Extracted from:

Prototype and script.aculo.us

You Never Knew JavaScript Could Do This!

Furious activity is no substitute for understanding.
► H. H. Williams

Chapter 2
Discovering Prototype

This part provides in-depth coverage of Prototype, which is the JavaScript library at the core of this book. Prototype is a very *dense* library: although rather small (at about 120KB raw, less than 30KB gzipped, it is no huge framework), it is replete with features, helper objects, and nifty tools, arranged in a reasonably consistent set.

But before we go ahead, we need to answer a few questions and tackle the more involved subjects with a clear mind and proper expectations. For example, what's Prototype exactly? What should we expect it to do for us? What kind of lingo may we need to learn? And apparently it relies on. . . well, prototypes, so what *are* JavaScript prototypes in the first place? So, I'll start with explaining all this quickly; you will then be armed with everything necessary to fully leverage the following chapters.

2.1 What Is Prototype, and What Is It Not?

Prototype is a JavaScript *library* designed to improve the browser's JavaScript environment; it extends DOM elements and built-in types with useful methods, has built-in support for class-style OOP (including inheritance), advanced support for event management, and powerful Ajax features.

Prototype is *not* a complete application development framework: it does not provide widgets or a full set of standard algorithms, I/O systems, or what have you. It stands in this middle ground between down-and-dirty manual coding of everything and full-fledged frameworks with their countless objects. Most massive frameworks do indeed use Prototype internally and build upon it.

Note, however, that there is a more visual-oriented library working closely with Prototype called script.aculo.us; we'll explore it in the second part of this book.

Although inspired by the Ruby programming language, Prototype is *not* attached to any server-side technology. True, it stems from the Ruby on Rails universe, but it is a stand-alone spin-off. It is indeed very easy to use Prototype when coding with Ruby on Rails, but the library can be used with no difficulty over any back end, such as PHP, J2EE, or ASP.NET. It is very successfully used in production for projects with all these technologies and more.

Prototype is distributed as a single file called prototype.js, currently weighing about 120KB (before any sort of packing or gzipping). Despite this relative litheness, it provides a large set of features, most of which interoperate in an intuitive way.

2.2 Using Prototype in Our Project

So, how do we go about enabling Prototype in a web page? It is really quite simple: we just need to load prototype.js, and loading it first will let us leverage its power in any other scripts we have. This loading is best done with a simple <*script*> element in the <*head*> of our page:

```
<head>
  ...
  <script type="text/javascript" src=".../prototype.js"></script>
  ...
</head>
```

Where Can We Get Prototype?

The official website is the authoritative source for the latest public version of Prototype and also provides detailed, up-to-date API documentation with plenty of examples, tutorial-style articles, and a blog updated by the Prototype Core team. It's located at http://prototypejs.org.

2.3 What Does Our JavaScript Look Like When Using Prototype?

Good question. To make a long story short, it looks *darn good*. It looks nifty. It looks smart. It looks Rubyesque. JavaScript is fun again. But don't take my word for it—see for yourself. Let's look at a simple example and then at a more involved, combined demo that will help you understand just how easy Prototype coding is.

A Note About Versions

This book covers Prototype 1.6. To understand how Prototype evolved, and where it's headed, it's worth looking at a short history.

The release of version 1.5, on January 18, 2007, was a major event for people using only the public versions. They had been stuck with 1.4 for a year, and 1.5 brought about a tremendous amount of improvements and new features.

These days, Prototype is rapidly pacing ahead, moving in swifter, shorter steps. Version 1.5.1 was released in April 2007 and brought a few new features and significant refactoring and cleanup of the code base. Version 1.5.1.1, a bug-fix release with a few nice surgical improvements to boot, was released in June. With a first release candidate in early August 2007 and a final release scheduled in October 2007, version 1.6 is a major step ahead. It introduces a complete overhaul of the event system, the first improvements on sub-classing, and many more new features. Prototype Core is considering a later 1.6.1 release with yet more event- and class-related improvements, and then we'll be done with the 1.*x* branch. The next steps will take us to 2.0. And we're hard at work on it already!

The information in this book is current at the time we're about to go to press. This means by the time this book is out, you're at worst one or two months behind; in other words, you're up-to-date on 95% of the library and have only to peruse the recent items in the change log to be on the very top of things.

You can get additional information on later releases and feature updates on the book's website and blog at: thebungeebook.net.

An important note: the code in the following two examples is intentionally heavy on Prototype "magic," which means it might use advanced syntaxes and concepts that you may not—yet—be familiar with. Fear not, however: this was done to let you feel the might of properly leveraging what Prototype has to offer you, and we'll dive together, in detail, into these capabilities and syntaxes in the following chapters. If some of the code is unclear as you go through this chapter, I'm confident you'll be able to come back and squeeze every ounce of meaning out of it once you're through the Prototype part of the book. In the meantime, I did try to lace the text with enough explanations that you can grab the idea and general dynamics of the code.

A Simple Example: Playing with People

Er, this sounds like an invitation to use pyramid scams on unsuspecting strangers. Actually, I just suggest we put together a simple class representing a person, then start spawning a few people with it, and finally fiddle with the resulting population to extract a few pieces of information. We'll do all of it the Prototype way.

I bet you could use some code before deciding whether what I just said made any sense. So, let's create an empty folder, put Prototype's prototype.js in it (version 1.5.1 or later), and write the following bench page for us to play in:

Download prototype/intro/basic/index.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-US" xml:lang="en-US">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>A basic demo of Prototype at work</title>
    <link rel="stylesheet" type="text/css" href="basic.css" />
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="basic.js"></script>
</head>
<body>

<h1>A basic demo of Prototype at work</h1>

<div id="result"></div>

</body>
</html>
```

The <*div*> with id="result" is just a placeholder for our upcoming script to spew HTML into.

Now, let's create this basic.js we referenced and write a Person class. In Prototype, we would do it this way:

Download prototype/intro/basic/basic.js

```
Line 1  var Person = Class.create({
     -    initialize: function(first, last, city, country) {
     -      this.first = first;
     -      this.last = last;
     5      this.city = city;
     -      this.country = country;
     -    },
     -
     -    getFullName: function() {
    10      return (this.first + ' ' + this.last).strip();
     -    },
     -
     -    getDisplayName: function() {
     -      var result = this.getFullName();
    15      if (this.city || this.country) {
     -        result += ' (';
     -        if (this.city) {
     -          result += this.city;
     -          if (this.country) result += ', ';
    20        }
     -        result += (this.country || '');
     -        result += ')';
     -      }
     -      return result;
    25    }
     -  });
```

This first fragment deserves some explanation:

- The Class.create() call on line 1 produces a Prototype class. For the JavaScript gurus among you, yes, that is a function object.
- When using Prototype classes, initialization is taken care of via a initialize() method, here on line 2, which receives all the arguments passed at construction time.
- Finally, our getDisplayName() method, starting on line 13, builds a variable-form string representation of the person, with the first name and/or last name and possibly city/country information between parentheses, all of it properly formatted and adjusted.

Being defined at the prototype level, all of these methods are basically *instance methods*. We'll add a *class method* (or *static method*) that provides a comparator between two people.

Just to make our code more “Prototypish” and to demonstrate neat JavaScript usage, we’ll make it conform to the following usage syntax:

```
Person.compare([criterion = 'first',] p1, p2) → (-1|0|1)
```

Now, that’s unusual—optional arguments appearing first! It’s actually easy to deal with once you regard your arguments as just an array of values, much like Ruby would allow. Here is the code:

Download prototype/intro/basic/basic.js

```
Line 1   Person.compare = function() {
   -       var prop = 'first', args = $A(arguments);
   -       if (args.length == 3 && typeof args[0] == 'string')
   -         prop = args.shift();
   5       var c1 = args[0][prop], c2 = args[1][prop];
   -       return (c1 < c2 ? -1 : (c2 < c1 ? 1 : 0));
   -     };
```

As you may know, functions in JavaScript get an automatic arguments variable that holds their arguments. It’s not an array properly speaking, but it looks like one (in other words, it features a [] operator and a length property), so we can readily convert it to an actual array with Prototype’s $A() utility function, as shown on line 2.

Prototype-enhanced arrays are mighty to say the least, but in this particular occasion all we need is their native shift() method, which will take the first element out and return it.

By simply checking whether there are three arguments instead of two, with a String-typed first one, we know we’ve been called with an explicit field name as the comparison criterion. So, we override our prop variable with the first argument, which we take out of the argument list at the same time.

Now that we have the name of the field we’re going to use for comparison, we need to dynamically access it for each of the two people we’re about to compare. This is trivially done in JavaScript with the square brackets operator, [], which we use on line 5. When used on an object, it takes an expression that evaluates to the name of a property in the object, and it returns the value of that property.

Finally, using nested ternary operators (?:), we return -1 if the first object looks lesser, 1 if it looks greater, and zero otherwise.

It's time we spawn a whole series of people to tinker with:

Download prototype/intro/basic/basic.js

```
var people = [
  new Person('Jes "Canllaith"', 'Hall', 'Wellington', 'NZ'),
  new Person('Sebastien', 'Gruhier', 'Carquefou', 'FR'),
  new Person('Clotile', 'Michel'),
  new Person('Stéphane', 'Akkaoui', 'Paris'),
  new Person('Elodie', 'Jaubert', 'Paris')
];
```

Notice how we do not need to pass all the arguments every time and how the objects are constructed: through the traditional **new** keyword.

OK, we're all set. We can now start playing with Prototype-induced power. For instance, let's say we need to get a sorted list of all the first names for these people, with no risk of duplicates:

Download prototype/intro/basic/basic.js

```
people.pluck('first').sort().uniq().join(', ')
// => 'Clotilde, Elodie, Jes "Canllaith", Sebastien, Stéphane'
```

Doesn't this rock? The pluck() method fetches a given property from all the objects in the series and returns an array of the resulting values. uniq() strips out duplicates. This is rather concise, don't you think?

How about getting full information on all people with a defined country, sorted by ascending country code:

Download prototype/intro/basic/basic.js

```
people.findAll(function(n) { return n.country; })
  .sort(Person.compare.bind(Person, 'country')).invoke('getDisplayName')
// => [ 'Sebastien Gruhier (Carquefou, FR)',
//      'Jes "Canllaith" Hall (Wellington, NZ)' ]
```

The findAll() method takes a predicate (a function taking an element and returning a boolean about it) and returns all the elements that passed it. Here, our predicate just returns each person's country property, whose value may very well be **undefined**. If it holds a nonempty string, it will be deemed **true**, so the predicate will pass. Otherwise, the predicate will fail.

Perhaps you come from a programming background with languages that do not have *higher-order functions*, meaning you can use functions as regular values to be assigned, passed around as arguments to other functions, returned as result values, and so on.

JavaScript, like many dynamic languages, has that important feature, so we can indeed pass a function around without having to resort to "ancient" tricks such as method pointers.

In the code we just saw, we're passing a function as an argument to the sort() method. This is one aspect of higher-order functions. The function we're passing is actually the result of calling bind() on the original Person.compare() method, which means this bind() thing, which I'll explain shortly in a moment, actually *returns* a function. This is another aspect of the language's support for higher-order functions.

In this code, we would like to use our comparator function, except we need to pass it with the first argument (the criterion one) prefilled. Prototype's bind() method on functions lets us do this, among other things (and we'll discuss it in depth in Section 4.2, *Proper Function Binding*, on page 60).

Finally, the invoke() method lets us call a given method on each element in the series returned by sort() (possibly with arguments, although we don't need any here) and returns an array of the resulting values. JavaScript places no restrictions on where you can use the dot (.) operator; as long as its left side is an object, you're in the clear. If that side is a method call, all you need is that method call to return an object; this lets you chain calls easily to any length you may need.

Finally, on page creation, once it is loaded and the DOM is all ready, we want to dynamically inject a bulleted list of all the people we have by ascending natural order (since the default value for the criterion is the first name, we'll get first-name ordering).

Manually creating all the required DOM nodes would be fastidious, so we elect to build valid XHTML text and inject it safely into the proper container. Here's the code:

Download prototype/intro/basic/basic.js

```
document.observe('dom:loaded', function() {
  html = '<ul>\n'
    + people.sort(Person.compare).map(function(p) {
        return '\t<li>' + p.getDisplayName().escapeHTML() + '</li>';
      }).join('\n')
    + '</ul>';
  $('result').update(html);
  $$('#result li:nth-child(2n)').invoke('addClassName', 'alternate');
});
```

Look at the map() call on line 3; this is the all-purpose transformation method (pluck() and invoke() are special-purpose optimizations of it). We get an array of *<li>...</li>* text with our "display" names inside, then join them with line delimiters, and finally wrap the whole thing in a *<ul>...</ul>*. To guard us against weird characters in the people data, we use escapeHTML() on the resulting strings, effectively "defanging" any markup in there.

This is all just markup. To safely inject it into the DOM, we need to grab the element with id="result", which is gracefully done with $(). This method also makes sure the element we get back is equipped with the countless DOM extensions Prototype provides, including the mighty update() method, that we use to inject our markup into the element's DOM fragment.

Notice that our whole anonymous method is passed to document.observe(), which is part of Prototype's unified API to event handling (if you've ever played with events with your bare hands, you noticed, for instance, that Internet Explorer superbly ignores most of the official W3C specifications about it). Our method will be run when the document's DOM has finished loading, which is just what we need.

Finally, the killer call is on line 8. You know these fancy CSS 3 selectors we just can't use because they're not all that well supported yet? Well, we sure can use them with Prototype's $$()[1] to select any set of elements in the DOM! Then Prototype comes with CSS-tweaking methods, such as addClassName(), that take an extra CSS class name argument, but such methods are designed to work on the element we're calling them on. How can we use it on all the elements $$() returned? That's what invoke() is for, and using it lets us alter all matching elements concisely. The matching CSS is very short:

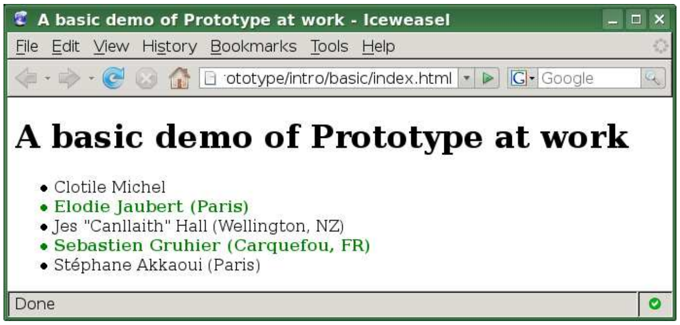
Download prototype/intro/basic/basic.css

```
#result li.alternate { font-weight: bold; color: green; }
```

Once loaded, our page looks like Figure 2.1, on the next page.

That's it for a first run. Excited? I hope so. Take some time to breathe. If you're on Firefox, why not bring up a Firebug[2] console and play with this script interactively? Or take a stroll. Go enjoy the company's free coffee. Check out the blogs.

1. Blazing fast since 1.5.1.
2. http://getfirebug.com


A basic demo of Prototype at work - Iceweasel
File Edit View History Bookmarks Tools Help
ototype/intro/basic/index.html
Google
Done

A basic demo of Prototype at work

- Clotile Michel
- Elodie Jaubert (Paris)
- Jes "Canllaith" Hall (Wellington, NZ)
- Sebastien Gruhier (Carquefou, FR)
- Stéphane Akkaoui (Paris)

Figure 2.1: Our dynamic, custom-styled content

Ready to move ahead with something more involved? Here we go.

One Good-Looking Script: A Table Sorter

To let you feel how using Prototype can lead to neat, cool JavaScript code, we'll build a simple table sorter. As long as our (X)HTML tables properly feature a <*thead*> and a <*tbody*>, our sorter object will be able to sort it.

The idea is to unobtrusively bind sorter objects to <*table*> elements so that the user can click the column heading and have the table sort accordingly. Clicking a second time on the current sort heading switches to a descending sort. We'll also use a few CSS class names so that styling can be applied to express the current sorting status.

Our table sorter system is "simpler" insofar as it does not deal with data types; it treats every cell as text. On the other hand, it does grab the cell's whole text, regardless of internal markup.

The full source code is available online. For this example, we'll focus on the neat parts, but there's very little we're leaving out anyway: support for CSS rules, status toggling for the sort, and extra <*table*> elements.

Laying the Groundwork

OK, so we need an HTML page with a couple tables on it (just to show the sorting capability is neatly wrapped into an object and we can reuse it multiple times on the same page).