavaScript data structure, we need to learn about two categories of methods in Element: those that let us walk the DOM easily and those that let us alter the contents of elements.

Bare-bones DOM walking is quite the nightmare: the properties provided by the W3C specification—firstChild, lastChild, childNodes, previousSibling, and nextSibling—work only at the node level, not at the element level. The immediate consequence of this low-level attitude is that we end up walking through empty text nodes produced by markup formatting (for example, line breaks and indentation), comment nodes, entity references, and so forth. This is indeed unfortunate, because in the vast majority of cases, we concern ourselves only with elements. Not only that, but we usually want to reach for a specific kind of element (for example, a <*ul*> or <*a*> element).

Prototype extends DOM elements with methods that let us do just that (it also lets us look at whole element chains in all directions, as we'll see in Section 7.2, *Meeting the Family: Ancestors, Children, Siblings...*, on page 162). These are named up(), down(), next(), and previous(), and all share they same signature:

- With no argument, they get you to the closest element in their direction.
- With a string argument, they interpret it as a CSS selector, relying on the amazing capabilities of the Selector class, which we will explore in greater depth on page 169. A common form of selector in this context is a simple tag name.
- With an integer argument, they get you to the *index*th element in their direction.
- With two arguments, a string and an integer, they get you to the *index*th element among those obtained by the selector, counting from the current element outward.

There is also an optimized method for a common use case, which just needs the first child element, with no additional requirement. It's covered by firstDescendant().

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Prototype and script.aculo.us's Home Page
http://pragprog.com/titles/cppsu
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/cppsu.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	orders@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com