Prototype 1.5

The Complete API Reference

Sam Stephenson and the Prototype Team

Prototype 1.5: The Complete API Reference

Sam Stephenson and the Prototype Team

Published March 2007. 2nd edition.

Prototype[1] is a JavaScript framework that aims to ease development of dynamic web applications. Prototype was created by Sam Stephenson who released the framework as an open-source project in February 2005. Other members of the core development team are: Thomas Fuchs, Justin Palmer, Andrew Dupont, Dan Webb, Scott Raymond, Seth Dillingham, Mislav Marohni#, Christophe Porteneuve and Tobie Langel.

This PDF book version of the Prototype API reference was created by Josh Clark[2], but all content comes from the Prototype site and is the intellectual property of Sam Stephenson and the Prototype core team.

[1] http://www.prototypejs.org/
[2] http://www.globalmoxie.com/
[3] http://creativecommons.org/licenses/by-sa/2.5/

Table of Contents

Chapter

1

Utility Methods

Prototype provides a number of "convenience" methods. Most are aliases of other Prototype methods, with the exception of the `$` method, which wraps DOM nodes with additional functionality.

These utility methods all address scripting needs that are **so common** that their names were made as concise as can be. Hence the `$`-based convention.

The most commonly used utility method is without doubt `$()`, which is, for instance, used pervasively within Prototype's code to let you pass either element IDs or actual DOM element references just about anywhere an element argument is possible. It actually goes **way beyond** a simple wrapper around `document.getElementById`; check it out to see just how useful it is.

These methods are one of the cornerstones of efficient Prototype-based JavaScript coding. Take the time to learn them well.

$

```
$(id | element) -> HTMLElement
$((id | element)...) -> [HTMLElement...]
```

If provided with a string, returns the element in the document with matching ID; otherwise returns the passed element. Takes in an arbitrary number of arguments. All elements returned by the function are extended with Prototype DOM extensions.

The `$` function is the cornerstone of Prototype, its Swiss Army knife. Not only does it provide a handy alias for `document.getElementById`, it also lets you pass indifferently IDs (strings) or DOM node references to your functions:

```
function foo(element) {
    element = $(element);
    /*  rest of the function... */
}
```

Example 1.1.

Code written this way is flexible — you can pass it the ID of the element or the element itself without any type sniffing.

Invoking it with only one argument returns the element, while invoking it with multiple arguments returns an array of elements (and this works recursively: if you're twisted, you could pass it an array containing some arrays, and so forth). As this is dependent on `getElementById`, W3C specs[1] apply: nonexistent IDs will yield `null` and IDs present multiple times in the DOM will yield erratic results. **If you're assigning the same ID to multiple elements, you're doing it wrong!**

The function also **extends every returned element** with `Element.extend` so you can use Prototype's DOM extensions on it. In the following code, the two lines are equivalent. However, the second one feels significantly more object-oriented:

```
// Note quite OOP-like...
Element.hide('itemId');
// A cleaner feel, thanks to guaranted extension
$('itemId').hide();
```

Example 1.2.

However, when using iterators, leveraging the `$` function makes for more elegant, more concise, and also more efficient code:

```
['item1', 'item2', 'item3'].each(Element.hide);
// The better way:
$('item1', 'item2', 'item3').invoke('hide');
```

Example 1.3.

See How Prototype extends the DOM[2] for more info.

[1] http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-getElBId
[2] http://http://www.prototypejs.org/learn/extensions

$$

```
$$(cssRule...) -> [HTMLElement...]
```

Takes an arbitrary number of CSS selectors (strings) and returns a document-order array of extended DOM elements that match any of them.

Sometimes the usual tools from your DOM arsenal -- `document.getElementById()` encapsulated by `$()`, `getElementsByTagName()` and even Prototype's very own `getElementsByClassName()` extensions -- just aren't enough to quickly find our elements or collections of elements. If you know the DOM tree structure, you can simply resort to CSS selectors to get the job done.

Performance: when better alternatives should be used instead of `$$`

Now, this function is powerful, but if misused, it will suffer performance issues. So here are a few guidelines:

- **If you're just looking for elements with a specific CSS class** name among their class name set, use Prototype's `document.getElementsByClassName()` extension.
- Better yet: if this search is **constrained within a given container** element, use the `Element.getElementsByClassName()` extension.

Those methods should be favored in the case of simple CSS-class-based lookups, because they're, at any rate, faster than `$$`. On browsers supporting DOM Level 3 XPath[3], they're potentially *blazing* fast.

Now, if you're going for more complex stuff, indeed use `$$`. But use it well. The `$$` function searches, by default, the whole document. So if you can, scope your CSS rules as early as you can, e.g. by having them start with ID selectors. That'll help reduce the search tree as fast as possible, and speed up your operation.

```
$$('div')
// -> all DIVs in the document.  Same as document.getElementsByTagName('div')!

$$('#contents')
// -> same as $('contents'), only it returns an array anyway.

$$('li.faux')
// -> all LI elements with class 'faux'

$$('#contents a[rel]')
// -> all links inside the element of ID "contents" with a rel attribute

$$('a[href="#"]')
// -> all links with a href attribute of value "#" (eyeew!)

$$('#navbar li', '#sidebar li')
// -> all links within the elements of ID "navbar" or "sidebar"
```

Example 1.4.

[3] http://www.w3.org/TR/DOM-Level-3-XPath/xpath.html

Supported CSS syntax

The `$$` function does not rely on the browser's internal CSS parsing capabilities (otherwise, we'd be in cross-browser trouble...), and therefore offers a consistent set of selectors across all supported browsers. The flip side is, it currently doesn't support as many selectors as browsers that are very CSS-capable. Here is the current set of supported selectors:

- Type selector: tag names, as in `div`.
- Descendant selector: the space(s) between other selectors, as in `#a li`.
- Attribute selectors: the full CSS 2.1 set of `[attr]`, `[attr=value]`, `[attr~=value]` and `[attr|=value]`. It also supports `[attr!=value]`. If the value you're matching against includes a space, be sure to enclose the value in quotation marks (`[title="Hello World!"]`).
- Class selector: CSS class names, as in `.highlighted` or `.example.wrong`.
- ID selector: as in `#item1`.

However, it currently **does not** support child selectors (`>`), adjacent sibling selectors (`+`), pseudo-elements (e.g. `:after`) and pseudo-classes (e.g. `:hover`).

$A

```
$A(iterable) -> actualArray
```

Accepts an array-like collection (anything with numeric indices) and returns its equivalent as an actual `Array` object. This method is a convenience alias of `Array.from`, but is the preferred way of casting to an `Array`.

The primary use of `$A()` is to obtain an actual `Array` object based on anything that could pass as an array (e.g. the `NodeList` or `HTMLCollection` objects returned by numerous DOM methods, or the predefined `arguments` reference within your functions).

The reason you would want an actual `Array` is simple: Prototype extends `Array` to equip it with numerous extra methods, and also mixes in the `Enumerable` module, which brings in another boatload of nifty methods. Therefore, in Prototype, actual `Array`s trump any other collection type you might otherwise get.

The conversion performed is rather simple: `null`, `undefined` and `false` become an empty array; any object featuring an explicit `toArray` method (as many Prototype objects do) has it invoked; otherwise, we assume the argument "looks like an array" (e.g. features a `length` property and the `[]` operator), and iterate over its components in the usual way.

The well-known DOM method `document.getElementsByTagName()`[4] doesn't return an `Array`, but a `NodeList` object that implements the basic array "interface." Internet Explorer does not allow us to extend `Enumerable` onto `NodeList.prototype`, so instead we cast the returned `NodeList` to an `Array`:

[4] http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-A6C9094

```
var paras = $A(document.getElementsByTagName('p'));
paras.each(Element.hide);
$(paras.last()).show();
```

Example 1.5.

Notice we had to use `each` and `Element.hide` because `$A` doesn't perform DOM extensions, since the array could contain anything (not just DOM elements). To use the `hide` instance method we first must make sure all the target elements are extended:

```
$A(document.getElementsByTagName('p')).map(Element.extend).invoke('hide');
```

Example 1.6.

Want to display your arguments easily? `Array` features a `join` method, but the `arguments` value that exists in all functions *does not* inherit from `Array`. So, the tough way, or the easy way?

```
// The hard way...
function showArgs() {
  alert(Array.prototype.join.call(arguments, ', '));
}

// The easy way...
function showArgs() {
  alert($A(arguments).join(', '));
}
```

Example 1.7.

$F

```
$F(element) -> value
```

Returns the value of a form control. This is a convenience alias of `Form.Element.getValue`. Refer to it for full details.

$H

```
$H([obj]) -> Hash
```

Creates a `Hash` (which is synonymous to "map" or "associative array" for our purposes). A convenience wrapper around the `Hash` constructor, with a safeguard that lets you pass an existing `Hash` object and get it

back untouched (instead of uselessly cloning it).

The `$H` function is the shorter way to obtain a hash (prior to 1.5 final, it was the *only* proper way of getting one).

$R

```
$R(start, end[, exclusive = false]) -> ObjectRange
```

Creates a new `ObjectRange` object. This method is a convenience wrapper around the `ObjectRange` constructor, but `$R` is the preferred alias.

`ObjectRange` instances represent a range of consecutive values, be they numerical, textual, or of another type that semantically supports value ranges. See the type's documentation for further details, and to discover how your own objects can support value ranges.

The `$R` function takes exactly the same arguments as the original constructor: the **lower and upper bounds** (value of the same, proper type), and **whether the upper bound is exclusive** or not. By default, the upper bound is inclusive.

```
$R(0, 10).include(10)
// -> true

$A($R(0, 5)).join(', ')
// -> '0, 1, 2, 3, 4, 5'

$A($R('aa', 'ah')).join(', ')
// -> 'aa, ab, ac, ad, ae, af, ag, ah'

$R(0, 10, true).include(10)
// -> false

$R(0, 10, true).each(function(value) {
  // invoked 10 times for value = 0 to 9
});
```

Example 1.8.

Note that `ObjectRange` mixes in the `Enumerable` module: this makes it easy to convert a range to an `Array` (`Enumerable` provides the `toArray` method, which makes the `$A` conversion straightforward), or to iterate through values. (Note, however, that getting the bounds back will be more efficiently done using the `start` and `end` properties than calling the `min()` and `max()` methods).

$w

```
$w(String) -> Array
```

Splits a string into an `Array`, treating all whitespace as delimiters. Equivalent to Ruby's `%w{foo bar}` or Perl's `qw(foo bar)`.

This is one of those life-savers for people who just hate commas in literal arrays :-)

```
$w('apples bananas kiwis')
// -> ['apples', 'bananas', 'kiwis']
```

Example 1.9.

This can slightly shorten code when writing simple iterations:

```
$w('apples bananas kiwis').each(function(fruit){
  var message = 'I like ' + fruit
  // do something with the message
})
```

Example 1.10.

This also becomes sweet when combined with `Element` functions:

```
$w('ads navbar funkyLinks').each(Element.hide);
```

Example 1.11.

Try.these

```
Try.these(Function...) -> firstOKResult
```

Accepts an arbitrary number of functions and returns the result of the first one that doesn't throw an error.

This method provides a simple idiom for trying out blocks of code in sequence. Such a sequence of attempts usually represents a downgrading approach to obtaining a given feature.

In this example from Prototype's `Ajax` library, we want to get an `XMLHttpRequest` object. Internet Explorer 6 and earlier, however, does not provide it as a vanilla JavaScript object, and will throw an error if we attempt a simple instantiation. Also, over time, its proprietary way evolved, changing COM interface names.

`Try.these` will try several ways in sequence, from the best (and, theoretically, most widespread) one to the oldest and rarest way, returning the result of the first successful function.

If none of the blocks succeeded, `Try.these` will return `undefined`, which will cause the `getTransport` method in the example below to return `false`, provided as a fallback result value.

```
getTransport: function() {
  return Try.these(
    function() { return new XMLHttpRequest() },
    function() { return new ActiveXObject('Msxml2.XMLHTTP') },
    function() { return new ActiveXObject('Microsoft.XMLHTTP') }
  ) || false;
}
```

Example 1.12.

document.getElementsByClassName

```
document.getElementsByClassName(className[, element]) -> [HTMLElement...]
```

Retrieves (and extends) all the elements that have a CSS class name of `className`. The optional `element` parameter specifies a parent element to search under.

Note that each returned element has been extended.

```
<body>
    <div id="one" class="foo">Single class name</div>
    <div id="two" class="foo bar thud">Multiple class names</div>
    <ul id="list">
      <li id="item_one" class="thud">List item 1</li>
      <li>List item 2</li>
      <li id="item_two" class="thud">List item 3</li>
    </ul>
  </body>
```

```
document.getElementsByClassName('foo');
  // -> [HTMLElement, HTMLElement] (div#one, div#two)

  document.getElementsByClassName('thud');
  // -> [HTMLElement, HTMLElement, HTMLElement] (div#two, li#item_one, li#item_two);

  document.getElementsByClassName('thud', $('list'));
  // -> [HTMLElement, HTMLElement] (li#item_one, li#item_two)
```

Example 1.13.

Chapter 2

Ajax

Prototype offers three objects to deal with AJAX communication, which are listed below. With Prototype, going Ajaxy is downright simple! All three objects share a common set of options, which are discussed separately.

The articles below provide you with several examples. The Learn section also features a more narrative, tutorial-style article[1].

Ajax Options

This details all core options (shared by all AJAX requesters) and callbacks.

All requester objects in the `Ajax` namespace share a common set of **options** and **callbacks**. Callbacks are called at various points in the life-cycle of a request, and always feature the same list of arguments. They are passed to requesters right along with their other options.

Common options

Option	Default	Description
`asynchronous`	`true`	Determines whether `XMLHttpRequest` is used asynchronously or not. Since synchronous usage is rather unsettling, and usually bad taste, you should avoid changing this. Seriously.
`contentType`	`'application/ x-www-form-urlencoded'`	The `Content-Type` header for your request. You might want to send XML in-

[1] http://http://www.prototypejs.org/learn/introduction-to-ajax

Option	Default	Description
		stead of the regular URL-encoded format, in which case you would have to change this.
`encoding`	`'UTF-8'`	The encoding for your request contents. It is best left as is, but should weird encoding issues arise, you may have to tweak it in accordance with other encoding-related parts of your page code and server side.
`method`	`'post'`	The HTTP method to use for the request. The other widespread possibility is `'get'`. As a Ruby On Rails special, Prototype also reacts to other verbs (such as `'put'` and `'delete'` by actually using `'post'` and putting an extra `'_method'` parameter with the originally requested method in there.
`parameters`	`''`	The parameters for the request, which will be encoded into the URL for a `'get'` method, or into the request body for the other methods. This can be provided either as a URL-encoded string or as any `Hash`-compatible object (basically anything), with properties representing parameters.
`postBody`	None	Specific contents for the request body on a `'post'` method (actual method, after possible conversion as described in the `method` opt ion above). If it is not provided, the contents of the `parameters` option will be used instead.
`requestHeaders`	See text	Request headers can be passed under two forms: • As an **object**, with properties representing headers. • As an **array**, with even-index (0, 2…) elements being header names, and odd-index (1, 3…) elements being values.

Option	Default	Description
		Prototype automatically provides a set of default headers, that this option can override and augment: • `X-Requested-With` is set to `'XMLHttpRequest'`. • `X-Prototype-Version` provides Prototype's current version (e.g. 1.5.0). • `Accept` defaults to `'text/javascript, text/html, application/xml, text/xml, */*'` • `Content-type` is built based on the `contentType` and `encoding` options.

Common callbacks

When used on individual instances, all callbacks (except `onException`) are invoked with two parameters: the `XMLHttpRequest` object and the result of evaluating the `X-JSON` response header, if any (can be `null`).

For another way of describing their chronological order and which callbacks are mutually exclusive, see `Ajax.Request`.

Callback	Description
`onComplete`	Triggered at the very end of a request's life-cycle, once the request completed, status-specific callbacks were called, and possible automatic behaviors were processed.
`onException`	Triggered whenever an XHR error arises. Has a custom signature: the first argument is the requester (i.e. an `Ajax.Request` instance), the second is the exception object.
`onFailure`	Invoked when a request completes and its status code exists but is not in the 2xy family. This is skipped if a code-specific callback is defined, and happens *before* `onComplete`.
`onInteractive`	(Not guaranteed) Triggered whenever the requester receives a part of the response (but not the final part), should it be sent in several packets.
`onLoaded`	(Not guaranteed) Triggered once the underlying XHR object is setup, the connection open, and ready to send its actual request.
`onLoading`	(Not guaranteed) Triggered when the underlying XHR object is being setup, and its

Callback	Description
	connection opened.
`onSuccess`	Invoked when a request completes and its status code is undefined or belongs in the 2xy family. This is skipped if a code-specific callback is defined, and happens *before* `onComplete`.
`onUninitialized`	(Not guaranteed) Invoked when the XHR object was just created.
`on`**XYZ**	With *XYZ* being an HTTP status code for the response. Invoked when the response just completed, and the status code is exactly the one we used in t he callback name. Prevents execution of `onSuccess` / `onFailure`. Happens *before* `onComplete`.

Responder callbacks

When used on responders, all callbacks (except `onException` and `onCreate`) are invoked with three parameters: the requester (i.e. the corresponding "instance" of `Ajax.Request`) object, the `XMLHttpRequest` object and the result of evaluating the `X-JSON` response header, if any (can be `null`). They also execute in the context of the responder, bound to the `this` reference.

Callback	Description
`onCreate`	Triggered whenever a requester object from the `Ajax` namespace is created, after its parameters where adjusted and its before its XHR connection is opened. This takes two arguments: the requester object and the underlying XHR object.
`onComplete`	Triggered at the very end of a request's life-cycle, once the request completed, status-specific callbacks were called, and possible automatic behaviors were processed.
`onException`	Triggered whenever an XHR error arises. Has a custom signature: the first argument is the requester (i.e. an `Ajax.Request` instance), the second is the exception object.
`onInteractive`	(Not guaranteed) Triggered whenever the requester receives a part of the response (but not the final part), should it be sent in several packets.
`onLoaded`	(Not guaranteed) Triggered once the underlying XHR object is setup, the connection open, and ready to send its actual request.
`onLoading`	(Not guaranteed) Triggered when the underlying XHR object is being setup, and its connection opened.
`onUninitialized`	(Not guaranteed) Invoked when the XHR object was just created.

Ajax.PeriodicalUpdater

```
new Ajax.PeriodicalUpdater(container, url[, options])
```

Periodically performs an AJAX request and updates a container's contents based on the response text. Offers a mechanism for "decay," which lets it trigger at widening intervals while the response is unchanged.

This object addresses the common need of periodical update, which is used by all sorts of "polling" mechanisms (e.g. in an online chatroom or an online mail client).

The basic idea is to run a regular `Ajax.Updater` at regular intervals, monitoring changes in the response text if the `decay` option (see below) is active.

Additional options

`Ajax.PeriodicalUpdater` features all the common options and callbacks, plus those added by `Ajax.Updater`. It also provides two new options that deal with the original period, and its decay rate (how Rocket Scientist does that make us sound, uh?!).

Option	Default	Description
`frequency`	`2`	Okay, this is not a frequency (e.g 0.5Hz), but a period (i.e. a number of seconds). Don't kill me, I didn't write this one! This is the minimum interval at which AJAX requests are made. You don't want to make it too short (otherwise you may very well end up with multiple requests in parallel, if they take longer to process and return), but you technically can provide a number below one, e.g. 0.75 second.
`decay`	1	This controls the rate at which the request interval grows when the response is unchanged. It is used as a multiplier on the current period (which starts at the original value of the `frequency` parameter). Every time a request returns an unchanged response text, the current period is multiplied by the decay. Therefore, the default value means regular requests (no change of interval). Values higher than one will yield growing intervals. Values below one are dangerous: the longer the response text stays the same, the more often you'll check, until the interval is so short your browser is left with no other choice than suicide. Note that, as soon as the response text *does* change, the current period resets to the original one.

To better understand decay, here is a small sequence of calls from the following example:

```
new Ajax.PeriodicalUpdater('items', '/items', {
  method: 'get', frequency: 3, decay: 2
});
```

Example 2.1.

Call#	When?	Decay before	Response changed?	Decay after	Next period	Comments
1	00:00	2	n/a	1	3	Response is deemed changed, since there is no prior response to compare to!
2	00:03	1	yes	1	3	Response did change again: we "reset" to 1, which was already the decay.
3	00:06	1	no	2	6	Response didn't change: decay augments by the `decay` option factor: we're waiting longer now…
4	00:12	2	no	4	12	Still no change, doubling again.
5	00:24	4	no	8	24	Jesus, is this thing going to change or what?

Call#	When?	Decay before	Response changed?	Decay after	Next period	Comments
6	00:48	8	yes	1	3	Ah, finally! Resetting decay to 1, and therefore using the original period.

Disabling and re-enabling a `PeriodicalUpdater`

You can pull the brake on a running `PeriodicalUpdater` by simply calling its `stop` method. If you wish to re-enable it later, just call its `start` method. Both take no argument.

Beware! Not a specialization!

`Ajax.PeriodicalUpdater` is not a specialization of `Ajax.Updater`, despite its name. When using it, do not expect to be able to use methods normally provided by `Ajax.Request` and "inherited" by `Ajax.Updater`, such as `evalJSON` or `getHeader`. Also the `onComplete` callback is hijacked to be used for update management, so if you wish to be notified of every successful request, use `onSuccess` instead (beware: it will get called *before* the update is performed).

Ajax.Request

```
new Ajax.Request(url[, options])
```

Initiates and processes an AJAX request.

This object is a general-purpose AJAX requester: it handles the life-cycle of the request, handles the boilerplate, and lets you plug in callback functions for your custom needs.

In the optional options hash, you usually provide a `onComplete` and/or `onSuccess` callback, unless you're in the edge case where you're getting a JavaScript-typed response, that will automatically be `eval`'d.

For a full list of common options and callbacks, see Ajax Options.

The only proper way to create a requester is through the `new` operator. As soon as the object is created, it initiates the request, then goes on processing it throughout its life-cyle.

A basic example

```
var url = '/proxy?url=' + encodeURIComponent('http://www.google.com/search?q=Prototype');
// notice the use of a proxy to circumvent the Same Origin Policy.

new Ajax.Request(url, {
  method: 'get',
  onSuccess: function(transport) {
    var notice = $('notice');
    if (transport.responseText.match(/href="http:\/\/prototypejs.org/))
      notice.update('Yeah! You are in the Top 10!').setStyle({ background: '#dfd' });
    else
      notice.update('Damn! You are beyond #10...').setStyle({ background: '#fdd' });
  }
});
```

Example 2.2.

Request life-cycle

Underneath our nice requester objects lies, of course, `XMLHttpRequest`. The defined life-cycle is as follows:

1. Created
2. Initialized
3. Request sent
4. Response being received (can occur many times, as packets come in)
5. Response received, request complete

As you can see in Ajax options, Prototype's AJAX objects define a whole slew of callbacks, which are triggered in the following order:

1. `onCreate` (this is actually a callback reserved to AJAX global responders)
2. `onUninitialized` (maps on Created)
3. `onLoading` (maps on Initialized)
4. `onLoaded` (maps on Request sent)
5. `onInteractive` (maps on Response being received)
6. on*XYZ* (numerical response status code), onSuccess or onFailure (see below)
7. `onComplete`

The two last steps both map on *Response received*, in that order. If a status-specific callback is defined, it gets invoked. Otherwise, if `onSuccess` is defined and the response is deemed a success (see below), it is invoked. Otherwise, if `onFailure` is defined and the response is *not* deemed a sucess, it is invoked. Only after that potential first callback is `onComplete` called.

A note on portability

Depending on how your browser implements `XMLHttpRequest`, one or more callbacks may never be invoked. In particular, `onLoaded` and `onInteractive` are not a 100% safe bet so far. However, the global `onCreate`, `onUninitialized` and the two final steps are very much guaranteed.

`onSuccess` and `onFailure`, the under-used callbacks

Way too many people use `Ajax.Request` in a similar manner to raw XHR, defining only an `onComplete` callback even when they're only interested in "successful" responses, thereby testing it by hand:

```
// This is too bad, there's better!
new Ajax.Request('/your/url', {
  onComplete: function(transport) {
    if (200 == transport.status)
      // yada yada yada
  }
});
```

Example 2.3.

First, as described below, you could use better "success" detection: success is generally defined, HTTP-wise, as either no response status or a "2xy" response status (e.g., 201 is a success, too). See the example below.

Second, you could dispense with status testing altogether! Prototype adds callbacks specific to success and failure, which we listed above. Here's what you could do if you're only interested in success, for instance:

```
new Ajax.Request('/your/url', {
  onSuccess: function(transport) {
      // yada yada yada
  }
});
```

Example 2.4.

Automatic JavaScript response evaluation

Any response whose MIME type is missing or JavaScript-related will automatically be passed to `eval`. Before yelling on what a security breach that is, remember that XHR is usually used on URLs from the same host that originated the current page (this is the famous *Same Origin Policy*[2], or SOP): these scripts are supposed to be under your control.

[2] http://en.wikipedia.org/wiki/Same_origin_policy

What this means is, you don't even need to provide a callback to leverage pure-JavaScript AJAX responses. That's pretty cool, wouldn't you say? The list of JavaScript-related MIME types handled by Prototype is:

- `application/ecmascript`
- `application/javascript`
- `application/x-ecmascript`
- `application/x-javascript`
- `text/ecmascript`
- `text/javascript`
- `text/x-ecmascript`
- `text/x-javascript`

The MIME type string is examined in a case-insensitive manner.

Methods you may find useful

Instances of the Request object provide several methods that can come in handy in your callback functions, especially once the request completed.

Is the response a successful one?

The `success()` method examines the XHR's `status` property, and follows general HTTP guidelines: unknown status is deemed successful, as is the whole 2xy status code family. It's a generally better way of testing your response than the usual `200 == transport.status`.

Getting HTTP response headers

While you can obtain response headers from the XHR object, using its `getResponseHeader` method, this makes for slightly verbose code, and several implementations may raise an exception when the header is not found. To make this easier, you can use the `getHeader` method, which just delegates to the longer version and returns `null` should an exception occur:

```
var myRequest = new Ajax.Request('/your/url', {
  onSuccess: function() {
    // Note how we brace against null values
    if ((myRequest.getHeader('Server') || '').match(/Apache/))
      ++gApacheCount;
    // Remainder of the code
  }
});
```

Example 2.5.

Evaluating JSON headers

Some backends will return JSON not as response text, but in the `X-JSON` header. In which case, you don't even need to evaluate the returned JSON yourself, as Prototype automatically does so and passes it as the final argument of each callback (except, obviously, `onCreate`). Note that if there is no such header or its contents is invalid, this argument will be set to `null`.

```
new Ajax.Request('/your/url', {
  onSuccess: function(transport, json) {
    // Remainder of the code
  }
});
```

Example 2.6.

Ajax.Responders

```
Ajax.Responders.register(responder)
Ajax.Responders.unregister(responder)
```

A repository of global listeners notified about every step of Prototype-based AJAX requests.

Sometimes, you need to provide generic behaviors over all AJAX operations happening in the page (through `Ajax.Request`, `Ajax.Updater` or `Ajax.PeriodicalUpdater`). For instance, you might want to automatically show an indicator when an AJAX request is ongoing, and hide it when none are. You may well want to factor out exception handling as well, logging those somewhere on the page in a custom fashion. The possibilities are plenty.

To achieve this, Prototype provides `Ajax.Responders`, which lets you register (and if you wish to, unregister later) **responders**, which are objects with properly-named methods. These names are the regular callback names, and your responders can implement any set of interest.

For instance, Prototype automatically registers a responder that maintains a nifty variable: `Ajax.activeRequestCount`. This contains, at any time, the amount of currently active AJAX requests (those created by Prototype, anyway), by monitoring their `onCreate` and `onComplete` events. The code for this is fairly simple:

```
Ajax.Responders.register({
  onCreate: function() {
    Ajax.activeRequestCount++;
  },
  onComplete: function() {
    Ajax.activeRequestCount--;
  }
});
```

Example 2.7.

All callbacks in the life-cycle are available; actually, `onCreate` is only available to responders, as it wouldn't make a lot of sense to individual requests: you do know when your code creates them, don't you? It is triggered even before the XHR connection is opened, which makes it happen right before `onUninitialized`.

Unregister: remember the reference…

As always, unregistering something requires you to use the very same object you used at registration. So if you plan on unregistering a responder, be sure to define it first, then pass the reference to `register`, and finally, when the time comes, to `unregister`.

Ajax.Updater

```
new Ajax.Updater(container, url[, options])
```

Performs an AJAX request and updates a container's contents based on the response text.

`Ajax.Updater` is a specialization of `Ajax.Request`: everything about the latter is true for the former. If you're unfamiliar with `Ajax.Request`, go read its documentation before going ahead with this one.

A simple example

```
new Ajax.Updater('items', '/items', {
  parameters: { text: $F('text') }
});
```

Example 2.8.

A note about timing

The `onComplete` callback will get invoked **after** the update takes place.

Additional options

Since the goal of `Ajax.Updater` is specifically to update the contents of a DOM element (`container`) with the response text brought back by the AJAX request, it features a couple of new options, in addition to the common options set. These are:

Option	Default	Description
`evalScripts`	`false`	This determines whether `<script>` elements in the response text are evaluated or not.
`insertion`	None	By default, `Element.update` is used, which replaces the whole contents of the container with the response text. You may want to instead insert the response text around existing contents. You just need to pass a valid `Insertion` object for this, such as `Insertion.Bottom`.

In the following example, we assume that creating a new item through AJAX returns an XHTML fragment representing only the new item, which we need to add within our list container, but at the bottom of its existing contents. Here it goes:

```
new Ajax.Updater('items', '/items', {
  parameters: { text: $F('text') },
  insertion: Insertion.Bottom
});
```

Example 2.9.

About `evalScripts` and defining functions

If you use `evalScripts: true`, any `<script>` block will be evaluated. This **does not** mean it will get included in the page: they won't. Their content will simply be passed to the native `eval()` function. There are two consequences to this:

- The local scope will be that of Prototype's internal processing function. Anything in your script declared with `var` will be discarded momentarily after evaluation, and at any rate will be invisible to the remainder of the page scripts.
- If you define functions in there, you need to actually **create** them, otherwise they won't be accessible to the remainder of the page scripts. That is, the following code won't work:

```
// This kind of script won't work if processed by Ajax.Updater:
function coolFunc() {
  // Amazing stuff!
}
```

Example 2.10.

You will need to use the following syntax:

```
// This kind of script WILL work if processed by Ajax.Updater:
coolFunc = function() {
  // Amazing stuff!
}
```

Example 2.11.

That's a common trickster, biting beginners in the ankle. So watch out!

Single container, or success/failure alternative?

The examples above all assume you're going to update the same container whether your request succeeds or fails. There may very well be times when you don't want that. You may want to update only for successful requests, or update a different container on failed requests.

To achieve this, you can pass an object instead of a DOM element for the `container` parameter. This object **must** exhibit a `success` property, whose value is the container to be updated upon successful requests. If you also provide it with a `failure` property, its value will be used as the container for failed requests.

In the following code, only successful requests get an update:

```
new Ajax.Updater({ success: 'items' }, '/items', {
  parameters: { text: $F('text') },
  insertion: Insertion.Bottom
});
```

Example 2.12.

The next example assumes failed requests will feature an error message as response text, and will go on to update another element with it, probably a status zone.

```
new Ajax.Updater({ success: 'items', failure: 'notice' }, '/items', {
  parameters: { text: $F('text') },
  insertion: Insertion.Bottom
});
```

Example 2.13.

Chapter 3

Array

Prototype extends all native Javascript arrays with quite a few powerful methods.

This is done in two ways:

- It mixes in the `Enumerable` module, which brings a ton of methods in already.
- It adds quite a few extra methods, which are documented in this section.

With Prototype, arrays become much, much more than the trivial objects we were used to manipulate, limiting ourselves to using their `length` property and their `[]` indexing operator. They become very powerful objects, that greatly simplify the code for 99% of the common use cases involving them.

Why you should stop using `for...in` to iterate (or never take it up)

Many JavaScript authors have been misled into using the `for...in` JavaScript construct to loop over array elements. This kind of code just won't work with Prototype.

You see, the ECMA 262[1] standard, which defines ECMAScript 3rd edition, supposedly implemented by all major browsers *including MSIE*, defines numerous methods on `Array` (§15.4.4), including such nice methods as `concat`, `join`, `pop` and `push`, to name but a few among the ten methods specified.

This same standard explicitely defines that the `for...in` construct (§12.6.4) exists to **enumerate the properties** of the object appearing on the right side of the `in` keyword. Only properties specifically marked as non-enumerable are ignored by such a loop. By default, the prototype and the `length` properties are so marked, which prevents you from enumerating over array methods when using `for...in`. This comfort led developers to use `for...in` as a shortcut for indexing loops, when it is not its actual purpose.

[1] http://www.ecma-international.org/publications/standards/Ecma-262.htm

However, Prototype has no way to mark the methods it adds to `Array.prototype` as non-enumerable. Therefore, using `for...in` on arrays when using Prototype will enumerate all extended methods as well, such as those coming from the `Enumerable` module, and those Prototype puts in the `Array` namespace (described in this section, and listed further below).

What is a developer to do?

You can revert to vanilla loops:

```
for (var index = 0; index < myArray.length; ++index) {
  var item = myArray[index];
  // Your code working on item here...
}
```

Or you can use **iterators**, such as `each` :

```
myArray.each(function(item) {
  // Your code working on item here...
});
```

This side-effect enforcement of the true purpose of `for...in` is actually not so much of a burden: as you'll see, most of what you used to loop over arrays for can be concisely done using the new methods provided by `Array` or the mixed-in `Enumerable` module. So manual loops should be fairly rare.

A note on performance

Should you have a very large array, using iterators with *lexical closures* (anonymous functions that you pass to the iterators, that get invoked at every loop iteration), such as `each`, or relying on repetitive array construction (such as `uniq`), may yield unsatisfactory performance. In such cases, you're better off writing manual indexing loops, but take care then to cache the `length` property and use the prefix ++ operator:

```
// Custom loop with cached length property:
// maximum full-loop performance on very large arrays!
for (var index = 0, len = myArray.length; index < len; ++index) {
  var item = myArray[index];
  // Your code working on item here...
}
```

Example 3.1.

clear

```
clear() -> Array
```

Clears the array (makes it empty).

```
var guys = ['Sam', 'Justin', 'Andrew', 'Dan'];
guys.clear();
// -> []
guys
// -> []
```

Example 3.2.

clone

```
clone() -> newArray
```

Returns a duplicate of the array, leaving the original array intact.

```
var fruits = ['Apples', 'Oranges'];
  var myFavs = fruits.clone();
  myFavs.pop();
  // fruits -> ['Apples', 'Oranges']
  // myFavs -> ['Apples']
```

Example 3.3.

compact

```
compact() -> newArray
```

Returns a new version of the array, without any `null`/`undefined` values.

```
['frank', , 'sue', , 'sally', null].compact()
// -> ['frank', 'sue', 'sally']
```

Example 3.4.

each

```
each(iterator) -> Array
```

Iterates over the array in ascending numerical index order.

This is actually the `each` method from the mixed-in `Enumerable` module. It is documented here to clearly state the order of iteration.

first

```
first() -> value
```

Returns the first item in the array, or `undefined` if the array is empty.

```
['Ruby', 'Php', 'Python'].first()
// -> 'Ruby'

[].first()
// -> undefined
```

Example 3.5.

flatten

```
flatten() -> newArray
```

Returns a "flat" (one-dimensional) version of the array. Nested arrays are recursively injected "inline." This can prove very useful when handling the results of a recursive collection algorithm, for instance.

```
['frank', ['bob', 'lisa'], ['jill', ['tom', 'sally']]].flatten()
// -> ['frank', 'bob', 'lisa', 'jill', 'tom', 'sally']
```

Example 3.6.

from

```
Array.from(iterable) -> actualArray
```

Clones an existing array or creates a new one from an array-like collection.

This is an alias for the $A() method. Refer to its page for complete description and examples.

indexOf

```
indexOf(value) -> position
```

Returns the position of the first occurrence of the argument within the array. If the argument doesn't exist in the array, returns -1.

Note: this uses the == equivalence operator, not the === strict equality operator. The bottom example below illustrates this.

Minor note: this uses a plain old optimized indexing loop, so there's no risk of extensions being detected by this method.

```
[3, 5, 6, 1, 20].indexOf(1)
// -> 3

[3, 5, 6, 1, 20].indexOf(90)
// -> -1

[0, false, 15].indexOf(false)
// -> 0 instead of 1, because 0 == false!
```

Example 3.7.

inspect

```
inspect() -> String
```

Returns the debug-oriented string representation of an array. For more information on `inspect` methods, see `Object.inspect`.

```
['Apples', {good: 'yes', bad: 'no'}, 3, 34].inspect()
// -> "['Apples', [object Object], 3, 34]"
```

Example 3.8.

Note

If you want to simply join the string elements of an array, use the native `join` method instead:

```
['apples', 'bananas', 'kiwis'].join(', ')
// -> 'apples, bananas, kiwis'
```

last

```
last() -> value
```

Returns the last item in the array, or `undefined` if the array is empty.

```
['Ruby', 'Php', 'Python'].last()
// -> 'Python'

[].last()
// -> undefined
```

Example 3.9.

reduce

```
reduce() -> Array | singleValue
```

Reduces arrays: one-element arrays are turned into their unique element, while multiple-element arrays are returned untouched.

```
[3].reduce();    // -> 3

[3, 5].reduce(); // -> [3, 5]
```

Example 3.10.

reverse

```
reverse([inline = true]) -> Array
```

Returns the reversed version of the array. By default, directly reverses the original. If `inline` is set to `false`, uses a clone of the original array.

```
var nums = [3, 5, 6, 1, 20];
nums.reverse(false)  // -> [20, 1, 6, 5, 3]
nums                 // -> [3, 5, 6, 1, 20]

nums.reverse()       // -> [20, 1, 6, 5, 3]
nums                 // -> [20, 1, 6, 5, 3]
```

Example 3.11.

size

```
size() -> Number
```

Returns the size of the array.

This is just a local optimization of the mixed-in `size` method from the `Enumerable` module, which avoids array cloning and uses the array's native `length` property.

toArray

```
toArray() -> newArray
```

This is just a local optimization of the mixed-in `toArray` from `Enumerable`.

This version aliases to `clone`, avoiding the default iterative behavior.

uniq

```
uniq() -> newArray
```

Produces a duplicate-free version of an array. If no duplicates are found, the original array is returned.

```
['Sam', 'Justin', 'Andrew', 'Dan', 'Sam'].uniq();
// -> ['Sam', 'Justin', 'Andrew', 'Dan']

['Prototype', 'prototype'].uniq();
// -> ['Prototype', 'prototype'] because String comparison is case-sensitive
```

Example 3.12.

Performance considerations

On large arrays with duplicates, this method has a potentially large performance cost:

- Since it does not require the array to be sorted, it has quadratic complexity.
- Since it relies on JavaScript's `Array.concat`, it will yield a new, intermediary array every time it encounters a new value (a value that wasn't already in the result array).

More efficient implementations could be devised. This page will get updated if such an optimization is committed.

without

```
without(value...) -> newArray
```

Produces a new version of the array that does not contain any of the specified values.

```
[3, 5, 6, 1, 20].without(3)
// -> [5, 6, 1, 20]

[3, 5, 6, 1, 20].without(20, 6)
// -> [3, 5, 1]
```

Example 3.13.

Chapter

4

Class

Prototype's object for class-based OOP.

Currently, inheritance is handled through Object.extend.

create

```
create() -> Function
```

Returns an function that acts like a Ruby class.

`Class.create()` returns a function that, when called, will fire its own `initialize` method. This allows for more Ruby-like OOP. It also lets you more easily subclass by overriding a parent's constructor.

```
var Animal = Class.create();
Animal.prototype = {
  initialize: function(name, sound) {
    this.name  = name;
    this.sound = sound;
  },

  speak: function() {
    alert(name + " says: " + sound + "!");
  }
};

var snake = new Animal("Ringneck", "hisssssssss");
snake.speak();
// -> alerts "Ringneck says: hisssssssss!"

var Dog = Class.create();

Dog.prototype = Object.extend(new Animal(), {
  initialize: function(name) {
    this.name  = name;
```

```
    this.sound = "woof";
  }
});

var fido = new Dog("Fido");
fido.speak();
// -> alerts "Fido says: woof!"
```

Example 4.1.

Chapter 5

Element

The `Element` object sports a flurry of powerful DOM methods which you can access either as methods of `Element` (but that's rather old-fashioned now) or **directly as methods of an extended element** (thanks to `Element.extend` for that added bit of syntactic sugar).

Before you pursue, you really *should* read "How Prototype extends the DOM"[1] which will walk you through the arcane inner workings of Prototype's magic DOM extension mechanism.

```
<div id="message" class=""></div>
```

```
// Toggle the CSS class name of div#message
$('message').addClassName('read');
// -> div#message

// You could also use a syntactic-sugar-free version:
Element.toggleClassName('message', 'read');
// -> div#message
```

Example 5.1.

Since most methods of `Element` return the element they are applied to, you can chain methods like so:

```
$('message').addClassName('read').update('I read this message!').setStyle({opacity: 0.5});
```

Example 5.2.

[1] http://http://www.prototypejs.org/learn/extensions

addClassName

```
addClassName(element, className) -> HTMLElement
```

Adds a CSS class to `element`.

```
<div id="mutsu" class="apple fruit"></div>
```

```
$('mutsu').addClassName('food')

$('mutsu').className
// -> 'apple fruit food'
$('mutsu').classNames()
// -> ['apple', 'fruit', 'food']
```

Example 5.3.

addMethods

```
addMethods([methods])
```

Takes a hash of `methods` and makes them available as methods of `extended` elements and of the `Element` object.

`Element.addMethods` makes it possible to mix in *your own* methods to the `Element` object, which you can later use as methods of `extended` elements - those returned by the `$()` utility, for example - or as methods of `Element`.

```
$(element).myOwnMethod([args...]);
```

Example 5.4.

Note that this will also works:

```
Element.myOwnMethod(element|id[, args...]);
```

Example 5.5.

To add new methods, simply feed `Element.addMethods` with a hash of methods. Note that each method's first argument *has to be* `element`. Inside each method, remember to pass `element` to `$()` and to return it to allow for method chaining if appropriate.

Here's what your hash should look like:

```
var myVeryOwnElementMethods = {
  myFirstMethod: function(element[, args...]){
    element = $(element);
    // do something
    return element;
  },

  mySecondMethod: function(element[, args...]){
    element = $(element);
    // do something else
    return element;
  }
};
```

Example 5.6.

One last warning before you pursue: `Element.addMethods` has a built in security which prevents you from overriding native element methods or properties (like `getAttribute` or `innerHTML` for instance), but nothing prevents you from overriding one of Prototype's method, so watch where you step!

Want clean, semantic markup, but need that extra `<div>` around your element, why not create an `Element.wrap('tagName')` method which encloses `element` in the provided `tagName` and returns the wrapper?

```
Element.addMethods({
  wrap: function(element, tagName) {
    element = $(element);
    var wrapper = document.createElement('tagName');
    element.parentNode.replaceChild(wrapper, element);
    wrapper.appendChild(element);
    return Element.extend(wrapper);
  }
});
```

Example 5.7.

which you'll be able to use like this:

```
// Before:
<p id="first">Some content...</p>
```

```
$(element).wrap('div');
// -> HTMLElement (div)
```

Example 5.8.

```
// After:
<div><p id="first">Some content...</p></div>
```

As you have thoughtfully decided that your `Element.wrap` method would return the newly created `<div>`, ready for prime time thanks to `Element.extend`, you can immediately chain a new method to it:

```
$(element).wrap('div').setStyle({backgroundImage: 'url(images/rounded-corner-top-left.png) top left'});
```

Example 5.9.

Are you using `Ajax.Updater` quite a bit around your web application to update DOM elements? Would you want a quick and nifty solution to cut down on the amount of code you are writing ? Try this:

```
Element.addMethods({
  ajaxUpdate: function(element, url, options){
    element = $(element);
    element.update('<img src="/images/spinner.gif" alt="loading..." />');
    new Ajax.Updater(element, url, options);
    return element;
  }
});
```

Example 5.10.

Now, whenever you wish to update the content of an element just do:

```
$(element).ajaxUpdate('/new/content');
// -> HTMLElement
```

Example 5.11.

This method will first replace the content of `element` with one of those nifty Ajax activity indicator. It will then create a new `Ajax.Updater`, which in turn will update the content of `element` with its request result, removing the spinner as it does.

Using `Element.addMethods` with no argument

There's a last dirty little secret to `Element.addMethods`. You can can call it *without* passing it an argument. What happens then? Well, it simply iterates over all of `Element.Methods`, `Element.Methods.Simulated`, `Form.Methods` and `Form.Element.Methods` and adds them to the relevant DOM elements (Form.Methods only gets added to, well the `form` element while `input`, `select` and `textarea` elements get extended with `Form.Element.Methods`).

When could that be usefull?

Imagine that you wish to add a method that deactivates a `submit` button and replaces its text with something like "Please wait...". You wouldn't want such a method to be applied to any element, would you? So here is how you would go about doing that:

```
Form.Element.Methods.processing = function(element, text) {
  element = $(element);
  if (element.tagName.toLowerCase() == 'input' && ['button', 'submit'].include(element.type))
    element.value = typeof text == 'undefined' ? 'Please wait...' : text;
  return element.disable();
};

Element.addMethods();
```

Example 5.12.

ancestors

```
ancestors(element) -> [HTMLElement...]
```

Collects all of `element`'s ancestors and returns them as an array of extended elements.

The returned array's first element is `element`'s direct ancestor (its `parentNode`), the second one is its grandparent and so on until the `html` element is reached. `html` will always be the last member of the array… unless you are looking for its ancestors obviously. But you wouldn't do that, would you ?

Note that all of Prototype's DOM traversal methods ignore text nodes and return element nodes only.

```
<html>
[…]
  <body>
    <div id="father">
      <div id="kid">
      </div>
    </div>
  </body>
</html>
```

```
$('kid').ancestors();
// -> [div#father, body, html]  // Keep in mind that
// the `body` and `html` elements will also be included!

document.getElementsByTagName('html')[0].ancestors();
// -> []
```

Example 5.13.

classNames

```
classNames(element) -> [className...]
```

Returns a new instance of `ClassNames`, an Enumerable object used to read and write class names of the element.

Practically, this means that you have access to your element's CSS `className` as an Enumerable rather than as the string that the native `className` property gives you (notice the singular form).

On top of that, this array is extended with a series of methods specifically targeted at dealing with CSS classNames: `set(className)`, `add(className)` and `remove(className)`. These are used internally by `Element.addClassName`, `Element.toggleClassName` and `Element.removeClassName`, but—unless you want to do some pretty wacky stuff—you usually won't need them.

```
<div id="mutsu" class="apple fruit food"></div>
```

```
$('mutsu').classNames()
// -> ['apple', 'fruit', 'food']

// change its class names:
$('mutsu').className = 'fruit round'

$('mutsu').classNames()
// -> ['fruit', 'round']
```

Example 5.14.

cleanWhitespace

```
cleanWhitespace(element) -> HTMLElement
```

Removes all of `element`'s text nodes which contain *only* whitespace. Returns `element`.

`Element.cleanWhitespace` removes whitespace-only text nodes. This can be very useful when using standard methods like `nextSibling`, `previousSibling`, `firstChild` or `lastChild` to walk the DOM.

If you only need to access element nodes (and not text nodes), try using `Element.up`, `Element.down`, `Element.next` and `Element.previous` instead. you won't regret it!

Consider the following HTML snippet:

```
<ul id="apples">
  <li>Mutsu</li>
  <li>McIntosh</li>
  <li>Ida Red</li>
</ul>
```

Let's grab what we think is the first list item:

```
var element = $('apples');
element.firstChild.innerHTML;
// -> undefined
```

Example 5.15.

That doesn't seem to work to well. Why is that ? `ul#apples`'s first child is actually a text node containing only whitespace that sits between `<ul id="apples">` and `<li>Mutsu</li>`.

Let's remove all this useless whitespace:

```
element.cleanWhitespace();
```

Example 5.16.

That's what our DOM looks like now:

```
<UL id="apples"><LI>Mutsu</LI><LI>McIntosh</LI><LI>Ida Red</LI></UL>
```

And guess what, `firstChild` now works as expected!

```
element.firstChild.innerHTML;
// -> 'Mutsu'
```

Example 5.17.

descendantOf

```
descendantOf(element, ancestor) -> Boolean
```

Checks if `element` is a descendant of `ancestor`.

As `Element.descendantOf` internally applies `$()` to `ancestor`, it accepts indifferently an element or an element's id as its second argument.

```
<div id="australopithecus">
  <div id="homo-herectus">
    <div id="homo-sapiens"></div>
  </div>
</div>
```

```
$('homo-sapiens').descendantOf('australopithecus');
// -> true

$('homo-herectus').descendantOf('homo-sapiens');
// -> false
```

Example 5.18.

descendants

```
descendants(element) -> [HTMLElement...]
```

Collects all of `element`'s descendants and returns them as an array of extended elements.

Note that all of Prototype's DOM traversal methods ignore text nodes and return element nodes only.

```
<div id="australopithecus">
  <div id="homo-herectus">
    <div id="homo-neanderthalensis"></div>
    <div id="homo-sapiens"></div>
  </div>
</div>
```

```
$('australopithecus').descendants();
// -> [div#homo-herectus, div#homo-neanderthalensis, div#homo-sapiens]

$('homo-sapiens').descendants();
// -> []
```

Example 5.19.

down

```
down(element[, cssRule][, index = 0]) -> HTMLElement | undefined
```

Returns element's first descendant (or the n-th descendant if `index` is specified) that matches `cssRule`. If no `cssRule` is provided, all descendants are considered. If no descendant matches these criteria, `undefined` is returned.

The `Element.down` method is part of Prototype's ultimate DOM traversal toolkit (check out `Element.up`, `Element.next` and `Element.previous` for some more Prototypish niceness). It allows precise index-based and/or CSS rule-based selection of any of the element's descendants.

As it totally ignores text nodes (it only returns elements), you don't have to worry about whitespace nodes.

And as an added bonus, all elements returned are already extended allowing chaining:

```
$(element).down(1).next('li', 2).hide();
```

Example 5.20.

Arguments

If no argument is passed, `element`'s first descendant is returned (this is similar as calling `firstChild` except `Element.down` returns an already extended element.

If an index is passed, `element`'s corresponding descendant is returned. (This is equivalent to selecting an element from the array of elements returned by the method `Element.descendants`.) Note that the first element has an index of 0.

If `cssRule` is defined, `Element.down` will return the first descendant that matches it. This is a great way to grab the first item in a list for example (just pass in 'li' as the method's first argument).

If both `cssRule` and `index` are defined, `Element.down` will collect all the descendants matching the given CSS rule and will return the one specified by the index.

In all of the above cases, if no descendant is found, `undefined` **will be returned.**

```
<ul id="fruits">
  <li id="apples">
    <ul>
      <li id="golden-delicious">Golden Delicious</li>
      <li id="mutsu" class="yummy">Mutsu</li>
      <li id="mcintosh" class="yummy">McIntosh</li>
      <li id="ida-red">Ida Red</li>
    </ul>
  </li>
</ul>
```

```
$('fruits').down();
// equivalent:
$('fruits').down(0);
// -> li#apple

$('fruits').down(3);
// -> li#golden-delicious

$('apples').down('li');
// -> li#golden-delicious

$('apples').down('li.yummy');
// -> li#mutsu

$('fruits').down('.yummy', 1);
// -> li#mcintosh

$('fruits').down(99);
// -> undefined
```

Example 5.21.

empty

```
empty(element) -> Boolean
```

Tests whether `element` is empty (i.e. contains only whitespace).

```
<div id="wallet">     </div>
<div id="cart">full!</div>
```

```
$('wallet').empty(); // -> true
$('cart').empty();   // -> false
```

Example 5.22.

extend

```
extend(element)
```

Extends `element` with *all* of the methods contained in Element.Methods and Element.Methods.Simulated. If `element` is an `input`, `textarea` or `select` tag, it will also be extended with the methods from Form.Element.Methods. If it is a `form` tag, it will also be extended with Form.Methods.

This is where the magic happens! By extending an element with Prototype's custom methods, we can achieve that syntactic sugar and ease of use we all crave for. For example, you can do the following with an extended element:

```
element.update('hello world');
```

Example 5.23.

And since most methods of `Element` return the element they are applied to, you can chain methods like so:

```
element.update('hello world').addClassName('greeting');
```

Example 5.24.

Note that all of the elements returned by `Element` methods are extended (yes even for methods like `Element.siblings` which return arrays of elements) and Prototype's flagship utility methods `$()` and `$$()` obviously also return extended elements. If you want to know more about how Prototype extends the DOM, jump to this article[2].

getDimensions

```
getDimensions(element) -> {height: Number, width: Number}
```

Finds the computed `width` and `height` of `element` and returns them as key/value pairs of an object.

This method returns correct values on elements whose display is set to `none` either in an inline style rule or in an CSS stylesheet.

In order to avoid calling the method twice, you should consider caching the values returned in a variable as shown below. If you only need `element`'s width or height, consider using `Element.getWidth` or `Element.getHeight` instead.

Note that all values are returned as *numbers only* although they are *expressed in pixels*.

```
<div id="rectangle" style="font-size: 10px; width: 20em; height: 10em"></div>
```

```
var dimensions = $('rectangle').getDimensions();
// -> {width: 200, height: 100}

dimensions.width;
// -> 200

dimensions.height;
// -> 100
```

Example 5.25.

getElementsByClassName

```
getElementsByClassName(element, className) -> [HTMLElement...]
```

Fetches all of `element`'s descendants which have a CSS class of `className` and returns them as an array of extended elements.

The returned array reflects the document order (e.g. an index of 0 refers to the topmost descendant of `element` with class `className`).

```
<ul id="fruits">
  <li id="apples">apples
    <ul>
      <li id="golden-delicious">Golden Delicious</li>
      <li id="mutsu" class="yummy">Mutsu</li>
      <li id="mcintosh" class="yummy">McIntosh</li>
      <li id="ida-red">Ida Red</li>
    </ul>
  </li>
  <li id="exotic" class="yummy">exotic fruits
    <ul>
```

[2] http://prototypejs.org/learn/extensions

```
      <li id="kiwi">kiwi</li>
      <li id="granadilla">granadilla</li>
    </ul>
  </li>
</ul>
```

```
$('fruits').getElementsByClassName('yummy');
// -> [li#mutsu, li#mcintosh, li#exotic]

$('exotic').getElementsByClassName('yummy');
// -> []
```

Example 5.26.

getElementsBySelector

```
getElementsBySelector(element, selector...) -> [HTMLElement...]
```

Takes an arbitrary number of CSS selectors (strings) and returns a document-order array of extended children of `element` that match any of them.

This method is very similar to $$() and therefore suffers from the same caveats. However, since it operates in a more restricted scope (`element`'s children) it is faster and therefore a much better alternative. The supported CSS syntax is identical, so please refer to the `$$()` docs for details.

```
<ul id="fruits">
  <li id="apples">
    <h3 title="yummy!">Apples</h3>
    <ul id="list-of-apples">
      <li id="golden-delicious" title="yummy!" >Golden Delicious</li>
      <li id="mutsu" title="yummy!">Mutsu</li>
      <li id="mcintosh">McIntosh</li>
      <li id="ida-red">Ida Red</li>
    </ul>
    <p id="saying">An apple a day keeps the doctor away.</p>
  </li>
</ul>
```

```
$('apples').getElementsBySelector('[title="yummy!"]');
// -> [h3, li#golden-delicious, li#mutsu]

$('apples').getElementsBySelector( 'p#saying', 'li[title="yummy!"]');
// -> [h3, li#golden-delicious, li#mutsu,  p#saying]

$('apples').getElementsBySelector('[title="disgusting!"]');
// -> []
```

Example 5.27.

getHeight

```
getHeight(element) -> Number
```

Finds and returns the computed height of `element`.

This method returns correct values on elements whose display is set to `none` either in an inline style rule or in an CSS stylesheet.

For performance reasons, if you need to query both width *and* height of `element`, you should consider using `Element.getDimensions` instead.

Note that the value returned is a *number only* although it is *expressed in pixels*.

```
<div id="rectangle" style="font-size: 10px; width: 20em; height: 10em"></div>
```

```
$('rectangle').getHeight();
// -> 100
```

Example 5.28.

getStyle

```
getStyle(element, property) -> String | null
```

Returns the given CSS property value of `element`. `property` can be specified in either of its CSS or camelized form.

This method looks up the CSS property of an element whether it was applied inline or in a stylesheet. It works around browser inconsistencies regarding `float`, `opacity`, which returns a value between 0 (fully transparent) and 1 (fully opaque), position properties (`left`, `top`, `right` and `bottom`) and when getting the dimensions (`width` or `height`) of hidden elements.

```
$(element).getStyle('font-size');
// equivalent:

$(element).getStyle('fontSize');
// -> '12px'
```

Example 5.29.

Note

Internet Explorer returns literal values while other browsers return computed values. Consider the following HTML snippet:

```
<style>
  #test {
    font-size: 12px;
    margin-left: 1em;
  }
</style>
<div id="test"></div>
```

```
$('test').getStyle('margin-left');
// -> '1em' in Internet Explorer,
// -> '12px' elsewhere.
```

Example 5.30.

Safari returns `null` for *any* non-inline property if the element is hidden (has `display` set to `'none'`).

Not all CSS shorthand properties are supported. You may only use the CSS properties described in the Document Object Model (DOM) Level 2 Style Specification[3].

getWidth

```
getWidth(element) -> Number
```

Finds and returns the computed width of `element`. This method returns correct values on elements whose display is set to `none` either in an inline style rule or in an CSS stylesheet.

For performance reasons, if you need to query both width *and* height of `element`, you should consider using `Element.getDimensions` instead.

Note that the value returned is a *number only* although it is *expressed in pixels*.

```
<div id="rectangle" style="font-size: 10px; width: 20em; height: 10em"></div>
```

```
$('rectangle').getWidth();  // -> 200
```

Example 5.31.

[3] http://www.w3.org/TR/DOM-Level-2-Style/css.html#CSS-ElementCSSInlineStyle

hasClassName

```
hasClassName(element, className) -> Boolean
```

Checks whether `element` has the given CSS `className`.

```
<div id="mutsu" class="apple fruit food"></div>
```

```
$('mutsu').hasClassName('fruit');
// -> true

$('mutsu').hasClassName('vegetable');
// -> false
```

Example 5.32.

hide

```
hide(element) -> HTMLElement
```

Hides and returns `element`.

```
<div id="error-message"></div>
```

```
$('error-message').hide();
// -> HTMLElement (and hides div#error-message)
```

Example 5.33.

Backwards compatibility change

Deprecated Usage

In previous versions of Prototype, you could pass an arbitrary number of elements to `Element.toggle`, `Element.show`, and `Element.hide`, for consistency, this is **no longer possible** in version 1.5!

You can however achieve a similar result by using Enumerables:

```
['content', 'navigation', 'footer'].each(Element.hide);
// -> ['content', 'navigation', 'footer']
// and hides #content, #navigation and #footer.
```

Example 5.34.

or even better:

```
$('content', 'navigation', 'footer').invoke('hide');
// -> [HTMLElement, HTMLElement, HTMLElement] (#content, #navigation and #footer)
// and hides #content, #navigation and #footer.
```

Example 5.35.

immediateDescendants

```
immediateDescendants(element) -> [HTMLElement...]
```

Collects all of the element's *immediate* descendants (i.e. *children*) and returns them as an array of extended elements.

The returned array reflects the children order in the document (e.g., an index of 0 refers to the topmost child of `element`).

Note that all of Prototype's DOM traversal methods ignore text nodes and return element nodes only.

```
<div id="australopithecus">
  <div id="homo-herectus">
    <div id="homo-neanderthalensis"></div>
    <div id="homo-sapiens"></div>
  </div>
</div>
```

```
$('australopithecus').immediateDescendants();
// -> [div#homo-herectus]

$('homo-herectus').immediateDescendants();
// -> [div#homo-neanderthalensis, div#homo-sapiens]

$('homo-sapiens').immediateDescendants();
// -> []
```

Example 5.36.

inspect

```
inspect(element) -> String
```

Returns the debug-oriented string representation of `element`. For more information on `inspect` methods, see `Object.inspect`.

```
<ul>
  <li id="golden-delicious">Golden Delicious</li>
  <li id="mutsu" class="yummy apple">Mutsu</li>
  <li id="mcintosh" class="yummy">McIntosh</li>
  <li</li>
</ul>
```

```
$('golden-delicious').inspect();
// -> '<li id="golden-delicious">'

$('mutsu').inspect();
// -> '<li id="mutsu" class="yummy apple">'

$('mutsu').next().inspect();
// -> '<li>'
```

Example 5.37.

makeClipping

```
makeClipping(element) -> HTMLElement
```

Simulates the poorly supported CSS `clip` property by setting `element`'s overflow value to `'hidden'`. Returns `element`.

To undo clipping, use Element.undoClipping.

The visible area is determined by `element`'s width and height.

```
<div id="framer">
  <img src="/assets/2007/1/14/chairs.jpg" alt="example" />
</div>
```

```
$('framer').makeClipping().setStyle({width: '100px', height: '100px'});
// -> HTMLElement
```

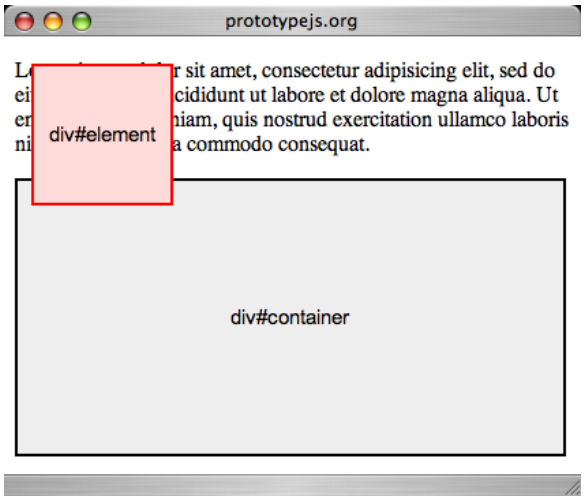
Example 5.38.

makePositioned

```
makePositioned(element) -> HTMLElement
```

Allows for the easy creation of CSS containing block[4] by setting an `element`'s CSS position to `'relative'` if its initial position is either `'static'` or `undefined`. Returns `element`. To revert back to `element`'s original CSS position, use undoPositioned().

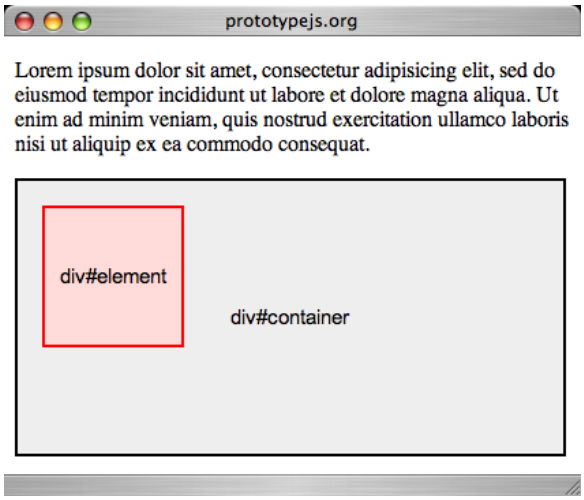
```
<p>lorem […]</p>
<div id="container">
  <div id="element" style="position: absolute; top: 20px; left: 20px;"></div>
</div>
```



To position `div#element` *relatively* to it's parent element:

```
$('container').makePositioned();  // -> HTMLElement
```

Which yields the expected layout:



[4] http://www.w3.org/TR/CSS21/visudet.html#containing-block-details

match

```
match(element, selector) -> Boolean
```

Checks if `element` matches the given CSS `selector`.

```
<ul id="fruits">
  <li id="apples">
    <ul>
      <li id="golden-delicious">Golden Delicious</li>
      <li id="mutsu" class="yummy">Mutsu</li>
      <li id="mcintosh" class="yummy">McIntosh</li>
      <li id="ida-red">Ida Red</li>
    </ul>
  </li>
</ul>
```

```
$('fruits').match('ul');  // -> true

$('mcintosh').match('li#mcintosh.yummy');  // -> true

$('fruits').match('p');  // -> false
```

Example 5.39.

next

```
next(element[, cssRule][, index = 0]) -> HTMLElement | undefined
```

Returns `element`'s following sibling (or the *index'th* one, if `index` is specified) that matches `cssRule`. If no `cssRule` is provided, all following siblings are considered. If no following sibling matches these criteria, `undefined` is returned.

The `Element.next` method is part of Prototype's ultimate DOM traversal toolkit (check out `Element.up`, `Element.down` and `Element.previous` for some more Prototypish niceness). It allows precise index-based and/or CSS rule-based selection of any of `element`'s **following siblings**. (Note that two elements are considered siblings if they have the same parent, so for example, the `head` and `body` elements are siblings—their parent is the `html` element.)

As it totally ignores text nodes (it only returns elements), you don't have to worry about whitespace-only nodes.

And as an added bonus, all elements returned are already extended allowing chaining:

```
$(element).down(1).next('li', 2).hide();
```

Example 5.40.

Arguments

If no argument is passed, `element`'s following sibling is returned (this is similar as calling `nextSibling` except `Element.next` returns an already extended element).

If an index is passed, `element`'s corresponding following sibling is returned. (This is equivalent to selecting an element from the array of elements returned by the method `Element.nextSiblings`). Note that the sibling *right below* `element` has an index of 0.

If `cssRule` is defined, `Element.next` will return the `element` first following sibling that matches it.

If both `cssRule` and `index` are defined, `Element.next` will collect all of `element`'s following siblings matching the given CSS rule and will return the one specified by the index.

In all of the above cases, if no following sibling is found, `undefined` **will be returned.**

```
<ul id="fruits">
  <li id="apples">
    <h3 id="title">Apples</h3>
    <ul id="list-of-apples">
      <li id="golden-delicious">Golden Delicious</li>
      <li id="mutsu">Mutsu</li>
      <li id="mcintosh" class="yummy">McIntosh</li>
      <li id="ida-red" class="yummy">Ida Red</li>
    </ul>
    <p id="saying">An apple a day keeps the doctor away.</p>
  </li>
</ul>
```

```
$('list-of-apples').next();
// equivalent:
$('list-of-apples').next(0);
// -> p#sayings

$('title').next(1);
// -> ul#list-of-apples

$('title').next('p');
// -> p#sayings

$('golden-delicious').next('.yummy');
// -> li#mcintosh

$('golden-delicious').next('.yummy', 1);
// -> li#ida-red

$('ida-red').next();
// -> undefined
```

Example 5.41.

nextSiblings

```
nextSiblings(element) -> [HTMLElement...]
```

Collects all of `element`'s next siblings and returns them as an array of extended elements.

Two elements are siblings if they have the same parent. So for example, the `head` and `body` elements are siblings (their parent is the `html` element). Next siblings are simply the ones which follow `element` in the document.

The returned array reflects the siblings order in the document (e.g. an index of 0 refers to the sibling right below `element`).

Note that all of Prototype's DOM traversal methods ignore text nodes and return element nodes only.

```
<ul>
  <li id="golden-delicious">Golden Delicious</li>
  <li id="mutsu">Mutsu</li>
  <li id="mcintosh">McIntosh</li>
  <li id="ida-red">Ida Red</li>
</ul>
```

```
$('mutsu').nextSiblings();
// -> [li#mcintosh, li#ida-red]

$('ida-red').nextSiblings();
// -> []
```

Example 5.42.

observe

```
observe(element, eventName, handler[, useCapture = false]) -> HTMLElement
```

Registers an event handler on `element` and returns `element`.

This is a simple proxy for Event.observe. Please refer to it for further information.

```
$(element).observe('click', function(event){
  alert(Event.element(event).innerHTML);
 });
// -> HTMLElement (and will display an alert dialog containing
// element's innerHTML when element is clicked).
```

Example 5.43.

previous

```
previous(element[, cssRule][, index = 0]) -> HTMLElement | undefined
```

Returns `element`'s previous sibling (or the *index'th* one, if `index` is specified) that matches `cssRule`. If no `cssRule` is provided, all previous siblings are considered. If no previous sibling matches these criteria, `undefined` is returned.

The `Element.previous` method is part of Prototype's ultimate DOM traversal toolkit (check out `Element.up`, `Element.down` and `Element.next` for some more Prototypish niceness). It allows precise index-based and/or CSS rule-based selection of any of `element`'s **previous siblings**. (Note that two elements are considered siblings if they have the same parent, so for example, the `head` and `body` elements are siblings—their parent is the `html` element.)

As it totally ignores text nodes (it only returns elements), you don't have to worry about whitespace-only nodes.

And as an added bonus, all elements returned are already extended allowing chaining:

```
$(element).down(1).next('li', 2).hide();
```

Example 5.44.

Arguments

If no argument is passed, `element`'s previous sibling is returned (this is similar as calling `previousSibling` except `Element.previous` returns an already extended element).

If an index is passed, `element`'s corresponding previous sibling is returned. (This is equivalent to selecting an element from the array of elements returned by the method `previousSiblings()`). Note that the sibling *right above* `element` has an index of 0.

If `cssRule` is defined, `Element.previous` will return the `element` first previous sibling that matches it.

If both `cssRule` and `index` are defined, `Element.previous` will collect all of `element`'s previous siblings matching the given CSS rule and will return the one specified by the index.

In all of the above cases, if no previous sibling is found, `undefined` **will be returned.**

```
<ul id="fruits">
  <li id="apples">
    <h3>Apples</h3>
    <ul id="list-of-apples">
      <li id="golden-delicious" class="yummy">Golden Delicious</li>
      <li id="mutsu" class="yummy">Mutsu</li>
      <li id="mcintosh">McIntosh</li>
      <li id="ida-red">Ida Red</li>
    </ul>
    <p id="saying">An apple a day keeps the doctor away.</p>
  </li>
</ul>
```

```
$('saying').previous();
// equivalent:
$('saying').previous(0);
// -> ul#list-of-apples

$('saying').previous(1);
// -> h3

$('saying').previous('h3');
// -> h3

$('ida-red').previous('.yummy');
// -> li#mutsu

$('ida-red').previous('.yummy', 1);
// -> li#golden-delicious

$('ida-red').previous(5);
// -> undefined
```

Example 5.45.

previousSiblings

```
previousSiblings(element) -> [HTMLElement...]
```

Collects all of `element`'s previous siblings and returns them as an array of extended elements.

Two elements are siblings if they have the same parent. So for example, the `head` and `body` elements are siblings (their parent is the `html` element). Previous siblings are simply the ones which precede `element` in the document.

The returned array reflects the siblings *inversed* order in the document (e.g. an index of 0 refers to the lowest sibling i.e., the one closest to `element`).

Note that all of Prototype's DOM traversal methods ignore text nodes and return element nodes only.

```
<ul>
  <li id="golden-delicious">Golden Delicious</li>
  <li id="mutsu">Mutsu</li>
  <li id="mcintosh">McIntosh</li>
  <li id="ida-red">Ida Red</li>
</ul>
```

```
$('mcintosh').previousSiblings();
// -> [li#mutsu, li#golden-delicious]

$('golden-delicious').previousSiblings();
// -> []
```

Example 5.46.

readAttribute

```
readAttribute(element, attribute) -> String | null
```

Returns the value of `element`'s `attribute` or `null` if `attribute` has not been specified.

This method serves two purposes. First it acts as a simple wrapper around `getAttribute` which isn't a "real" function in Safari and Internet Explorer (it doesn't have `.apply` or `.call` for instance). Secondly, it cleans up the horrible mess Internet Explorer makes when handling attributes.

```
<a id="tag" href="/tags/prototype" rel="tag" title="view related bookmarks." my_widget="some info.">Prototype</a>
```

```
$('tag').readAttribute('href');
// -> '/tags/prototype'

$('tag').readAttribute('title');
// -> 'view related bookmarks.'

$('tag').readAttribute('my_widget');
// -> 'some info.'
```

Example 5.47.

recursivelyCollect

```
recursivelyCollect(element, property) -> [HTMLElement...]
```

Recursively collects elements whose relationship is specified by `property`. `property` has to be a *property* (a method won't do!) of `element` that points to a *single* DOM node. Returns an array of extended elements.

This method is used internally by `Element.ancestors`, `Element.descendants`, `Element.nextSiblings`, `Element.previousSiblings` and `Element.siblings` which offer really convenient way to grab elements, so directly accessing `Element.recursivelyCollect` should seldom be needed. However, if you are after something out of the ordinary, it is the way to go.

Note that all of Prototype's DOM traversal methods ignore text nodes and return element nodes only.

```
<ul id="fruits">
  <li id="apples">
    <ul id="list-of-apples">
      <li id="golden-delicious"><p>Golden Delicious</p></li>
      <li id="mutsu">Mutsu</li>
      <li id="mcintosh">McIntosh</li>
      <li id="ida-red">Ida Red</li>
    </ul>
  </li>
</ul>
```

```
$('fruits').recursivelyCollect('firstChild');
// -> [li#apples, ul#list-of-apples, li#golden-delicious, p]
```

Example 5.48.

remove

```
remove(element) -> HTMLElement
```

Completely removes `element` from the document and returns it.

If you would rather just hide the element and keep it around for further use, try `Element.hide` instead.

```
// Before:
<ul>
  <li id="golden-delicious">Golden Delicious</li>
  <li id="mutsu">Mutsu</li>
  <li id="mcintosh">McIntosh</li>
  <li id="ida-red">Ida Red</li>
</ul>
```

```
$('mutsu').remove();  // -> HTMLElement (and removes li#mutsu)
```

Example 5.49.

```
// After:
<ul>
  <li id="golden-delicious">Golden Delicious</li>
  <li id="mcintosh">McIntosh</li>
  <li id="ida-red">Ida Red</li>
</ul>
```

removeClassName

```
removeClassName(element, className) -> HTMLElement
```

Removes `element`'s CSS `className` and returns `element`.

```
<div id="mutsu" class="apple fruit food"></div>
```

```
$('mutsu').removeClassName('food');  // -> HTMLElement
$('mutsu').classNames();  // -> ['apple', 'fruit']
```

Example 5.50.

replace

```
replace(element[, html]) -> HTMLElement
```

Replaces `element` by the content of the `html` argument and returns the removed `element`.

`html` can be either plain text, an HTML snippet or any JavaScript object which has a `toString()` method.

If it contains any `<script>` tags, these will be evaluated after `element` has been replaced (`Element.replace()` internally calls String#evalScripts).

Note that if no argument is provided, `Element.replace` will simply clear `element` of its content. However, using `Element.remove` to do so is both faster and more standard compliant.

```
<div id="food">
  <div id="fruits">
    <p id="first">Kiwi, banana <em>and</em> apple.</p>
  </div>
</div>
```

Passing an HTML snippet:

```
$('first').replace('<ul id="favorite"><li>kiwi</li><li>banana</li><li>apple</li></ul>');
// -> HTMLElement (p#first)
$('fruits').innerHTML;
// -> '<ul id="favorite"><li>kiwi</li><li>banana</li><li>apple</li></ul>'
```

Example 5.51.

Again, with a <script> tag thrown in:

```
$('favorite').replace('<p id="still-first">Melon, oranges <em>and</em> grapes.</p><script>alert("removed!")</script>');
// -> HTMLElement (ul#favorite) and prints "removed!" in an alert dialog.
$('fruits').innerHTML
// -> '<p id="still-first">Melon, oranges <em>and</em> grapes.</p>'
```

Example 5.52.

With plain text:

```
$('still-first').replace('Melon, oranges and grapes.');  // -> HTMLElement (p#still-first)
$('fruits').innerHTML  // -> 'Melon, oranges and grapes.'
```

Example 5.53.

Finally, relying on the `toString()` method:

```
$('fruits').update(123);  // -> HTMLElement
$('food').innerHTML;  // -> '123'
```

Example 5.54.

scrollTo

```
scrollTo(element) -> HTMLElement
```

Scrolls the window so that `element` appears at the top of the viewport. Returns `element`.

This has a similar effect than what would be achieved using HTML anchors[5] (except the browser's history is not modified).

```
$(element).scrollTo();
// -> HTMLElement
```

Example 5.55.

setStyle

```
setStyle(element, styles) -> HTMLElement
```

Modifies `element`'s CSS style properties. Styles are passed as a hash of property-value pairs in which the properties are specified in their camelized form.

```
$(element).setStyle({
  backgroundColor: '#900',
  fontSize: '12px'
});
// -> HTMLElement
```

Example 5.56.

[5] http://www.w3.org/TR/html401/struct/links.html#h-12.2.3

Note

The method transparently deals with browser inconsistencies for `float`—however, as `float` is a reserved keyword, you must either escape it or use `cssFloat` instead—and `opacity`—which accepts values between 0 (fully transparent) and 1 (fully opaque). You can safely use either of the following across all browsers:

```
$(element).setStyle({
  cssFloat: 'left',
  opacity: 0.5
});
// -> HTMLElement

$(element).setStyle({
  'float': 'left', // notice how float is surrounded by single quotes
  opacity: 0.5
});
// -> HTMLElement
```

Example 5.57.

Not all CSS shorthand properties are supported. You may only use the CSS properties described in the Document Object Model (DOM) Level 2 Style Specification[6].

show

```
show(element) -> HTMLElement
```

Displays and returns `element`.

```
<div id="error-message" style="display:none;"></div>
```

```
$('error-message').show();
// -> HTMLElement (and displays div#error-message)
```

Example 5.58.

[6] http://www.w3.org/TR/DOM-Level-2-Style/css.html#CSS-ElementCSSInlineStyle

Note

`Element.show` *cannot* display elements hidden via CSS stylesheets. Note that this is not a Prototype limitation but a consequence of how the CSS `display` property works.

```
<style>
  #hidden-by-css {
    display: none;
  }
</style>

[…]

<div id="hidden-by-css"></div>
```

```
$('hidden-by-css').show(); // DOES NOT WORK!
// -> HTMLElement (div#error-message is still hidden!)
```

Example 5.59.

Backwards compatibility change

Deprecated Usage

In previous versions of Prototype, you could pass an arbitrary number of elements to `Element.toggle`, `Element.show`, and `Element.hide`, for consistency, this is **no longer possible** in version 1.5!

You can however achieve a similar result by using Enumerables:

```
['content', 'navigation', 'footer'].each(Element.show);
// -> ['content', 'navigation', 'footer']
// and displays #content, #navigation and #footer.

//or even better:
$('content', 'navigation', 'footer').invoke('show');
// -> [HTMLElement, HTMLElement, HTMLElement] (#content, #navigation and #footer)
// and displays #content, #navigation and #footer.
```

Example 5.60.

siblings

```
siblings(element) -> [HTMLElement...]
```

Collects all of element's siblings and returns them as an array of extended elements.

Two elements are siblings if they have the same parent. So for example, the `head` and `body` elements are siblings (their parent is the `html` element).

The returned array reflects the siblings order in the document (e.g. an index of 0 refers to `element`'s top-most sibling).

Note that all of Prototype's DOM traversal methods ignore text nodes and return element nodes only.

```
<ul>
  <li id="golden-delicious">Golden Delicious</li>
  <li id="mutsu">Mutsu</li>
  <li id="mcintosh">McIntosh</li>
  <li id="ida-red">Ida Red</li>
</ul>
```

```
$('mutsu').siblings();
// -> [li#golden-delicious, li#mcintosh, li#ida-red]
```

Example 5.61.

stopObserving

```
stopObserving(element, eventName, handler) -> HTMLElement
```

Unregisters `handler` and returns `element`.

This is a simple proxy for Event.stopObserving. Please refer to it for further information.

```
$(element).stopObserving('click', coolAction);
// -> HTMLElement (and unregisters the 'coolAction' event handler).
```

Example 5.62.

toggle

```
toggle(element) -> HTMLElement
```

Toggles the visibility of `element`.

```
<div id="welcome-message"></div>
<div id="error-message" style="display:none;"></div>
```

```
$('welcome-message').toggle();
// -> HTMLElement (and hides div#welcome-message)

$('error-message').toggle();
// -> HTMLElement (and displays div#error-message)
```

Example 5.63.

Note

`Element.toggle` *cannot* display elements hidden via CSS stylesheets. Note that this is not a Prototype limitation but a consequence of how the CSS `display` property works.

```
<style>
  #hidden-by-css {
    display: none;
  }
</style>
[…]
<div id="hidden-by-css"></div>
```

```
$('hidden-by-css').toggle(); // WONT' WORK!
// -> HTMLElement (div#hidden-by-css is still hidden!)
```

Example 5.64.

Deprecated Usage

In previous versions of Prototype, you could pass an arbitrary number of elements to `Element.toggle`, `Element.show`, and `Element.hide`, for consistency, this is **no longer possible** in version 1.5!

You can however achieve a similar result by using Enumerables:

```
['error-message', 'welcome-message'].each(Element.toggle);
// -> ['error-message', 'welcome-message']
// and toggles the visibility of div#error-message and div#confirmation-message.

//or even better:
$('error-message', 'welcome-message').invoke('toggle');
// -> [HTMLElement, HTMLElement] (div#error-message and div#welcome-message)
// and toggles the visibility of div#error-message and div#confirmation-message.
```

Example 5.65.

toggleClassName

```
toggleClassName(element, className) -> HTMLElement
```

Toggles `element`'s CSS `className` and returns `element`.

```
<div id="mutsu" class="apple"></div>
```

```
$('mutsu').hasClassName('fruit');  // -> false

$('mutsu').toggleClassName('fruit');  // -> element

$('mutsu').hasClassName('fruit');  // -> true
```

Example 5.66.

undoClipping

```
undoClipping(element) -> HTMLElement
```

Sets `element`'s CSS `overflow` property back to the value it had before Element.makeClipping() was applied. Returns `element`.

```
<div id="framer">
  <img src="/assets/2007/1/14/chairs.jpg" alt="example" />
</div>
```

```
$('framer').undoClipping();
// -> HTMLElement (and sets the CSS overflow property to its original value).
```

Example 5.67.

undoPositioned

```
undoPositioned(element) -> HTMLElement
```

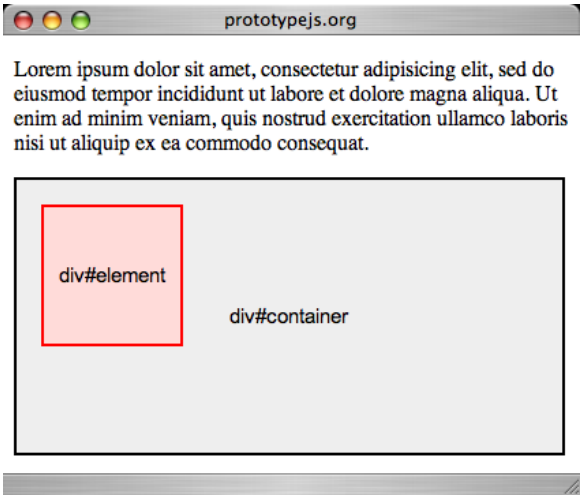
Sets `element` back to the state it was *before* Element.makePositioned was applied to it. Returns `element`.

`element`'s absolutely positioned children will now have their positions set relatively to `element`'s nearest ancestor with a CSS `position` of `'absolute'`, `'relative'` or `'fixed'`.

```
<p>lorem […]</p>
<div id="container">
  <div id="element" style="position: absolute; top: 20px; left: 20px;"></div>
</div>
```

```
$('container').makePositioned();
// -> HTMLElement
```

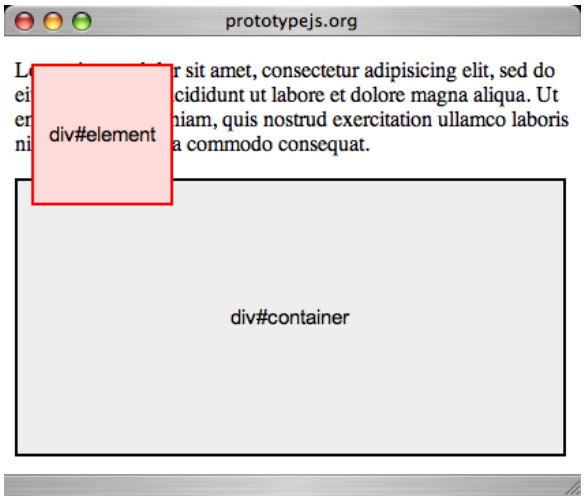
Example 5.68.



To return to the orginal layout, use `Element.undoPositioned`;

```
$('container').undoPositioned();
// -> HTMLElement
```

Example 5.69.



up

```
up([cssRule][, index = 0]) -> HTMLElement | undefined
```

Returns `element`'s first ancestor (or the *index'th* ancestor, if `index` is specified) that matches `cssRule`. If no `cssRule` is provided, all ancestors are considered. If no ancestor matches these criteria, `undefined` is returned.

The `Element.up` method is part of Prototype's ultimate DOM traversal toolkit (check out `Element.down`, `Element.next` and `Element.previous` for some more Prototypish niceness). It allows precise index-based and/or CSS rule-based selection of any of `element`'s **ancestors**.

As it totally ignores text nodes (it only returns elements), you don't have to worry about whitespace-only nodes.

And as an added bonus, all elements returned are already extended allowing chaining:

```
$(element).down(1).next('li', 2).hide();
```

Example 5.70.

Walking the DOM has never been that easy!

Arguments

If no argument is passed, `element`'s first ancestor is returned (this is similar as calling `parentNode` except `Element.up` returns an already extended element.

If an index is passed, `element`'s corresponding ancestor is is returned. (This is equivalent to selecting an element from the array of elements returned by the method `Element.ancestors`). Note that the first element has an index of 0.

If `cssRule` is defined, `Element.up` will return the first ancestor that matches it.

If both `cssRule` and `index` are defined, `Element.up` will collect all the ancestors matching the given CSS rule and will return the one specified by the index.

In all of the above cases, if no descendant is found, `undefined` **will be returned.**

```
<html>
  […]
  <body>
    <ul id="fruits">
      <li id="apples" class="keeps-the-doctor-away">
        <ul>
          <li id="golden-delicious">Golden Delicious</li>
          <li id="mutsu" class="yummy">Mutsu</li>
          <li id="mcintosh" class="yummy">McIntosh</li>
          <li id="ida-red">Ida Red</li>
        </ul>
      </li>
    </ul>
  </body>
</html>
```

```
$('fruits').up();
// equivalent:
$('fruits').up(0);
// -> body

$('mutsu').up(2);
// -> ul#fruits

$('mutsu').up('li');
// -> li#apples

$('mutsu').up('.keeps-the-doctor-away');
// -> li#apples

$('mutsu').up('ul', 1);
// -> ul#fruits

$('mutsu').up('div');
// -> undefined
```

Example 5.71.

update

```
update(element[, newContent]) -> HTMLElement
```

Replaces the content of `element` with the provided `newContent` argument and returns `element`.

`newContent` can be plain text, an HTML snippet, or any JavaScript object which has a `toString()` method.

If it contains any `<script>` tags, these will be evaluated after `element` has been updated (`Element.update` internally calls String.evalScripts).

If no argument is provided, `Element.update` will simply clear `element` of its content.

Note that this method allows seamless content update of table related elements in Internet Explorer 6 and beyond.

```
<div id="fruits">carrot, eggplant and cucumber</div>
```

Passing a regular string:

```
$('fruits').update('kiwi, banana and apple');
// -> HTMLElement
$('fruits').innerHTML
// -> 'kiwi, banana and apple'
```

Example 5.72.

Clearing the element's content:

```
$('fruits').update();
// -> HTMLElement
$('fruits').innerHTML;
// -> '' (an empty string)
```

Example 5.73.

And now inserting an HTML snippet:

```
$('fruits').update('<p>Kiwi, banana <em>and</em> apple.</p>');
// -> HTMLElement
$('fruits').innerHTML;
// -> '<p>Kiwi, banana <em>and</em> apple.</p>'
```

Example 5.74.

… with a <script> tag thrown in:

```
$('fruits').update('<p>Kiwi, banana <em>and</em> apple.</p><script>alert("updated!")</script>');
// -> HTMLElement (and prints "updated!" in an alert dialog).
$('fruits').innerHTML;
// -> '<p>Kiwi, banana <em>and</em> apple.</p>'
```

Example 5.75.

Relying on the `toString()` method:

```
$('fruits').update(123);
// -> HTMLElement
$('fruits').innerHTML;
// -> '123'
```

Example 5.76.

Finally, you can do some pretty funky stuff by defining your own `toString()` method on your custom objects:

```
var Fruit = Class.create();
Fruit.prototype = {
  initialize: function(fruit){
    this.fruit = fruit;
  },
  toString: function(){
    return 'I am a fruit and my name is "' + this.fruit + '".';
  }
}
var apple = new Fruit('apple');

$('fruits').update(apple);
$('fruits').innerHTML;
// -> 'I am a fruit and my name is "apple".'
```

Example 5.77.

visible

```
visible(element) -> Boolean
```

Returns a `Boolean` indicating whether or not `element` is visible (i.e. whether its inline `style` property is set to `"display: none;"`).

```
<div id="visible"></div>
<div id="hidden" style="display: none;"></div>
```

```
$('visible').visible();
// -> true

$('hidden').visible();
// -> false
```

Example 5.78.

Note

Styles applied via a CSS stylesheet are *not* taken into consideration. Note that this is not a Prototype limitation, it is a CSS limitation.

```
<style>
  #hidden-by-css {
    display: none;
  }
</style>

[…]

<div id="hidden-by-css"></div>
```

```
$('hidden-by-css').visible();
// -> true
```

Example 5.79.

Chapter

6

Enumerable

`Enumerable` provides a large set of useful methods for *enumerations*, that is, objects that act as collections of values. It is a cornerstone of Prototype.

`Enumerable` is what we like to call a *module*: a consistent set of methods intended not for independent use, but for *mixin*: incorporation into other objects that "fit" with it. This meaning of the term "module" is borrowed from the Ruby world, which is fitting enough, since `Enumerable` attempts to mimic at least part of its Ruby-world namesake.

Quite a few objects, in Prototype, mix `Enumerable` in already. The most visible cases are `Array` and `Hash`, but you'll find it in less obvious spots as well, such as in `ObjectRange` and various DOM- or AJAX-related objects.

Aliases: it's all about having it your way

Just like its Ruby counterpart, `Enumerable` cares about your comfort enough to provide more than one name for a few behaviors. This is intended to reduce your learning curve when your technical background made you familiar with one name or another. However, the documentation attempts to clearly state when one name is "preferred" over the other (perhaps due to readability, widely accepted intuitiveness, etc.).

Here are the aliases you'll find in `Enumerable`:

- `map` is the same as `collect`.
- `find` is the preferred way of using `detect`.
- `findAll` is the same as `select`.
- `include` is the same as `member`.
- `entries` is the same as `toArray`.

Using it efficiently

When using `Enumerable`, beginners often create sub-par code, performance-wise, by simple lack of familiarity with the API. There are several use cases when one method will be significantly faster (and often make for more readable code!) than another. Here are the two main points about this.

`collect`, `invoke`, `pluck` and `each`: thinking about the use case

Beginners tend to use `each` whenever they need to manipulate all elements in the enumeration, and `collect` whenever they need to yield a value the same way for each element. This is the proper way for the generic case, but there are specific use cases where it can be written way more concisely, more elegantly, and with much better performance.

- When you need to invoke the same method on all the elements, go with `invoke`.
- When you need to fetch the same property on all the elements, go with `pluck`.

`reject` and `findAll` vs. `partition`

The `findAll/select` methods retrieve all the elements that match a given predicate. Conversely, the `reject` method retrieves all the elements that *do not* match a predicate. In the specific case where you need *both sets*, you can avoid looping twice: just use `partition`.

Mixing `Enumerable` in your own objects

So, let's say you've created your very own collection-like object (say, some sort of Set, or perhaps something that dynamically fetches data ranges from the server side, lazy-loading style). You want to be able to mix `Enumerable` in (and we commend you for it). How do you go about this?

The `Enumerable` module basically makes only one requirement on your object: it must provide a method named `_each` (note the leading underscore), that will accept a function as its unique argument, and will contain the actual "raw iteration" algorithm, invoking its argument with each element in turn.

As detailed in the documentation for `each`, `Enumerable` provides all the extra layers (handling iteration short-circuits, passing numerical indices, etc.). You just need to implement the actual basic iteration, as fits your internal structure.

If this leaves you goggling, just have a look at Prototype's `Array`, `Hash` or `ObjectRange` objects' source code. They all begin with their own `_each` method, which should help you grasp the idea.

Once you're done with this, you just need to mix `Enumerable` in, which you'll usually do *before* defining your methods, so as to make sure whatever overrides you provide for `Enumerable` methods will indeed prevail. In short, your code will probably end up looking like this:

```
var YourObject = Class.create();
Object.extend(YourObject.prototype, Enumerable);
Object.extend(YourObject.prototype, {
  initialize: function() { // with whatever constructor arguments you need
    // Your construction code
  },

  _each: function(iterator) {
    // Your iteration code, invoking iterator at every turn
  },

  // Your other methods here, including Enumerable overrides
});
```

Example 6.1.

Then, obviously, your object can be used like this:

```
var obj = new YourObject();
  // Whatever use here, e.g. to fill it up
  obj.pluck('somePropName');
  obj.invoke('someMethodName');
  obj.size();
  // etc.
```

Example 6.2.

all

```
all([iterator = Prototype.K]) -> Boolean
```

Determines whether all the elements are boolean-equivalent to `true`, either directly or through computation by the provided iterator.

The code obviously short-circuits as soon as it finds an element that "fails" (that is boolean-equivalent to `false`). If no iterator is provided, the elements are used directly. Otherwise, each element is passed to the iterator, and the result value is used for boolean equivalence.

```
[].all()
// -> true (empty arrays have no elements that could be false-equivalent)

$R(1, 5).all()
// -> true (all values in [1..5] are true-equivalent)

[0, 1, 2].all()
// -> false (with only one loop cycle: 0 is false-equivalent)

[9, 10, 15].all(function(n) { return n >= 10; })
```

```
// -> false (the iterator will return false on 9)

$H({ name: 'John', age: 29, oops: false }).all(function(pair) { return pair.value; })
// -> false (the oops/false pair yields a value of false)
```

Example 6.3.

See also

If you need to determine whether at least one element matches a criterion, you would be better off using `any`.

any

```
any([iterator = Prototype.K]) -> Boolean
```

Determines whether at least one element is boolean-equivalent to `true`, either directly or through computation by the provided iterator.

The code obviously short-circuits as soon as it finds an element that "passes" (that is boolean-equivalent to `true`). If no iterator is provided, the elements are used directly. Otherwise, each element is passed to the iterator, and the result value is used for boolean equivalence.

```
[].any()
// -> false (empty arrays have no elements that could be true-equivalent)

$R(0, 2).any()
// -> true (on the second loop cycle, 1 is true-equivalent)

[2, 4, 6, 8, 10].any(function(n) { return 0 == n % 3; })
// -> true (the iterator will return true on 6: the array does have 1+ multiple of 3)

$H({ opt1: null, opt2: false, opt3: '', opt4: 'pfew!' }).any(function(pair) { return pair.value; })
// -> true (thanks to the opt4/'pfew!' pair, whose value is true-equivalent)
```

Example 6.4.

See also

If you need to determine whether all elements match a criterion, you would be better off using `all`.

collect

```
collect(iterator) -> Array
```

Returns the results of applying the iterator to each element. Aliased as `map`.

This is a sort of Swiss-Army knife for sequences. You can turn the original values into virtually anything! Here are a few examples:

```
['Hitch', "Hiker's", 'Guide', 'To', 'The', 'Galaxy'].collect(function(s) {
  return s.charAt(0).toUpperCase();
}).join('')
// -> 'HHGTTG'

$R(1,5).collect(function(n) {
  return n * n;
})
// -> [1, 4, 9, 16, 25]
```

Example 6.5.

Optimized versions

There are two very common use-cases that will be much better taken care of by specialized variants.

First, the method-calling scenario: you want to invoke the same method on all elements, possibly with arguments, and use the result values. This can be achieved easily with `invoke`.

Second, the property-fetching scenario: you want to fetch the same property on all elements, and use those. This is a breeze with `pluck`.

Both variants perform much better than `collect`, since they avoid lexical closure costs.

detect

```
detect(iterator) -> firstElement | undefined
```

Finds the first element for which the iterator returns `true`. Aliased by the `find` method, which is considered the more readable way of using it.

each

```
each(iterator) -> Enumerable
```

The cornerstone of `Enumerable`. It lets you iterate over all the elements in a generic fashion, then returns the `Enumerable`, thereby allowing chain-calling.

Iterations based on `each` are the core of `Enumerable`. The iterator function you pass will be called with two parameters:

1. The current element in the iteration.
2. The numerical index, starting at zero, of the current cycle. This second parameter is unused (and therefore undeclared) most of the time, but can come in handy sometimes.

`$break` and `$continue`

Regular loops can be short-circuited in JavaScript using the `break` and `continue` statements. However, when using iterator functions, your code is outside of the loop scope: the looping code happens behind the scene.

In order to provide you with equivalent (albeit less optimal) functionality, Prototype provides two global exception objects, `$break` and `$continue`. Throwing these is equivalent to using the corresponding native statement in a vanilla loop. These exceptions are properly caught internally by the `each` method.

Deprecated Usage

The usage of `$continue` is deprecated. This feature will not be available in releases after Prototype 1.5 in favor of speed. Instead—since your blocks are in fact functions—**simply issue a `return` statement**. This will skip the rest of the block, jumping to the next iteration.

```
['one', 'two', 'three'].each(function(s) {
  alert(s);
});

[ 'hello', 'world'].each(function(s, index) {
  alert(index + ': ' + s);
});
// alerts -> '0: hello' then '1: world'

// This could be done better with an accumulator using inject, but humor me
// here...
var result = [];
$R(1,10).each(function(n) {
  if (0 == n % 2)
    throw $continue;
  if (n > 6)
    throw $break;
  result.push(n);
});
// result -> [1, 3, 5]
```

Example 6.6.

`each` vs. `_each`

If you read the main `Enumerable` page, you may recall that in order for a class to mix in `Enumerable`, it has to provide the fundamental looping code appropriate to its internal structure. This basic iteration method must be called `_each`, and it only receives one argument: the iterator function. You'll find further details on the main page.

Basically, `Enumerable.each` wraps the actual looping code provided by `_each` with:

1. Support for break/continue, as described above.
2. Proper maintenance and passing of the value/index arguments.

Optimized version

There is a very common use-case that will probably be better taken care of by a specialized variant: the method-calling scenario.

Say you want to invoke the same method on all elements, possibly with arguments. You may or may not want to use the result values. This can be achieved easily with `invoke`.

This variant usually performs better than `each`, since it avoids lexical closure costs. However, it does accumulate the result values in an array, which you might not need. At any rate, you might want to benchmark both options in your specific use case.

---

entries

```
entries() -> Array
```

Alias for the more generic `toArray` method.

---

find

```
find(iterator) -> firstElement | undefined
```

Finds the first element for which the iterator returns `true`. Convenience alias for `detect`, but constitutes the preferred (more readable) syntax.

This is the short-circuit version of the full-search `findAll`. It just returns the first element that matches your predicate, or `undefined` if no element matches.

```
// An optimal exact prime detection method, slightly compacted.
function isPrime(n) {
  if (2 > n) return false;
  if (0 == n % 2) return (2 == n);
  for (var index = 3; n / index > index; index += 2)
    if (0 == n % index) return false;
  return true;
```

```
} // isPrime

$R(10,15).find(isPrime)
// -> 11

[ 'hello', 'world', 'this', 'is', 'nice'].find(function(s) {
  return s.length <= 3;
})
// -> 'is'
```

Example 6.7.

findAll

```
findAll(iterator) -> Array
```

Returns all the elements for which the iterator returned `true`. Aliased as `select`.

This is a sort of all-purpose version of `grep` (which is specific to String representations of the values). `findAll` lets you define your predicate for the elements, providing maximum flexibility.

```
$R(1, 10).findAll(function(n) { return 0 == n % 2; })
// -> [2, 4, 6, 8, 10]

[ 'hello', 'world', 'this', 'is', 'nice'].findAll(function(s) {
  return s.length >= 5;
})
// -> ['hello', 'world']
```

Example 6.8.

See also

The `reject` method is the opposite of this one. If you need to split elements in two groups based upon a predicate, look at `partition`.

grep

```
grep(regex[, iterator = Prototype.K]) -> Array
```

Returns all the elements whose string representations match the regular expression. If an iterator is provided, it is used to produce the returned value for each selected element.

This is a variant of `findAll`, which is specific to pattern-matching `String` representations of the elements. It is mostly useful on sequences of `String`s, obviously, but also on any objects with a `toString` method that fits such a usage.

```
// Get all strings with a repeated letter somewhere
['hello', 'world', 'this', 'is', 'cool'].grep(/(.)\1/)
// -> ['hello', 'cool']

// Get all numbers ending with 0 or 5
$R(1,30).grep(/[05]$/)
// -> [5, 10, 15, 20, 25, 30]

// Those, minus 1
$R(1,30).grep(/[05]$/, function(n) { return n - 1; })
// -> [4, 9, 14, 19, 24, 29]
```

Example 6.9.

include

```
include(object) -> Boolean
```

Determines whether a given object is in the `Enumerable` or not, based on the `==` comparison operator. Aliased as `member`.

Note this is not strict equality (`===`, comparing both value and type), but equivalence (just value, with implicit conversions).

If you need to check whether there is an element matching a given predicate, use `any` instead.

```
$R(1,15).include(10)  // -> true

['hello', 'world'].include('HELLO')  // -> false

[1, 2, '3', '4', '5'].include(3)
// -> true (== ignores actual type)
```

Example 6.10.

inject

```
inject(accumulator, iterator) -> accumulatedValue
```

Incrementally builds a result value based on the successive results of the iterator. This can be used for array construction, numerical sums/averages, etc.

```
$R(1,10).inject(0, function(acc, n) { return acc + n; })  // -> 55 (sum of 1 to 10)

$R(2,5).inject(1, function(acc, n) { return acc * n; })  // -> 120 (factorial 5)

['hello', 'world', 'this', 'is', 'nice'].inject([], function(array, value, index) {
  if (0 == index % 2)
    array.push(value);
  return array;
})
// -> ['hello', 'this', 'nice']

// Note how we can use references (see next section):

var array1 = [];
var array2 = [1, 2, 3].inject(array1, function(array, value) {
  array.push(value * value);
  return array;
});
array2  // -> [1, 4, 9]
array1  // -> [1, 4, 9]
array2.push(16);
array1  // -> [1, 4, 9, 16]
```

Example 6.11.

Performance considerations

When injecting on arrays, you can leverage JavaScript's reference-based scheme to avoid creating ever-larger cloned arrays (as opposed to JavaScript's native `concat` method, which returns a new array, guaranteed).

invoke

```
invoke(methodName[, arg...]) -> Array
```

Optimization for a common use-case of `each` or `collect`: invoking the same method, with the same potential arguments, for all the elements. Returns the results of the method calls.

Since it avoids the cost of a lexical closure over an anonymous function (like you would do with `each` or `collect`), this performs much better. Perhaps more importantly, it definitely makes for more concise *and* more readable source code.

```
['hello', 'world', 'cool!'].invoke('toUpperCase')  // -> ['HELLO', 'WORLD', 'COOL!']

['hello', 'world', 'cool!'].invoke('substring', 0, 3)  // => ['hel', 'wor', 'coo']

// Of course, this works on Prototype extensions (why shouldn't it?!)
$('navBar', 'adsBar', 'footer').invoke('hide')
```

Example 6.12.

See also

The `pluck` method does much the same thing for property fetching.

map

```
map(iterator) -> Array
```

Returns the results of applying the iterator to each element. Convenience alias for `collect`.

max

```
max([iterator = Prototype.K]) -> maxValue
```

Returns the maximum element (or element-based computation), or `undefined` if the enumeration is empty. Elements are either compared directly, or by first applying the iterator and comparing returned values.

Note: for equivalent elements, the **latest** one is returned.

```
$R(1,10).max()
// -> 10

['hello', 'world', 'gizmo'].max()
// -> 'world'

function Person(name, age) {
   this.name = name;
   this.age = age;
}

var john = new Person('John', 20);
var mark = new Person('Mark', 35);
var daisy = new Person('Daisy', 22);

[john, mark, daisy].max(function(person) {
  return person.age;
})
// -> 35
```

Example 6.13.

member

```
member(object) -> Boolean
```

Determines whether a given object is in the `Enumerable` or not, based on the == comparison operator. Convenience alias for `include`.

min

```
min([iterator = Prototype.K]) -> minValue
```

Returns the minimum element (or element-based computation), or `undefined` if the enumeration is empty. Elements are either compared directly, or by first applying the iterator and comparing returned values.

Note: for equivalent elements, the **earliest** one is returned.

```
$R(1,10).min()
// -> 1

['hello', 'world', 'gizmo'].min()
// -> 'gizmo'

function Person(name, age) {
   this.name = name;
   this.age = age;
}

var john = new Person('John', 20);
var mark = new Person('Mark', 35);
var daisy = new Person('Daisy', 22);

[john, mark, daisy].min(function(person) {
  return person.age;
})
// -> 20
```

Example 6.14.

partition

```
partition([iterator = Prototype.K]) -> [TrueArray, FalseArray]
```

Partitions the elements in two groups: those regarded as `true`, and those considered `false`. By default, regular JavaScript boolean equivalence is used, but an iterator can be provided, that computes a boolean representation of the elements. This is a preferred solution to using both `findAll/select` and `reject`: it only loops through the elements once!

```
['hello', null, 42, false, true, , 17].partition()
// -> [['hello', 42, true, 17], [null, false, undefined]]

$R(1, 10).partition(function(n) {
  return 0 == n % 2;
})
// -> [[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]]
```

Example 6.15.

pluck

```
pluck(propertyName) -> Array
```

Optimization for a common use-case of `collect`: fetching the same property for all the elements. Returns the property values.

Since it avoids the cost of a lexical closure over an anonymous function (like you would do with `collect`), this performs much better.

Perhaps more importantly, it definitely makes for more concise *and* more readable source code.

```
['hello', 'world', 'this', 'is', 'nice'].pluck('length')
// -> [5, 5, 4, 3, 4]

document.getElementsByClassName('superfluous').pluck('tagName').sort().uniq()
// -> sorted list of unique canonical tag names for elements with this
// specific CSS class...
```

Example 6.16.

See also

The `invoke` method does much the same thing for method invoking.

reject

```
reject(iterator) -> Array
```

Returns all the elements for which the iterator returned `false`.

```
$R(1, 10).reject(function(n) { return 0 == n % 2; })
// -> [1, 3, 5, 7, 9]

[ 'hello', 'world', 'this', 'is', 'nice'].reject(function(s) {
  return s.length >= 5;
})
// -> ['this', 'is', 'nice']
```

Example 6.17.

See also

The `findAll` method (and its `select` alias) are the opposites of this one. If you need to split elements in two groups based upon a predicate, look at `partition`.

select

```
select(iterator) -> Array
```

Alias for the `findAll` method.

size

```
size() -> Number
```

Returns the size of the enumeration.

```
$R(1, 10).size()
// -> 10

['hello', 42, true].size()
// -> 3

$H().size()
// -> 0
```

Example 6.18.

Performance considerations

This method exists solely to provide a generic size-getting behavior for all objects enumerable. The default implementation actually performs the loop, which means it has exact linear complexity. Most objects that

mix in `Enumerable` will try to optimize this by redefining their own version of `size`; this is, for instance, the case of `Array`, which redefines `size` to delegate to arrays' native `length` property.

sortBy

```
sortBy(iterator) -> Array
```

Provides a custom-sorted view of the elements based on the criteria computed, for each element, by the iterator.

Elements of equivalent criterion value are left in existing order. Computed criteria must have well-defined strict weak ordering semantics (i.e. the < operator must exist between any two criteria).

Note that arrays already feature a native `sort` method, which relies on *natural ordering* of the array's elements (i.e. the semantics of the < operator when applied to two such elements). You should use `sortBy` only whe natural ordering is inexistent or otherwise unsatisfactory.

```
['hello', 'world', 'this', 'is', 'nice'].sortBy(function(s) { return s.length; })
// -> 'is', 'this', 'nice', 'hello', 'world']

['hello', 'world', 'this', 'is', 'cool'].sortBy(function(s) {
  var md = s.match(/[aeiouy]/g);
  return null == md ? 0 : md.length;
})
// -> [ 'world', 'this', 'is', 'hello', 'cool'] (sorted by vowel count)
```

Example 6.19.

toArray

```
toArray() -> Array
```

Returns an `Array` representation of the enumeration. Aliased as `entries`.

Note: this makes any object that mixes in `Enumerable` amenable to the `$A` utility function.

```
$R(1, 5).toArray()
// -> [1, 2, 3, 4, 5]
```

Example 6.20.

Performance considerations

Obviously, objects that mix in `Enumerable` may override the default code, as `Array` does.

zip

```
zip(Sequence...[, iterator = Prototype.K]) -> Array
```

Zips together (think of the zip on a pair of trousers) 2+ sequences, providing an array of tuples. Each tuple contains one value per original sequence. Tuples can be converted to something else by applying the optional iterator on them.

For those who never encountered a `zip` function before (i.e. have not worked enough with languages such as Haskell or Ruby ;-)), the exact behavior of this method might be difficult to grasp. Here are a few examples that should clear it up.

```
var firstNames = ['Justin', 'Mislav', 'Tobie', 'Christophe'];
var lastNames = ['Palmer', 'Marohni#', 'Langel', 'Porteneuve'];

firstNames.zip(lastNames)
// -> [['Justin', 'Palmer'], ['Mislav', 'Marohni#'], ['Tobie', 'Langel'], ['Christophe', 'Porteneuve']]

firstNames.zip(lastNames, function(a) { return a.join(' '); })
// -> ['Justin Palmer', 'Mislav Marohni#', 'Tobie Langel', 'Christophe Porteneuve']

var cities = ['Memphis', 'Zagreb', 'Montreal', 'Paris'];
firstNames.zip(lastNames, cities, function(p) {
  return p[0] + ' ' + p[1] + ', ' + p[2];
})
// -> ['Justin Palmer, Memphis', 'Mislav Marohni#, Zagreb', 'Tobie Langel, Montreal', 'Christophe Porteneuve, Paris']

firstNames.zip($R(1, 100), function(a) { return a.reverse().join('. '); })
// -> ['1. Justin', '2. Mislav', '3. Tobie', '4. Christophe']
```

Example 6.21.

Chapter 7

Event

What a wonderful mess (it would be)

Event management is one of the really sore spots of cross-browser scripting.

True, the prominent issue is: everybody does it the W3C way, and MSIE does it another way altogether. But there are quite a few subtler, sneakier issues here and there waiting to bite your ankle, such as the `keypress`/`keydown` issue with KHTML-based browsers (Konqueror and Safari). Also, MSIE has a tendency to leak memory when it comes to discarding event handlers.

Prototype to the rescue!

Of course, Prototype smooths it over so well you'll forget these troubles even exist. Enter the `Event` namespace. It is replete with methods (listed at the top and bottom of this page), that all take the current event object as an argument, and happily produce the information you're requesting, across all major browsers.

`Event` also provides a standardized list of key codes you can use with keyboard-related events. The following constants are defined in the namespace: `KEY_BACKSPACE`, `KEY_TAB`, `KEY_RETURN`, `KEY_ESC`, `KEY_LEFT`, `KEY_UP`, `KEY_RIGHT`, `KEY_DOWN`, `KEY_DELETE`, `KEY_HOME`, `KEY_END`, `KEY_PAGEUP`, `KEY_PAGEDOWN`. The names are self-explanatory.

The functions you're most likely to use a lot are `observe`, `element` and `stop`. As for the others, your mileage may vary; it's all about what your web app does.

element

```
Event.element(event) -> Element
```

Returns the DOM element on which the event occurred.

Note that if the browser does not support *native DOM extensions* (see this page[1] for further details), the element you'll get may very well **not be extended**. If you intend to use methods from `Element.Methods` on it, wrap the call in the `$()` function.

Here's a simple code that lets you click everywhere on the page and, if you click directly on paragraphs, hides them.

```
Event.observe(document.body, 'click', function(event) {
  var elt = Event.element(event);
  if ('P' == elt.tagName)
    $(elt).hide();
});
```

Example 7.1.

See also

There is a subtle distinction between this function and `findElement`.

findElement

```
Event.findElement(event, tagName) -> Element
```

Returns the first DOM element with a given tag name, upwards from the one on which the event occurred.

Sometimes, you're not interested in the actual element that got hit by the event. Sometimes you're interested in its "closest element," (either the original one, or its container, or its container's container, etc.), defined by its tag (e.g., `<p>`). This is what `findElement` is for.

The provided tag name will be compared in a case-insensitive manner.

If no matching element is found, the document itself (`HTMLDocument` node) is returned.

[1] http://http://www.prototypejs.org/learn/extensions

Here's a simple code that lets you click everywhere on the page and hides the closest-fitting paragraph around your click (if any).

```
Event.observe(document.body, 'click', function(event) {
  var elt = Event.findElement(event, 'P');
  if (elt != document)
    $(elt).hide();
});
```

Example 7.2.

For more complex searches, you'll need to get the actual element and use `up` on it, which lets you express your request with CSS syntax, and also search farther than the first match (plus, the result is extension-guaranteed):

```
Event.observe(document.body, 'click', function(event) {
  // First element from event source with 'container' among its CSS classes
  var elt = $(Event.element(event)).up('.container');
  // Or: second DIV from the event source
  // elt = $(Event.element(event)).up('div', 1);
  // Or: second DIV with 'holder' among its CSS classes...
  // elt = $(Event.element(event)).up('div.holder', 1);
  elt.hide();
});
```

Example 7.3.

See also

If you're looking for the element directly hit by the event, just use the `element` function.

isLeftClick

```
Event.isLeftClick(event) -> Boolean
```

Determines whether a button-related mouse event was about the "left" (primary, actually) button.

Note: this is not an absolute left, but "left for default" (right-handed). On systems configured for left-handed users, where the primary button is the right one (from an absolute perspective), this function examines the proper button.

observe

```
Event.observe(element, eventName, handler[, useCapture = false])
```

Registers an event handler on a DOM element.

An important note

First, if you're coming from a background where you'd use HTML event attributes (e.g. `<body onload="return myFunction()">`) or DOM Level-0 event properties (e.g. `window.onload = myFunction;`), you must shed those vile ways :-) and understand what `observe` does.

It does not *replace* existing handlers for that same element+event pair. It *adds* to the *list of handlers* for that pair. Using `observe` will never incapacitate earlier calls.

What are those arguments about?

1. The DOM element you want to observe; as always in Prototype, this can be either an actual DOM reference, or the ID string for the element.
2. The standardized event name, as per the DOM level supported by your browser (usually DOM Level 2 Events[2], see section 1.6 for event names and details). This can be as simple as `'click'`.
3. The handler function. This can be an anonymous function you create on-the-fly, a vanilla function, a bound event listener, it's up to you.
4. Optionally, you can request *capturing* instead of *bubbling*. The details are in the DOM spec referred to above. Note that capturing is not supported on several major browsers, and is seldom what you need, anyway. Most often, you won't even provide this argument.

The requirement people too often forget...

To register a function as an event handler, the DOM element that you want to observe **must already exist** in the DOM (in other words, it must have appeared in the source, or been dynamically created and inserted, before your handler-registration script line runs).

[2] http://www.w3.org/TR/DOM-Level-2-Events/events.html

A simple example

Let us assume the following (X)HTML fragment:

```
<form id="signinForm" method="post" action="/auth/signin">
…
</form>
```

Here's how to register your function `checkForm` on form submission:

```
Event.observe('signinForm', 'submit', checkForm);
```

Example 7.4.

Of course, you'd want this line of code to run once the form exists in the DOM; but putting inline scripts in the document is pretty obstrusive, so instead we'll go for a simple approach that waits till the page is fully loaded:

```
Event.observe(window, 'load', function() {
  Event.observe('signinForm', 'submit', checkForm);
});
```

Example 7.5.

Just a little wrapping…

Note that if your page is heavy, you might want to run this code before the page is fully loaded: just wait until the DOM is loaded, that will be enough. There is currently no standard event for this, but here is a helpful article[3] you can use.

The tricky case of methods that need `this`

Passing your event handler as a function argument, you lose its *binding*. That is, you lose its ability to know what `this` means to the original function. If you're passing in a method that does need to use the `this` reference (for instance, to access fields of its container object), you're in trouble.

Or not.

This is an issue specifically addressed by Prototype's `bindAsEventListener` function. Check it out if you don't know it already. Usage is simple:

[3] http://tanny.ica.com/ica/tko/tkoblog.nsf/dx/domcontentloaded-event-for-browsers

```
var Checks = {
  // some stuff our 'generic' function needs

  generic: function(event) {
    // Some generic, all-purpose checking (e.g. empty required fields)
  }
};

Event.observe('signinForm', 'submit', Checks.generic.bindAsEventListener(Checks));
```

Example 7.6.

See also

The `stopObserving` and `unloadCache` methods are closely related, and worth a look.

pointerX

```
Event.pointerX(event) -> Number
```

Returns the absolute horizontal position for a mouse event.

Note: the position is absolute on the *page*, not on the *viewport*: scrolling right increases the returned value for events on the same viewport location.

pointerY

```
Event.pointerY(event) -> Number
```

Returns the absolute vertical position for a mouse event.

Note: the position is absolute on the *page*, not on the *viewport*: scrolling down increases the returned value for events on the same viewport location.

stop

```
Event.stop(event)
```

Stops the event's propagation and prevents its default action from being triggered eventually.

There are two aspects to how your browser handles an event once it fires up:

- The browser usually triggers event handlers on the actual element the event occurred on, then on its parent element, and so on and so forth, until the document's root element is reached. This is called *event bubbling*, and is the most common form of event propagation. You may very well want to stop this propagation when you just handled an event, and don't want it to keep bubbling up (or see no need for it).
- Once your code got a chance to process the event, the browser handles it as well, if that event has a *default behavior*. For instance, clicking on links navigates to them; submitting forms sends them over to the server side; hitting the Return key in a single-line form field submits it; etc. You may very well want to prevent this default behavior if you do your own handling.

Because stopping one of those aspects means, in 99.9% of the cases, preventing the other one as well, Prototype bundles both in this `stop` function. Calling it on an event object stop propagation *and* prevents the default behavior.

Here's a simple code that prevents a form from being sent to the server side if a certain field is empty.

```
Event.observe('signinForm', 'submit', function(event) {
  var login = $F('login').strip();
  if ('' == login) {
    Event.stop(event);
    // Display the issue one way or another
  }
});
```

Example 7.7.

stopObserving

```
Event.stopObserving(element, eventName, handler[, useCapture = false])
```

Unregisters an event handler.

This function is called with exactly the same argument semantics as `observe`. It unregisters an event handler, so the handler is not called anymore for this element+event pair.

Why won't it stop observing!?

For `stopObserving` to work, you must pass *exactly the same arguments* as those you did to the corresponding `observe` call. Complying with this seems straightforward enough, but there is a common pattern where code is not what it seems to be:

```
var obj = {
  …
  fx: function(event) {  … }
};

Event.observe(elt, 'click', obj.fx.bindAsEventListener(obj));
…

// THIS IS WRONG, DON'T DO IT!
Event.stopObserving(elt, 'click', obj.fx.bindAsEventListener(obj)); // Won't work!
```

Example 7.8.

Here, although it may seem fine at first glance, you must remember that `bindAsEventListener` returns a fresh anonymous function that wraps your method. This means that every call to it returns a new function. Therefore, the code above requests stopping on another function than was used when setting up observation. No match is found, and the original observer is left untroubled.

To avoid this, you need to "cache" the bound functions (which is, for instance, what script.aculo.us [4] does in many of its classes), like this:

```
var obj = {
  …
  fx: function(event) { … }
};
obj.bfx = obj.fx.bindAsEventListener(obj);

Event.observe(elt, 'click', obj.bfx);
…
Event.stopObserving(elt, 'click', obj.bfx);
```

Example 7.9.

See also

The `unloadCache` function is related and worth a look.

[4] http://script.aculo.us

unloadCache

```
Event.unloadCache()
```

Unregisters all event handlers registered through `observe`. Automatically wired for you.

The sad tale of MSIE, event handlers and memory leaks

There is a significant issue with MSIE, which is that under a variety of conditions, it just will not release event handlers when the page unloads. These handlers will stay in RAM, filling it up slowly, clogging the browser's arteries. This is known as a memory leak.

Of course, manually keeping tabs on all the handlers you register (which you do through `observe`, being such a nice person) is pretty tedious. And boring. It would be, in short, the essence of un-cool.

Which is why Prototype takes care of it for you. It keeps tabs, and when `unloadCache` is called, it unregisters everything and frees references, which is akin to sending a big pink lavender-perfumed invitation to the garbage collector.

You don't even need to know this

What's even better is, Prototype automatically hooks `unloadCache` to page unloading, exclusively for MSIE. So you don't have anything to do. It's all taken care of. We just though you'd like to know. Go do something productive, some value-added JavaScript code for instance. We're not here to hinder you with automatable details.

Chapter 8

Form

Form is a namespace and a module for all things form-related, packed with form manipulation and serialization goodness. While it holds methods dealing with forms as whole, its submodule `Form.Element` deals with specific form controls.

Most of these methods are also available directly on FORM elements that have been extended (see "How Prototype extends the DOM"[1]).

disable

```
disable(formElement) -> HTMLFormElement
```

Disables the form as whole. Form controls will be visible but uneditable.

Disabling the form is done by iterating over form elements and disabling them.

Note

Keep in mind that *disabled elements are skipped* by serialize methods! You cannot serialize a disabled form.

[1] http://http://www.prototypejs.org/learn/extensions

enable

```
enable(formElement) -> HTMLFormElement
```

Enables a fully or partially disabled form.

Enabling the form is done by iterating over form elements and enabling them.

See the interactive example in the disable() method, which is basically it.

Note

This will enable all form controls regardless of how they were disabled (by scripting or by HTML attributes).

findFirstElement

```
findFirstElement(formElement) -> HTMLElement
```

Finds first non-hidden, non-disabled form control.

The returned object is either an INPUT, SELECT or TEXTAREA element. This method is used by the `focusFirstElement()` method.

Note

The result of this method is the element that comes first in the *document* order, not the tabindex order[2].

focusFirstElement

```
focusFirstElement(formElement) -> HTMLFormElement
```

Gives keyboard focus to the first element of the form.

Uses `Form.findFirstElement()` to get the first element and calls `activate()` on it. This is useful for enhancing usability on your site by bringing focus on page load to forms such as search forms or contact forms where a user is ready to start typing right away.

[2] http://www.w3.org/TR/html4/interact/forms.html#h-17.11.1

getElements

```
getElements(formElement) -> array
```

Returns a collection of all form controls within a form.

Note

OPTION elements are not included in the result; only their parent SELECT control is.

getInputs

```
getInputs(formElement [, type [, name]]) -> array
```

Returns a collection of all INPUT elements in a form. Use optional `type` and `name` arguments to restrict the search on these attributes.

```
var form = $('myform')

form.getInputs()       // -> all INPUT elements
form.getInputs('text') // -> only text inputs

var buttons = form.getInputs('radio', 'education')
// -> only radio buttons of name "education"

// now disable these radio buttons:
buttons.invoke('disable')
```

Example 8.1.

Note

Input elements are returned in the *document* order, not the tabindex order[3].

[3] http://www.w3.org/TR/html4/interact/forms.html#h-17.11.1

reset

```
reset(formElement) -> HTMLFormElement
```

Resets a form to its default values. Example usage:

```
Form.reset('contact')
$('contact').reset() // equivalent
// both have the same effect as pressing the reset button
```

Example 8.2.

This method allows you to programatically reset a form. It is a wrapper for the `reset()` method native to `HTMLFormElement`.

serialize

```
serialize(formElement[, getHash]) -> String | object
```

Serialize form data to a string suitable for Ajax requests (default behavior) or, if optional `getHash` evaluates to `true`, an object hash where keys are form control names and values are data.

Depending of whether or not the optional parameter `getHash` evaluates to `true`, the result is either an object of the form `{name: "johnny", color: "blue"}` or a string of the form `"name=johnny&color=blue"`, suitable for parameters in an Ajax request. This method mimics the way browsers serialize forms natively so that form data can be sent without refreshing the page.

Deprecated Usage

As of Prototype 1.5 the *preferred* form of passing parameters to an Ajax request is with an *object hash*. This means you should pass `true` for the optional argument. The old behavior (serializing to string) is kept for backwards-compatibility.

The following code is all there is to it:

```
$('person-example').serialize()
// -> 'username=sulien&age=22&hobbies=coding&hobbies=hiking'

$('person-example').serialize(true)
// -> {username: 'sulien', age: '22', hobbies: ['coding', 'hiking']}
```

Example 8.3.

Note

Disabled form elements are not serialized (as per W3C HTML recommendation). Also, file inputs are skipped as they cannot be serialized and sent using only JavaScript.

Keep in mind that "hobbies" multiple select should really be named "hobbies[]" if we're posting to a PHP or Ruby on Rails backend because we want to send an *array* of values instead of a single one. This has nothing to do with JavaScript - Prototype doesn't do any magic with the names of your controls, leaving these decisions entirely up to you.

serializeElements

```
serializeElements(elements) -> string
```

Serialize an array of form elements.

The preferred method to serialize a form is `Form.serialize`. However, with `serializeElements` you can serialize *specific* input elements of your choice, allowing you to specify a subset of form elements that you want to serialize data from.

To serialize all input elements of type "text":

```
Form.serializeElements( $('myform').getInputs('text') )
// -> serialized data
```

Example 8.4.

Chapter

9

Form.Element

This is a collection of methods that assist in dealing with form controls. They provide ways to focus, serialize, disable/enable or extract current value from a specific control.

In Prototype, `Form.Element` is also aliased `Field` and all these methods are available directly on INPUT, SELECT and TEXTAREA elements that have been extended (see "How Prototype extends the DOM"[1]). Therefore, these are equivalent:

```
Form.Element.activate('myfield')
Field.activate('myfield')
$('myfield').activate()
```

Example 9.1.

Naturally, you should always prefer the shortest form suitable in a situation. Most of these methods also return the element itself (as indicated by the return type) for chainability.

activate

```
activate(element) -> HTMLElement
```

Gives focus to a form control and selects its contents if it is a text input.

This method is just a shortcut for focusing and selecting; therefore, these are equivalent (aside from the fact that the former one will **not** return the field) :

[1] http://http://www.prototypejs.org/learn/extensions

```
Form.Element.focus('myelement').select()
$('myelement').activate()
```

Example 9.2.

clear

```
clear(element) -> HTMLElement
```

Clears the contents of a text input.

This code sets up a text field in a way that it clears its contents the first time it receives focus:

```
$('some_field').onfocus = function() {
  // if already cleared, do nothing
  if (this._cleared) return

  // when this code is executed, "this" keyword will in fact be the field itself
  this.clear()
  this._cleared = true
}
```

Example 9.3.

disable

```
disable(element) -> HTMLElement
```

Disables a form control, effectively preventing its value to be changed until it is enabled again.

This method sets the native `disabled` property of an element to `true`. You can use this property to check the state of a control. See the interactive example in the Form.disable() method, which is basically it.

Note

Disabled form controls are never serialized.

Never disable a form control as a security measure without having validation for it server-side. A user with minimal experience of JavaScript can enable these fields on your site easily using any browser. Instead, use disabling as a usability enhancement - with it you can indicate that a specific value should not be changed at the time being.

enable

```
enable(element) -> HTMLElement
```

Enables a previously disabled form control.

See the interactive example in the Form.disable() method, which is basically it.

focus

```
focus(element) -> HTMLElement
```

Gives keyboard focus to an element.

```
Form.Element.focus('searchbox')
// Almost equivalent, but does NOT return the form element
// (uses the native focus() method):
$('searchbox').focus()
```

Example 9.4.

getValue

```
getValue(element) -> string | array
```

Returns the current value of a form control. A string is returned for most controls; only multiple select boxes return an array of values. The global shortcut for this method is `$F()`.

present

```
present(element) -> boolean
```

Returns true if a text input has contents, false otherwise.

```
$('example').onsubmit = function(){
  var valid, msg = $('msg')

  // are both fields present?
  valid = $(this.username).present() && $(this.email).present()

  if (valid) {
    // in real world we would return true here to allow the form to be submitted
    // return true
    msg.update('Passed validation!').style.color = 'green'
```

```
  } else {
    msg.update('Please fill out all the fields.').style.color = 'red'
  }
  return false
}
```

Example 9.5.

select

```
select(element) -> HTMLElement
```

Selects the current text in a text input.

Some search boxes are set up so that they auto-select their content when they receive focus.

```
$('searchbox').onfocus = function() {
  Form.Element.select(this)
  // You can also rely on the native method, but this will NOT return the element!
  this.select()
}
```

Example 9.6.

Focusing + selecting: use `activate`!

The `activate` method is a nifty way to both focus a form field and select its current text, all in one portable JavaScript call.

serialize

```
serialize(element) -> string
```

Creates an URL-encoded string representation of a form control in the `name=value` format. The result of this method is a string suitable for Ajax requests. However, it serializes only a single element - if you need to serialize the whole form use `Form.serialize()` instead.

Note

Serializing a disabled control or a one without a name will always result in an empty string. If you simply need an element's value for reasons other than Ajax requests, use `getValue()`.

Chapter 10

Function

Prototype takes issue with only one aspect of functions: **binding**.

What is binding?

"Binding" basically determines the meaning, when a function runs, of the `this` keyword. While there usually is a proper default binding (`this` refers to whichever object the method is called on), this can be "lost" sometimes, for instance when passing a function reference as an argument.

If you don't know much about the `this` keyword in JavaScript, hop to the docs for the `bind()` method. The examples there will clear it up.

Prototype to the rescue!

Prototype solves this. You'll find two new methods on any function: one that guarantees binding (it can even guarantee early parameters!), and one that is specific to functions intended as event handlers.

bind

```
bind(thisObj[, arg...]) -> Function
```

Wraps the function in another, locking its execution scope to an object specified by `thisObj`.

As discussed on the general `Function` page, binding can be a pretty tricky thing for a newcomer, but it generally is a very simple concept. It requires the basic understanding of the JavaScript language.

In JavaScript, functions are executed in a specific context (often referred to as "scope"). **Inside the function the `this` keyword becomes a reference to that scope.** Since every function is in fact a property of some object—global functions are properties of the `window` object—the execution scope is the object from

which the function was called, or (more precisely) the object that holds a reference to the function:

```
window.name = "the window object"

function scopeTest() {
  return this.name
}

// calling the function in global scope:
scopeTest()
// -> "the window object"

var foo = {
  name: "the foo object!",
  otherScopeTest: function() { return this.name }
}

foo.otherScopeTest()
// -> "the foo object!"
```

Example 10.1.

Because of the dynamic nature of the language, we can't be sure that, for instance, `otherScopeTest()` will always be called on our `foo` object. The reference to it can be copied somewhere else, like on the `window` object:

```
// ... continuing from the last example

// note that we aren't calling the function, we're simply referencing it
window.test = foo.otherScopeTest
// now we are actually calling it:
test()
// -> "the window object"
```

Example 10.2.

The last call demonstrates how the same function can behave differently depending on its execution scope.

When you begin passing around function references in your code, you often want them to become fixated on a specific scope. Prototype can guarantee that your function will execute with the object you want under the `this` keyword just by invoking `bind` on it. You can also save the returned function and use it multiple times if you need so.

The code below is simply proof-of-concept:

```
var obj = {
  name: 'A nice demo',
  fx: function() {
    alert(this.name);
  }
};
```

```
window.name = 'I am such a beautiful window!';

function runFx(f) {
  f();
}

var fx2 = obj.fx.bind(obj);

runFx(obj.fx);
runFx(fx2);
```

Example 10.3.

Now, what few people realize is, `bind` can also be used to prepend arguments to the final argument list:

```
var obj = {
  name: 'A nice demo',
  fx: function() {
    alert(this.name + '\n' + $A(arguments).join(', '));
  }
};

var fx2 = obj.fx.bind(obj, 1, 2, 3);
fx2(4, 5); // Alerts the proper name, then "1, 2, 3, 4, 5"
```

Example 10.4.

bindAsEventListener

```
bindAsEventListener(thisObj[, arg...]) -> Function
```

An event-specific variant of `bind` which makes sure the function will recieve the current event object as the first argument when executing.

If you're unclear on what "binding" is, check out `Function`'s API page. If you don't quite understand what `bind()` does, check out its specific article.

When you're creating methods that you want to use as event handlers, you need to get the current event somehow, as well as control the *context* in which the method will run. `bindAsEventListener` takes care of both, as it binds the handler to the specified context (`thisObj`) and makes sure the event object gets passed to the handler when the event actually occurs.

This method also works around the problem in MSIE when using DOM level 0 style of event handling and the event object *isn't* passed as the first argument, but has to be read from `window.event` instead. You can forget about that with this method as you don't have to do it manually.

You typically use this method in conjunction with `Event.observe`, and anywhere you need to pass a method as an event listener.

Here is a consolidated example:

```
var obj = { name: 'A nice demo' };

function handler(e) {
  var tag = Event.element(e).tagName.toLowerCase();
  var data = $A(arguments);
  data.shift();
  alert(this.name + '\nClick on a ' + tag + '\nOther args: ' + data.join(', '));
}

Event.observe(document.body, 'click', obj.fx.bindAsEventListener(obj, 1, 2, 3));
// Now any click on the page displays obj.name, the lower-cased tag name
// of the clicked element, and "1, 2, 3".
```

Example 10.5.

Chapter 11

Hash

Hash can be thought of as an *associative array*, binding unique keys to values (which are not necessarily unique), though it can not guarantee consistent order its elements when iterating. Because of the nature of JavaScript programming language, every object is in fact a hash; but `Hash` adds a number of methods that let you enumerate keys and values, iterate over key/value pairs, merge two hashes together, encode the hash into a query string representation, etc.

Creating a hash

There are two ways to construct a `Hash` instance: the first is regular JavaScript object instantiation with the `new` keyword, and the second is using the `$H` function. Passing a plain JavaScript object to any of them would clone it, keeping your original object intact, but passing a `Hash` instance to `$H` will return the same instance unchanged. You can call both constructor methods without arguments, too; they will assume an empty hash.

```
var h = $H({ name: 'Prototype', version: 1.5 });
var h = new Hash({ ... });  // equivalent

h.keys().sort().join(', ')
// -> 'name, version'

h.merge({ version: '1.5 final', author: 'Sam Stephenson' });
h.each(function(pair) {
  alert(pair.key + ' = "' + pair.value + '"');
});
// Alerts, in non-guaranteed order:
// 'name = "Prototype"', 'version = "1.5 final"', 'author = "Sam Stephenson"'

$H({ action: 'ship', order_id: 123, fees: ['fee1', 'fee2'] }).toQueryString()
// -> action=ship&order_id=123&fees=fee1&fees=fee2
```

Example 11.1.

Note

`Hash` can not hold *any* key because of having Enumerable mixed in, as well as its own methods. After adding a key that has the same name as any of those methods, this method will no longer be callable. You can get away with doing that to methods you will not need, but imagine the following:

```
var h = new Hash({ ... });
h['each'] = 'my own stuff';
h.map();
// -> errors out because 'each' is not a function
```

Example 11.2.

The most important method in Enumerable is 'each', and—since almost every other method uses it—overwriting it renders our hash instance practically useless. You won't get away with using '_each', too, since it also is an internal Enumerable method.

each

```
each(iterator) -> Hash
```

Iterates over the name/value pairs in the hash.

This is actually the `each` method from the mixed-in `Enumerable` module. It is documented here to illustrate the structure of the passed first argument, and the order of iteration.

Pairs are passed as the first argument of the iterator, in the form of objects with two properties:

1. `key`, which is the key name as a `String`
2. `value`, which is the corresponding value (and can, possibly, be `undefined`)

The order of iteration is browser-dependent, as it relies on the native `for ... in` loop. Although most modern browsers exhibit *ordered* behavior, this may not always be the case, so don't count on it in your scripts.

It is possible to have function values in a hash, though the iteration skips over Hash and Enumerable methods (naturally). More precisely, it skips the properties found on the object's prototype.

```
var h = $H({ version: 1.5, author: 'Sam Stephenson' });
h.each(function(pair) {
  alert(pair.key + ' = "' + pair.value + '"');
});
// Alerts, in non-guaranteed order, ''version = "1.5"' and 'author = "Sam Stephenson"'.
```

Example 11.3.

inspect

```
inspect() -> String
```

Returns the debug-oriented string representation of the hash.

For more information on `inspect` methods, see `Object.inspect`.

```
$H({ name: 'Prototype', version: 1.5 }).inspect()
// -> "<#Hash:{name: 'Prototype', version: 1.5}>" // Order not guaranteed
```

Example 11.4.

keys

```
keys() -> [String...]
```

Provides an Array of keys (that is, property names) for the hash.

Note: the order of key names is browser-dependent (based on the `for…in` loop). Also, this currently skips any property whose value is a function (such as hash methods).

```
$H({ name: 'Prototype', version: 1.5 }).keys().sort()
// -> ['name', 'version']

$H().keys()
// -> []
```

Example 11.5.

merge

```
merge(hash) -> alteredHash
```

Injects all the pairs in the given hash into the current one, which is then returned.

Duplicate keys will cause an overwrite (the argument hash prevails), and new keys from the argument hash are also used. This is useful for selectively overwriting values on specific keys (e.g. exerting some level of control over a series of options).

Note the argument needs not be a `Hash` object, as it it will get passed to the `$H` function anyway, to ensure compatibility.

```
var h = $H({ name: 'Prototype', version: 1.5 });
h.merge({ version: '1.5 final', author: 'Sam' });

h.invoke('join', ' = ').sort().join(', ')
// -> "author = Sam, name = Prototype, version = 1.5 final"
```

Example 11.6.

remove

```
remove(key) -> value
remove(key1, key2...) -> Array
```

Removes keys from a hash and returns their values.

```
var h = new Hash({ a:'apple', b:'banana', c:'coconut' })

h.remove('a', 'c')  // -> ['apple', 'coconut']
h.values()          // -> ['banana']
```

Example 11.7.

toQueryString

```
toQueryString() -> String
```

Turns a hash into its URL-encoded query string representation. This is a form of serialization, and is mostly useful to provide complex parameter sets for stuff such as objects in the Ajax namespace (e.g. `Ajax.Request`).

Undefined-value pairs will be serialized as if empty-valued. Array-valued pairs will get serialized with one name/value pair per array element. All values get URI-encoded using JavaScript's native `encodeURIComponent` function.

The order of pairs in the serialized form is not guaranteed (and mostly irrelevant anyway), except for array-based parts, which are serialized in array order.

```
$H({ action: 'ship', order_id: 123, fees: ['f1', 'f2'], 'label': 'a demo' }).toQueryString()
// -> 'action=ship&order_id=123&fees=f1&fees=f2&label=a%20demo'

$H().toQueryString()  // -> ''
```

Example 11.8.

Note

This method can be called in two ways: as an instance method (as in the above examples) or as a class method on `Hash`.

```
Hash.toQueryString({ foo:'bar' })  // -> 'foo=bar'
```

Example 11.9.

This way you can generate a query string from an object without converting it to a `Hash` instance, making it possible to serialize hashes that have keys corresponding to Enumerable method names.

values

```
values() -> Array
```

Collect the values of a hash and returns them in an array.

The order of values is browser implementation-dependent (based on the `for...in` loop on keys), so—although most of the time you will see it as consistent—it's better not to rely on a specific order. Also remember that the hash may contain values such as `null` or even `undefined`.

```
$H({ name: 'Prototype', version: 1.5 }).values().sort()  // -> [1.5, 'Prototype']
$H().values()  // -> []
```

Example 11.10.

Chapter

12

Insertion

`Insertion` provides a cross-browser solution to the dynamic insertion of HTML snippets (or plain text, obviously). Comes in four flavors: `After`, `Before`, `Bottom` and `Top`, which behave just as expected.

Note that if the inserted HTML contains any `<script>` tag, these will be automatically evaluated after the insertion (`Insertion` internally calls String.evalScripts).

After

```
new Insertion.After(element, html)
```

Inserts the `html` into the page as the next sibling of `element`.

Note that if the inserted HTML contains any `<script>` tag, these will be automatically evaluated after the insertion (`Insertion.After` internally calls String.evalScripts).

Original HTML

```
<div>
    <p id="animal_vegetable_mineral">In short, in all things vegetable, animal, and mineral...</p>
  </div>
```

```
new Insertion.After('animal_vegetable_mineral', "<p>I am the very model of a modern major general.</p>");
```

Example 12.1.

Resulting HTML

```
<div>
    <p id="animal_vegetable_mineral">In short, in all things vegetable, animal, and mineral...</p>
    <p>I am the very model of a modern major general.</p>
  </div>
```

Before

```
new Insertion.Before(element, html)
```

Inserts the `html` into the page as the previous sibling of `element`.

Note that if the inserted HTML contains any `<script>` tag, these will be automatically evaluated after the insertion (`Insertion.Before` internally calls String.evalScripts).

Original HTML

```
<div>
    <p id="modern_major_general">I am the very model of a modern major general.</p>
  </div>
```

```
new Insertion.Before('modern_major_general', "<p>In short, in all things vegetable, animal, and mineral...</p>");
```

Example 12.2.

Resulting HTML

```
<div>
    <p>In short, in all things vegetable, animal, and mineral...</p>
    <p id="modern_major_general">I am the very model of a modern major general.</p>
  </div>
```

Bottom

```
new Insertion.Bottom(element, html)
```

Inserts the `html` into the page as the last child of `element`.

Note that if the inserted HTML contains any `<script>` tag, these will be automatically evaluated after the insertion (`Insertion.Bottom` internally calls String.evalScripts).

Original HTML

```
<div id="modern_major_general">
    <p>In short, in all things vegetable, animal, and mineral...</p>
  </div>
```

```
new Insertion.Bottom('modern_major_general', "<p>I am the very model of a modern major general.</p>");
```

Example 12.3.

Resulting HTML

```
<div id="modern_major_general">
    <p>In short, in all things vegetable, animal, and mineral...</p>
    <p>I am the very model of a modern major general.</p>
  </div>
```

Top

```
new Insertion.Top(element, html)
```

Inserts the `html` into the page as the first child of `element`.

Note that if the inserted HTML contains any `<script>` tag, these will be automatically evaluated after the insertion (`Insertion.Top` internally calls String.evalScripts).

Original HTML

```
<div id="modern_major_general">
    <p>I am the very model of a modern major general.</p>
  </div>
```

```
new Insertion.Top('modern_major_general', "<p>In short, in all things vegetable, animal, and mineral...</p>");
```

Example 12.4.

Resulting HTML

```
<div id="modern_major_general">
    <p>In short, in all things vegetable, animal, and mineral...</p>
    <p>I am the very model of a modern major general.</p>
  </div>
```

Chapter

13

Number

Prototype extends native JavaScript numbers in order to provide:

- `ObjectRange` compatibility, through a `succ` method.
- Ruby-like numerical loops with a `times` method.
- Simple utility methods such as `toColorPart`.

What becomes possible

```
$R(1, 10).each(function(index) {
  // This gets invoked with index from 1 to 10, inclusive
});

(5).times(function(n) {
  // This gets invoked with index from 0 to 5, *exclusive*
  // The parentheses are due to JS syntax, if we did not use a literal, they'd be superfluous
});

128.toColorPart()
// -> '70'

10.toColorPart()
// -> '0a'

'#' + [128, 10, 16].invoke('toColorPart').join('')
// -> '#800a10'
```

Example 13.1.

succ

```
succ() -> Number
```

Returns the successor of the current `Number`, as defined by current + 1. Used to make numbers compatible with `ObjectRange`.

```
(5).succ()  // -> 6
$A($R(1, 5)).join('')  // -> '12345'
```

Example 13.2.

times

```
times(iterator) -> Number
```

Encapsulates a regular [0..n[loop, Ruby-style.

The callback function is invoked with a single argument, ranging from 0 to the number, **exclusive**.

```
var s = '';
(5).times(function(n) {
  s += n;
});

s  // -> '01234'
```

Example 13.3.

toColorPart

```
toColorPart() -> String
```

Produces a 2-digit hexadecimal representation of the number (which is therefore assumed to be in the [0..255] range). Useful for composing CSS color strings.

```
128.toColorPart()  // -> '70'
10.toColorPart()  // -> '0a'
'#' + [128, 10, 16].invoke('toColorPart').join('')  // -> '#800a10'
```

Example 13.4.

Chapter 14

Object

`Object` is used by Prototype as a namespace; that is, it just keeps a few new methods together, which are intended for namespaced access (i.e. starting with "`Object.`").

For the regular developer (who simply uses Prototype without tweaking it), the most commonly used methods are probably `inspect` and, to a lesser degree, `clone`.

Advanced users, who wish to create their own objects like Prototype does, or explore objects as if they were hashes, will turn to `extend`, `keys` and `values`.

clone

```
Object.clone(obj) -> Object
```

Clones the passed object using shallow copy (copies all the original's properties to the result).

Do note that this is shallow copy, not deep copy.

```
var o = { name: 'Prototype', version: 1.5, authors: ['sam', 'contributors'] };
var o2 = Object.clone(o);

o2.version = '1.5 weird';
o2.authors.pop();

o.version
// -> 1.5

o2.version
// -> '1.5 weird'

o.authors
// -> ['sam'] // Ouch!  Shallow copy!
```

Example 14.1.

extend

```
Object.extend(dest, src) -> alteredDest
```

Copies all properties from the source to the destination object. Used by Prototype to simulate inheritance (rather statically) by copying to prototypes.

Documentation should soon become available that describes how Prototype implements OOP, where you will find further details on how Prototype uses `Object.extend` and `Class.create` (something that may well change in version 2.0). It will be linked from here.

Do not mistake this method with its quasi-namesake `Element.extend`, which implements Prototype's (much more complex) DOM extension mechanism.

inspect

```
Object.inspect(obj) -> String
```

Returns the debug-oriented string representation of the object.

- `undefined` and `null` are represented as such.
- Other types are looked up for a `inspect` method: if there is one, it is used, otherwise, it reverts to the `toString` method.

Prototype provides `inspect` methods for many types, both built-in and library-defined, such as in `String`, `Array`, `Enumerable` and `Hash`, which attempt to provide most-useful string representations (from a developer's standpoint) for their respective types.

```
Object.inspect()
// -> 'undefined'

Object.inspect(null)
// -> 'null'

Object.inspect(false)
// -> 'false'

Object.inspect([1, 2, 3])
// -> '[1, 2, 3]'

Object.inspect('hello')
// -> "'hello'"
```

Example 14.2.

keys

```
Object.keys(obj) -> [String...]
```

Treats any object as a `Hash` and fetches the list of its property names.

Note that the order of the resulting Array is browser-dependent (it relies on the `for&#8230;in` loop), and is therefore not guaranteed to follow either declaration or lexicographical order. Sort the array if you wish to guarantee order.

```
Object.keys()
// -> []

Object.keys({ name: 'Prototype', version: 1.5 }).sort()
// -> ['name', 'version']
```

Example 14.3.

values

```
Object.values(obj) -> Array
```

Treats any object as a `Hash` and fetches the list of its property values.

Note that the order of the resulting Array is browser-dependent (it relies on the `for&#8230;in` loop), and is therefore not guaranteed to follow either declaration or lexicographical order. Also, remember that while property names are unique, property values have no constraint whatsoever.

```
Object.values()
// -> []

Object.values({ name: 'Prototype', version: 1.5 }).sort()
// -> [1.5, 'Prototype']
```

Example 14.4.

Chapter

15

ObjectRange

Ranges represent an interval of values. The value type just needs to be "compatible," that is, to implement a `succ` method letting us step from one value to the next (its *successor*).

Prototype provides such a method for `Number` and `String`, but you are of course welcome to implement useful semantics in your own objects, in order to enable ranges based on them.

`ObjectRange` mixes in `Enumerable`, which makes ranges very versatile. It takes care, however, to override the default code for `include`, to achieve better efficiency.

While `ObjectRange` does provide a constructor, the preferred way to obtain a range is to use the `$R` utility function, which is strictly equivalent (only way more concise to use).

The most common use of ranges is, undoubtedly, numerical:

```
$A($R(1, 5)).join(', ')
// -> '1, 2, 3, 4, 5'

$R(1, 5).zip(['one', 'two', 'three', 'four', 'five'], function(tuple) {
  return tuple.join(' = ');
})
// -> ['1 = one', '2 = two', '3 = three', '4 = four', '5 = five']
```

Example 15.1.

Be careful with `String` ranges: as described in its `succ` method, it does not use alphabetical boundaries, but goes all the way through the character table:

```
$A($R('a', 'e'))
// -> ['a', 'b', 'c', 'd', 'e'], no surprise there

$A($R('ax', 'ba'))
// -> Ouch!  Humongous array, starting as ['ax', 'ay', 'az', 'a{', 'a|', 'a}', 'a~'...]
```

Example 15.2.

include

```
include(value) -> Boolean
```

Determines whether the value is included in the range.

This assumes the values in the range have a valid strict weak ordering (have valid semantics for the `<` operator). While `ObjectRange` mixes in `Enumerable`, this method overrides the default version of `include`, and is way more efficient (it uses a maximum of two comparisons).

```
$R(1, 10).include(5)
// -> true

$R('a', 'h').include('x')
// -> false

$R(1, 10).include(10)
// -> true

$R(1, 10, true).include(10)
// -> false
```

Example 15.3.

Chapter

16

PeriodicalExecuter

This is a simple facility for periodical execution of a function. This essentially encapsulates the native `clearInterval/setInterval` mechanism found in native `Window` objects.

The only notable advantage provided by `PeriodicalExecuter` is that it shields you against multiple parallel executions of the callback function, should it take longer than the given interval to execute (it maintains an internal "running" flag, which is shielded against exceptions in the callback function). This is especially useful if you use one to interact with the user at given intervals (e.g. use a `prompt` or `confirm` call): this will avoid multiple message boxes all waiting to be actioned.

Of course, one might very well argue that using an actual object, not needing to maintain a global interval handle, etc. constitute notable advantages as well.

Creating a `PeriodicalExecuter`

The constructor takes two arguments: the callback function, and the interval (in **seconds**) between executions. Once launched, a `PeriodicalExecuter` triggers indefinitely, until the page unloads (which browsers usually take as an opportunity to clear all intervals and timers) or the executer is manually stopped.

```
// Campfire style :-)
new PeriodicalExecuter(pollChatRoom, 3);

new PeriodicalExecuter(function(pe) {
  if (!confirm('Want me to annoy you again later?'))
    pe.stop();
}, 5);
// Note that there won't be a stack of such messages if the user takes too long
// answering to the question...
```

Example 16.1.

stop

```
stop()
```

Stops the periodical executer (there will be no further triggers).

Once a `PeriodicalExecuter` is created, it constitues an infinite loop, triggering at the given interval until the page unloads. This method lets you stop it any time you want.

While there currently is a `registerCallback` method that technically re-enables the executer, it is unclear whether it is considered internal (and therefore should not be used as a feature) or not. In doubt, always instantiate a fresh `PeriodicalExecuter` when you need to start one.

```
var gCallCount = 0;
new PeriodicalExecuter(function(pe) {
  if (++gCallCount > 3)
    pe.stop();
  else
    alert(gCallCount);
}, 1);
// Will only alert 1, 2 and 3, then the PE stops.
```

Example 16.2.

Chapter 17

Position

The `Position` object provides a series of methods that help with element positioning and layout-related issues. These are mainly used by third party UI libraries like script.aculo.us[1].

absolutize

```
absolutize(element)
```

Turns `element` into an absolutely-positioned element *without* changing its position in the page layout.

clone

```
clone(source, target[, options]) -> [Number, Number]
```

Clones the position and/or dimensions of `source` onto `target` as defined by the optional argument `options`.

Note that `target` will be positioned exactly like `source` whether or not it is part of the same CSS containing block[2].

[1] http://script.aculo.us
[2] http://www.w3.org/TR/CSS21/visudet.html#containing-block-details

Options for clone

Name	Default	Description
`setLeft`	`true`	clones `source`'s `left` CSS property onto `target`.
`setTop`	`true`	clones `source`'s `top` CSS property onto `target`.
`setWidth`	`true`	clones `source`'s `width` onto `target`.
`setHeight`	`true`	clones `source`'s `width` onto `target`.
`offsetLeft`	`0`	Number by which to offset `target`'s `left` CSS property.
`offsetTop`	`0`	Number by which to offset `target`'s `top` CSS property.

cumulativeOffset

```
cumulativeOffset(element) -> [Number, Number]
```

Returns the offsets of `element` from the top left corner of the document.

Adds the cumulative `offsetLeft` and `offsetTop` of an element and all its parents.

Note that all values are returned as *numbers only* although they are *expressed in pixels*.

offsetParent

```
offsetParent(element) -> HTMLElement
```

Returns `element`'s closest *positioned* ancestor. If none is found, the `body` element is returned.

The returned element is `element`'s CSS containing block[3].

overlap

```
overlap(mode, element) -> Number
```

Returns a Number between `0` and `1` corresponding to the proportion to which `element` overlaps the point previously defined by Position.within. `mode` can be set to either `vertical` or `horizontal`.

[3] http://www.w3.org/TR/CSS21/visudet.html#containing-block-details

Imagine a block-level `element` (i.e., with dimensions) and a point `x, y` measured in pixels from the top left corner of the page. Calling Position.within will indicate whether that point is within the area occupied by `element`.

Now imagine an element of equal dimensions to `element` with its top left corner at `x, y`. `Position.overlap` indicates the amount these two boxes overlap in either the horizontal or vertical direction.

Note that Position.within must be called right before calling this method.

```
var element = $('some_positioned_element');
Position.cumulativeOffset(element);
// -> [100, 100] (element is 100px from the top and left edges of the page)
element.getDimensions();
// -> { width: 150, height: 150 }

Position.within(element, 175, 145);
// -> true

Position.overlap('horizontal', element);
// -> 0.5 (point is halfway across the element's length)

Position.overlap('vertical', element);
// -> 0.3 (point is 3/10ths of the way across the element's height)
```

Example 17.1.

page

```
page(element) -> [Number, Number]
```

Returns the X/Y coordinates of `element` relative to the *viewport*.

Note that all values are returned as *numbers only* although they are *expressed in pixels*.

positionedOffset

```
positionedOffset(element) -> [Number, Number]
```

Calculates the element's offset relative to its closest positioned ancestor (i.e., the element that would be returned by Position.offsetParent(element).

Calculates the cumulative `offsetLeft` and `offsetTop` of an element and all its parents *until* it reaches an element with a position of `static`.

Note that all values are returned as *numbers only* although they are *expressed in pixels*.

prepare

```
prepare()
```

Calculates document scroll offsets for use with Position.withinIncludingScrollOffsets.

realOffset

```
realOffset(element) -> [Number, Number]
```

Calculates the cumulative scroll offset of an element in nested scrolling containers. Adds the cumulative `scrollLeft` and `scrollTop` of an element and all its parents.

Used for calculating the scroll offset of an element that is in more than one scroll container (e.g., a draggable in a scrolling container which is itself part of a scrolling document).

Note that all values are returned as *numbers only* although they are *expressed in pixels*.

relativize

```
relativize(element)
```

Turns `element` into an relatively-positioned element *without* changing its position in the page layout.

within

```
within(element, x, y) -> Boolean
```

Indicates whether the point `x, y` (measured from the top-left corner of the document) is within the boundaries of `element`. Must be called immediately before Position.overlap.

This function uses `Position.cumulativeOffset` to determine `element`'s offset from the top of the page, then combines those values with `element`'s height and width to identify the offsets of all four corners of the element. It then compares these coordinates to the `x` and `y` arguments, returning `true` if those coordinates fall within the bounding box of `element`.

```
var element = $('some_positioned_element');
Position.cumulativeOffset(element); // -> [100, 100] (100px from left and top)
Element.getDimensions(element);     // -> { width: 150, height: 150 }

Position.within(element, 200, 200); // -> true
Position.within(element, 260, 260); // -> false
```

Example 17.2.

withinIncludingScrolloffsets

```
withinIncludingScrollOffsets(element, x, y) -> Boolean
```

Indicates whether the point `x, y` (measured from the top-left corner of the document) is within the boundaries of `element`. Used instead of Position.within whenever `element` is a child of a scrolling container. Must be called immediately before Position.overlap and immediately after Position.prepare.

This method handles an edge case of `Position.within`: when `element` is the child of a scrolling container. (Scriptaculous, for instance, uses it whenever a Draggable's container is scrollable.) For performance reasons, this method should not be used unless you need this specific edge case.

You *must* call `Position.prepare` first, since it calculates offsets that are used by this method.

Chapter

18

Prototype

The `Prototype` namespace provides fundamental information about the Prototype library you're using, as well as a central repository for default iterators or functions.

We say "namespace," because the `Prototype` object is not intended for instanciation, nor for mixing in other objects. It's really just… a namespace.

Your version of Prototype

Your scripts can check against a particular version of Prototype by examining `Prototype.Version`, which is a version string (e.g. "1.5.0"). The famous script.aculo.us[1] library does this at load time to ensure it's being used with a reasonably recent version of Prototype, for instance.

Browser features

Prototype also provides a (nascent) repository of browser feature information, which it then uses here and there in its source code. The idea is, first, to make Prototype's source code more readable; and second, to centralize whatever scripting trickery might be necessary to detect the browser feature, in order to ease maintenance.

The only currently available feature detection is browser support for DOM Level 3 XPath[2], accessible as a `boolean` at `Prototype.BrowserFeatures.XPath`.

[1] http://script.aculo.us
[2] http://www.w3.org/TR/DOM-Level-3-XPath/xpath.html

Default iterators and functions

Numerous methods in Prototype objects (most notably the `Enumerable` module) let the user pass in a custom iterator, but make it optional by defaulting to an "identity function" (an iterator that just returns its argument, untouched). This is the `Prototype.K` function, which you'll see referred to in many places.

Many methods also take it easy by protecting themselves against missing methods here and there, reverting to empty functions when a supposedly available method is missing. Such a function simply ignores its potential arguments, and does nothing whatsoever (wich is, oddly enough, blazing fast). The quintessential empty function sits, unsurprisingly, at `Prototype.emptyFunction` (note the lowercase first letter).

K

```
K(argument) -> argument
```

`K` is Prototype's very own identity function[3], i.e. it returns its `argument` untouched.

This is used throughout the framework, most notably in the Enumerable module as a default value for iterators.

```
Prototype.K('hello world!');
// -> 'hello world!'

Prototype.K(1.5);
// -> 1.5

Prototype.K(Prototype.K);
// -> Prototype.K
```

Example 18.1.

emptyFunction

```
emptyFunction([argument...])
```

The `emptyFunction` does nothing… and returns nothing!

It is used thoughout the framework to provide a fallback function in order to cut down on conditionals. Typically you'll find it as a default value for optional callback functions.

[3] http://en.wikipedia.org/wiki/Identity_function

Chapter

19

String

Prototype enhances the `String` object with a series of useful methods for `String.prototype` ranging from the trivial to the complex. Tired of stripping trailing whitespaces, try our `String#strip` method. Want to replace `replace` ? Have a look at String#sub and String#gsub. Need to parse a query string ? We have just what you need.

camelize

```
camelize() -> string
```

Converts a string separated by dashes into a camelCase equivalent. For instance, `'foo-bar'` would be converted to `'fooBar'`.

Prototype uses this internally for translating CSS properties into their DOM `style` property equivalents.

```
'background-color'.camelize();  // -> 'backgroundColor'
'-moz-binding'.camelize();  // -> 'MozBinding'
```

Example 19.1.

capitalize

```
capitalize() -> string
```

Capitalizes the first letter of a string and downcases all the others.

```
'hello'.capitalize();  // -> 'Hello'
'HELLO WORLD!'.capitalize();  // -> 'Hello world!'
```

Example 19.2.

dasherize

```
dasherize() -> string
```

Replaces every instance of the underscore character ("_") by a dash ("-").

```
'border_bottom_width'.dasherize();  // -> 'border-bottom-width'
```

Example 19.3.

Note

Used in conjunction with `String.underscore`, `String.dasherize` converts a DOM style into its CSS equivalent.

```
'borderBottomWidth'.underscore().dasherize();  // -> 'border-bottom-width'
```

Example 19.4.

escapeHTML

```
escapeHTML() -> string
```

Converts HTML special characters to their entity equivalents.

```
'<div class="article">This is an article</div>'.escapeHTML();
// -> "&lt;div class="article"&gt;This is an article&lt;/div&gt;"
```

Example 19.5.

evalScripts

```
evalScripts() -> [returnedValue...]
```

Evaluates the content of any `script` block present in the string. Returns an array containing the value returned by each script.

```
'lorem... <script>2 + 2</script>'.evalScripts();
// -> [4]

'<script>2 + 2</script><script>alert("hello world!")</script>'.evalScripts();
// -> [4, undefined] (and displays 'hello world!' in the alert dialog)
```

Example 19.6.

extractScripts

```
extractScripts() -> [script...]
```

Exctracts the content of any `script` block present in the string and returns them as an array of strings.

This method is used internally by `String.evalScripts`. It does *not* evaluate the scripts (use `String.evalScripts` to do that), but can be usefull if you need to evaluate the scripts at a later date.

```
'lorem... <script>2 + 2</script>'.extractScripts();
// -> ['2 + 2']

'<script>2 + 2</script><script>alert("hello world!")</script>'.extractScripts();
// -> ['2 + 2', 'alert("hello world!")']
```

Example 19.7.

To evaluate the scripts later on, you can use the following:

```
var myScripts = '<script>2 + 2</script><script>alert("hello world!")</script>'.extractScripts();
// -> ['2 + 2', 'alert("hello world!")']

var myReturnedValues = myScripts.map(function(script) {
  return eval(script);
});
// -> [4, undefined] (and displays 'hello world!' in the alert dialog)
```

Example 19.8.

gsub

```
gsub(pattern, replacement) -> string
```

Returns the string with *every* occurence of a given pattern replaced by either a regular string, the returned value of a function or a `Template` string. The pattern can be a string or a regular expression.

If its second argument is a string `String.gsub` works just like the native JavaScript method `replace()` set to global match.

```
var mouseEvents = 'click dblclick mousedown mouseup mouseover mousemove mouseout';

mouseEvents.gsub(' ', ', ');
// -> 'click, dblclick, mousedown, mouseup, mouseover, mousemove, mouseout'

mouseEvents.gsub(/\s+/, ', ');
// -> 'click, dblclick, mousedown, mouseup, mouseover, mousemove, mouseout'
```

Example 19.9.

If you pass it a function, it will be invoked for every occurrence of the pattern with the match of the current pattern as its unique argument. Note that this argument is the returned value of the `match()` method called on the current pattern. It is in the form of an array where the first element is the entire match and every subsequent one corresponds to a parenthesis group in the regex.

```
mouseEvents.gsub(/\w+/, function(match){return 'on' + match[0].capitalize()});
// -> 'onClick onDblclick onMousedown onMouseup onMouseover onMousemove onMouseout'

var markdown = '![a pear](/img/pear.jpg) ![an orange](/img/orange.jpg)';

markdown.gsub(/!\[(.*?)\]\((.*?)\)/, function(match){
  return '<img alt="' + match[1] + '" src="' + match[2] + '" />';
});
// -> '<img alt="a pear" src="/img/pear.jpg" /> <img alt="an orange" src="/img/orange.jpg" />'
```

Example 19.10.

Lastly, you can pass `String.gsub` a `Template` string in which you can also access the returned value of the `match()` method using the ruby inspired notation: `#{0}` for the first element of the array, `#{1}` for the second one, and so on. So our last example could be easily re-written as:

```
markdown.gsub(/!\[(.*?)\]\((.*?)\)/, '<img alt="#{1}" src="#{2}" />');
// -> '<img alt="a pear" src="/img/pear.jpg" /> <img alt="an orange" src="/img/orange.jpg" />'
```

Example 19.11.

If you need an equivalent to `String.gsub` but without global match set on, try `String.sub`.

Note

Do *not* use the `"g"` flag on the regex as this will create an infinite loop.

inspect

```
inspect([useDoubleQuotes = false]) -> String
```

Returns a debug-oriented version of the string (i.e. wrapped in single or double quotes, with backslashes and quotes escaped).

For more information on `inspect` methods, see `Object.inspect`.

```
'I\'m so happy.'.inspect();
// -> '\'I\\\'m so happy.\'' (displayed as 'I\'m so happy.' in an alert dialog or the console)

'I\'m so happy.'.inspect(true);
// -> '"I'm so happy."'  (displayed as "I'm so happy." in an alert dialog or the console)
```

Example 19.12.

parseQuery

Alias of `toQueryParams`.

scan

```
scan(pattern, iterator) -> string
```

Allows iterating over every occurrence of the given pattern (which can be a string or a regular expression). Returns the original string.

Internally just calls `String.gsub` passing it `pattern` and `iterator` as arguments.

```
'apple, pear & orange'.scan(/\w+/, alert);
// -> 'apple pear orange' (and displays 'apple', 'pear' and 'orange' in three successive alert dialogs)
```

Example 19.13.

Can be used to populate an array:

```
var fruits = [];
'apple, pear & orange'.scan(/\w+/, function(match){ fruits.push(match[0])});
fruits.inspect()
// -> ['apple', 'pear', 'orange']
```

Example 19.14.

or even to work on the DOM:

```
'failure-message, success-message & spinner'.scan(/(\w|-)+/, Element.toggle)
// -> 'failure-message, success-message & spinner' (and toggles the visibility of each DOM element)
```

Example 19.15.

Note

Do *not* use the `"g"` flag on the regex as this will create an infinite loop.

strip

```
strip() -> string
```

Strips all leading and trailing whitespace from a string.

```
'    hello world!    '.strip();
// -> 'hello world!'
```

Example 19.16.

stripScripts

```
stripScripts() -> string
```

Strips a string of anything that looks like an HTML `script` block.

```
'a <a href="#">link</a><script>alert("hello world!")</script>'.stripScripts();
// -> 'a <a href="#">link</a>'
```

Example 19.17.

stripTags

```
stripTags() -> string
```

Strips a string of any HTML tag.

Watch out for `<script>` tags in your string, as `String.stripTags` will *not* remove their content. Use `String.stripScripts` to do so.

```
'a <a href="#">link</a>'.stripTags();
// -> 'a link'

'a <a href="#">link</a><script>alert("hello world!")</script>'.stripTags();
// -> 'a linkalert("hello world!")'

'a <a href="#">link</a><script>alert("hello world!")</script>'.stripScripts().stripTags();
// -> 'a link'
```

Example 19.18.

sub

```
sub(pattern, replacement[, count = 1]) -> string
```

Returns a string with the *first* `count` occurrences of `pattern` replaced by either a regular string, the returned value of a function or a `Template` string. `pattern` can be a string or a regular expression.

Unlike `String.gsub`, `String.sub` takes a third optional parameter which specifies the number of occurrences of the pattern which will be replaced. If not specified, it will default to 1.

Apart from that, `String.sub` works just like `String.gsub`. Please refer to it for a complete explanation.

```
var fruits = 'apple pear orange';

 fruits.sub(' ', ', ');
// -> 'apple, pear orange'

 fruits.sub(' ', ', ', 1);
// -> 'apple, pear orange'
```

```
 fruits.sub(' ', ', ', 2);
// -> 'apple, pear, orange'

fruits.sub(/\w+/, function(match){return match[0].capitalize() + ','}, 2);
// -> 'Apple, Pear, orange'

var markdown = '![a pear](/img/pear.jpg) ![an orange](/img/orange.jpg)';

markdown.sub(/!\[(.*?)\]\((.*?)\)/, function(match){
  return '<img alt="' + match[1] + '" src="' + match[2] + '" />';
});
// -> '<img alt="a pear" src="/img/pear.jpg" /> ![an orange](/img/orange.jpg)'

markdown.sub(/!\[(.*?)\]\((.*?)\)/, '<img alt="#{1}" src="#{2}" />');
// -> '<img alt="a pear" src="/img/pear.jpg" /> ![an orange](/img/orange.jpg)'
```

Example 19.19.

Note

Do *not* use the `"g"` flag on the regex as this will create an infinite loop.

succ

```
succ() -> string
```

Used internally by `ObjectRange`. Converts the last character of the string to the following character in the Unicode alphabet.

```
'a'.succ();
// -> 'b'
'aaaa'.succ();
// -> 'aaab'
```

Example 19.20.

toArray

```
toArray() -> [character...]
```

Splits the string character-by-character and returns an array with the result.

```
'a'.toArray();
// -> ['a']

'hello world!'.toArray();
// -> ['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '!']
```

Example 19.21.

toQueryParams

```
toQueryParams([separator = '&']) -> Object
```

Parses a URI-like query string and returns an object composed of parameter/value pairs.

This method is realy targeted at parsing query strings (hence the default value of `"&"` for the `separator` argument).

For this reason, it does *not* consider anything that is either before a question mark (which signals the beginning of a query string) or beyond the hash symbol (`"#"`), and runs `decodeURIComponent()` on each parameter/value pair.

`String.toQueryParams` also aggregates the values of identical keys into an array of values.

Note that parameters which do not have a specified value will be set to `undefined`.

```
'section=blog&id=45'.toQueryParams();
// -> {section: 'blog', id: '45'}

'section=blog;id=45'.toQueryParams();
// -> {section: 'blog', id: '45'}

'http://www.example.com?section=blog&id=45#comments'.toQueryParams();
// -> {section: 'blog', id: '45'}

'section=blog&tag=javascript&tag=prototype&tag=doc'.toQueryParams();
// -> {section: 'blog', tag: ['javascript', 'prototype', 'doc']}

'tag=ruby on rails'.toQueryParams();
// -> {tag: 'ruby%20on%20rails'}

'id=45&raw'.toQueryParams();
// -> {id: '45', raw: undefined}
```

Example 19.22.

truncate

```
truncate([length = 30[, suffix = '...']]) -> string
```

Truncates a string to the given length and appends a suffix to it (indicating that it is only an excerpt).

Of course, `String.truncate`does not modify strings which are shorter than the specified length.

If unspecified, the length parameter defaults to 30 and the suffix to `"..."`.

Note that `String.truncate` takes into consideration the length of the appended suffix so as to make the returned string of exactly the specified length.

```
'A random sentence whose length exceeds 30 characters.'.truncate();
// -> 'A random sentence whose len...'

'Some random text'.truncate();
// -> 'Some random text.'

'Some random text'.truncate(10);
// -> 'Some ra...'

'Some random text'.truncate(10, ' [...]');
// -> 'Some [...]'
```

Example 19.23.

underscore

```
underscore() -> string
```

Converts a camelized string into a series of words separated by an underscore (`"_"`).

```
'borderBottomWidth'.underscore();  // -> 'border_bottom_width'
```

Example 19.24.

Used in conjunction with `String#dasherize`, `String#underscore` converts a DOM style into its CSS equivalent.

```
'borderBottomWidth'.underscore().dasherize();  // -> 'border-bottom-width'
```

Example 19.25.

unescapeHTML

```
unescapeHTML() -> string
```

Strips tags and converts the entity forms of special HTML characters to their normal form.

```
'x &gt; 10'.unescapeHTML()
// -> 'x > 10'

'<h1>Pride &amp; Prejudice</h1>'.unescapeHTML()
// -> 'Pride & Prejudice'
```

Example 19.26.

Chapter

20

Template

Any time you have a group of similar objects and you need to produce formatted output for these objects, maybe inside a loop, you typically resort to concatenating string literals with the object's fields. There's nothing wrong with the above approach, except that it is hard to visualize the output immediately just by glancing at the concatenation expression. The Template class provides a much nicer and clearer way of achieving this formatting.

Straight forward templates

The Template class users a basic formatting syntax, similar to what is used in Ruby. The templates are created from strings that have embedded symbols in the form **#{fieldName}** that will be replaced by actual values when the template is applied (evaluated) to an object. A simple example follows.

```
// the template (our formatting expression)
var myTemplate = new Template('The TV show #{title} was created by #{author}.');

// our data to be formatted by the template
var show = {title: 'The Simpsons', author: 'Matt Groening', network: 'FOX' };

// let's format our data
myTemplate.evaluate(show);
// -> The TV show The Simpsons was created by Matt Groening.
```

Example 20.1.

Templates are meant to be reused

As the previous example illustrated, the Template objects are not statically tied to the data. The data is bound to the template only during the evaluation of the template, without affecting the template itself. The next example shows the same template being used with a handful of distinct objects.

```
//creating a few similar objects
var conversion1 = {from: 'meters', to: 'feet', factor: 3.28};
var conversion2 = {from: 'kilojoules', to: 'BTUs', factor: 0.9478};
var conversion3 = {from: 'megabytes', to: 'gigabytes', factor: 1024};

//the template
var templ = new Template('Multiply by #{factor} to convert from #{from} to #{to}.');

//let's format each object
[conversion1, conversion2, conversion3].each( function(conv){
    templ.evaluate(conv);
});
// -> Multiply by 3.28 to convert from meters to feet.
// -> Multiply by 0.9478 to convert from kilojoules to BTUs.
// -> Multiply by 1024 to convert from megabytes to gigabytes.
```

Example 20.2.

Escape sequence

There's always the chance that one day you'll need to have a literal in your template that looks like a symbol, but is not supposed to be replaced. For these situations there's an escape sequence - the backslash character (\ .)

```
// note: you're seeing two backslashes here because the backslash is also a
// escaping character in JavaScript strings
var t = new Template('in #{lang} we also use the \\#{variable} syntax for templates.');
var data = {lang:'Ruby', variable: '(not used)'};
t.evaluate(data);
// -> in Ruby we also use the #{variable} syntax for templates.
```

Example 20.3.

Custom syntaxes

The default syntax of the template strings will probably be enough for most scenarios. In the rare occasion where the default Ruby-like syntax is inadequate there's provision for customization. The Template's constructor accepts an optional second argument that is a regular expression object to match the replaceable

symbols in the template string. Let's put together a template that uses a syntax similar to the ubiquitous **<%= %>** constructs.

```
var syntax = /(^|.|\r|\n)(\<%=\s*(\w+)\s*%\>)/; //matches symbols like '<%= field %>'
var t = new Template('<div>Name: <b><%= name %></b>, Age: <b><%=age%></b></div>', syntax);
t.evaluate( {name: 'John Smith', age: 26} ); // -> <div>Name: <b>John Smith</b>, Age: <b>26</b></div>
```

Example 20.4.

There are important constraints to any custom syntax. Any syntax must provide at least three groupings in the regular expression. The first grouping is to capture what comes before the symbol, to detect the back-slash escape character (no, you cannot use a different character.) The second grouping captures the entire symbol and will be completely replaced upon evaluation. Lastly, the third required grouping captures the name of the field inside the symbol.

evaluate

```
evaluate(object) -> String
```

Applies the template to the given `object`'s data, producing a formatted string with symbols replaced by corresponding `object`'s properties.

```
var hrefTemplate = new Template('/dir/showAll?lang=#{language}&categ=#{category}&lv=#{levels}');
var selection = {category: 'books' , language: 'en-US'};

hrefTemplate.evaluate(selection);
// -> '/dir/showAll?lang=en-US&categ=books&lv='

hrefTemplate.evaluate({language: 'jp', levels: 3, created: '10/12/2005'});
// -> '/dir/showAll?lang=jp&categ=&lv=3'

hrefTemplate.evaluate({});
// -> '/dir/showAll?lang=&categ=&lv='

hrefTemplate.evaluate(null);
// -> error !
```

Example 20.5.

Chapter

21

TimedObserver

An abstract observer class which instances can be used to periodically check some value and trigger a callback when the value has changed. The frequency is in seconds.

A `TimedObserver` object will try to check some value using the `getValue()` instance method which isn't defined in this class. You must use the concrete implementations of `TimedObserver` like `Form.Observer` or `Form.Element.Observer`. The former serializes a form and triggers when the result has changed, while the latter simply triggers when the value of a certain form control changes.

Using `TimedObserver` implementations is straightforward; simply instantiate them with appropriate arguments. For example:

```
new Form.Element.Observer(
  'myelement',
  0.2,  // 200 milliseconds
  function(el, value){
    alert('The form control has changed value to: ' + value)
  }
)
```

Example 21.1.

Now that we have instantiated an object, it will check the value of the form control every 0.2 seconds and alert us of any change. While it is useless to alert the user of his own input (like in the example), we could be doing something useful like updating a certain part of the UI or informing the application on server of stuff happening (over Ajax).

The callback function is always called with 2 arguments: the element given when the observer instance was made and the actual value that has changed and caused the callback to be triggered in the first place.

Form.Element.Observer

```
new Form.Element.Observer(element, frequency, callback)
```

A timed observer for a specific form control.

Form.Element observer implements the `getValue()` method using `Form.Element.getValue()` on the given element. See `Abstract.TimedObserver` for general documentation on timed observers.

Form.Observer

```
new Form.Observer(element, frequency, callback)
```

A timed observer that triggers when any value changes within the form.

Form observer implements the `getValue()` method using `Form.serialize()` on the element from the first argument. See `Abstract.TimedObserver` for general documentation on timed observers.

```
new Form.Observer('example', 0.3, function(form, value){
  $('msg').update('Your preferences have changed. Resubmit to save').style.color = 'red'
  form.down().setStyle({ background:'lemonchiffon', borderColor:'red' })
})

$('example').onsubmit = function() {
  $('msg').update('Preferences saved!').style.color = 'green'
  this.down().setStyle({ background:null, borderColor:null })
  return false
}
```

Example 21.2.