Extracted from:

Prototype and script.aculo.us

You Never Knew JavaScript Could Do This!

Disabled? Let It Show!

A few browsers do not efficiently display disabled elements; for instance, Konqueror and Internet Explorer do not alter the font color of disabled drop-down listboxes or the background color of disabled flat listboxes or multiline text fields (and single-line too, on Internet Explorer). Opera does not alter the background color of checkboxes and radio buttons. Such lackings make it impossible for the user to understand at a glance that part of the UI is off-limits.

This can be fixed quickly with a bit of CSS styling. If you're targeting CSS 2–compatible browsers, just use something like this:

```
*[disabled] { background-color: #ccc; color: gray; }
```

(CSS 3 would even let us use *:disabled, but do you really want to exclude Internet Explorer for another decade?)

If you must support browsers that do not feature CSS 2 attribute selectors, you will need to equip your disabled elements with specific class names, too. This can be done easily with Prototype's magic:

```
$('myForm).select('*:disabled').invoke('addClassName',
  'disabled');
//...and later on, re-enabling...
$('myForm').select('.disabled').invoke('removeClassName',
  'disabled');
```

There is also a common use case where you need to disable an entire form (or enable it). Well, that's a piece of cake—just use the same methods over the form itself, instead of specific elements.

8.2 Looking at Form Fields

```
formElt.getElements() → [ fieldElt, ... ]
formElt.getInputs([typeName] [, name]) → [ fieldElt, ... ]
```

We often need to grab some or all of the elements in a specific form. Even inside Prototype, this is a common task: to disable or enable all the fields, to find the first one, or to serialize the form or determine whether anything has changed since the last time we looked. . . .

The catchall approach is to use getElements(), which returns all fields in the form, in document order. This includes <*input*> tags having

type="hidden", disabled fields, and so on. However, <*button*> fields are omitted; only <*input*>, <*select*>, and <*textarea*> tags are taken into account.

If you need to be more surgical about it (which is a good thing), you can go with getInputs(). This one is designed specifically for those cases where you need to fetch <*input*> fields, usually with a specific type=, name=, or both.

For this second method to work properly, you'd better stick to lowercase (official XHTML) type names in your markup, your DOM generation code, and your getInputs() calls. In the same vein, name filtering is case-sensitive. If you need to filter on name but not on type, simply pass **null** as the first argument. Elements are returned in document order.

Here's a common use case: do you want to check whether any of the checkboxes with name="answer" is, indeed, checked? There you go:

```
if ($('myForm').getInputs('checkbox', 'answer').pluck('checked').any())
  // ...
```

However, if you need more advanced filtering, you'll have a simpler time using methods such as $$() or select(). For instance, assuming your required fields all have *Req* in their id= attributes, you could check (naively, because this relies on empty strings, not blank strings) that they're all filled in like this:

```
if ($('myForm').select('*[name*="Req"]').invoke('present').all())
  // Missing fields!
```

Whatever the method you used, all returned elements are extended for your DOM extension pleasure.

8.3 Submitting Forms Through Ajax

Indeed, Ajax-based form submission is at the heart of the new generation of web applications, so if you haven't got on board yet, it's about time. I'll help you in, don't worry. But before plunging into Ajax (something we'll do in detail in Chapter 9, *Ajax Has Never Been So Easy*, on page 188), we need to consider *what's in a form?*

Shape Shifters: The Changing Nature of Field Values

```
fieldElt.getValue() → value | [ value, ... ]
fieldElt.setValue(value | [ value, ... ]) → HTMLElement
```

Depending on the nature of a field, its value can take one of two forms: a single value (usually a string or, for checkboxes and radio buttons,

a boolean) or an array of values. This second variant happens in only one situation: listboxes with multiple selections enabled (<select multiple=multiple">).

Now for those pesky details that you will wonder about at least once:

- Unchecked radio buttons and checkboxes yield the value **null**. Otherwise, they yield their value property, which is based on their value= attribute (you should *always* specify this attribute, which has no normalized default value).
- Other <*input*> elements yield their value property, based on user interaction (full text contents for text fields, for instance).
- Single-selection listboxes (drop-down or flat) yield the value of the selected option in a DOM-compliant way (Internet Explorer would otherwise fail to use the option's text if no value= attribute were specified).
- Multiple-selection listboxes yield an array of option values, obtained in a DOM-compliant way in the document order of the relevant <*option*> elements.

Version 1.6 introduces the reverse operation, setValue(), which lets you set a field's value using the same value syntax you'd get as a result of getValue(). It relies on the same internal mechanisms, so consistency is guaranteed. This comes in handy when you need to populate a form dynamically (perhaps from JSON data fetched through Ajax).

One last thing: remember the $F() utility function is actually an alias of getValue().

Serializing Fields and Whole Forms

Getting a single field's value in a unified way is nice enough, but most of the time you'll need to take some or all of the fields in a form, mash them together into some reliable string representation, and send that over to the server side.

How Can I Serialize Then?

To serialize, just use a nice method from Prototype, of course. And you have a few to choose from, depending on your use case:

```
fieldElt.serialize() → String
formElt.serialize([options]) → String | hashObj
Form.serializeElements(elements, [options]) → String | hashObj
```

Let's start simple, with the serialization of only one field. All fields feature a serialize() method, which either returns an empty string (if the

Joe Asks...

Why Should I Serialize?

You use serialization when you captured information thanks to a form field (or a whole form) and now need to send this data over the wire in some suitable format. And Prototype's serialization plays nicely with HTTP.

When a form is submitted the regular way, the browser takes care of this for you. When you take over, you're a bit more on your own. Serialization becomes your business, not the browser's anymore.

You can do this from an HTML page in two ways: with a GET request, using URL-encoded parameters right in the URL (as in /myapp/users/list?filter=john&details=yes), or with a POST request, using parameters in the request body. The default format for these is, indeed, the very same URL-encoding you would use in a GET request. It is the format attached to the MIME type *application/x-www-form-urlencoded*, which rules supreme on the Web.

That is why our serialization methods use that format. If you need something else (say, XML or JSON), you can easily grab the form fields you're interested in—which is what we learned to do in the previous section—and cook it up just to your taste (a pinch of basil would be nice). It can be as simple as this:

```
Object.toJSON($(ourForm).serialize(true))
```

Calling serialize(true) returns a hash-like object instead of the default serialized string. The generic Object.toJSON() mechanism will easily process this vanilla object. But for most cases, you should be happy with the default serialization. After all, this is what most server-side technologies natively work with.

field value is **undefined**, which should almost never happen except for files) or returns a *name=value* string, with its two parts properly URL-encoded.

Now, here's the nitty-gritty, which mostly boils down to regular HTML form serialization:

- Values are based on the getValue() method we discussed earlier.
- Fields with **null** values will be handled as if their value were an empty string.
- Fields with undefined values will be serialized with only their name as key (no = sign).
- Fields with array values (that is, multiple-selection listboxes) get serialized as if there were multiple fields with this same name, one per value.

The two other methods are closely related and deal with serializing part or all of a form. Calling serialize() on a form simply forwards to Form.serializeElements() using all elements in the form (except for type="submit" elements, where only one will be used, which is by default the first one).

Most of the time you're happy with a URL-encoded string representation, which is what you get by default. If you pass a hash option with the value **true**, you'll get the resulting hash object (not actually a Hash instance, just a vanilla JavaScript object) back, containing all field names and values, which you can then use to build your own serialization.

By default, if there are multiple type="submit" fields, only the first one in document order will be serialized. You can change that by specifying the value of the name= attribute for the submission field you want to serialize. Just specify it as the submit option.

For instance, the following call:

```
$('myForm').serialize({ hash: true, submit: 'delete' })
```

. . . will return a serialization hash (not a preencoded string), having used the name="delete" submit field instead of the first one in the form.

What About File Fields?

When a form contains a file field (<*input*> with type="file"), traditional serialization cannot happen anymore. Instead of using regular URL-style encoding, the form must be transmitted as *multipart/form-data* and encode the file bytes in a specific MIME part.

Manually creating this multipart encoding would not be difficult, but JavaScript security prevents it from accessing the contents of local files directly (unless you tinker with it, which is beyond most user's abilities or access rights). Because of this, Prototype cannot use actual Ajax for sending local files.

The usual workaround for this is to use a hidden <*iframe*> as the target for the <*form*> and submit the form the regular way. Once the <*iframe*> is done loading the result, we can access it through scripting. This is rather old-school and a bit ugly, but so far this is all we have.

Note than a later version of Prototype may autoswitch to such a technique when you're trying to Ajaxify a form with file fields. In the meantime, you can find detailed walkthroughs for this on many web pages, such as http://www.webtoolkit.info/ajax-file-upload.html (and a Google search on *Ajax file upload* will yield tons of other options).

Streamlining Ajax Forms with request

This chapter is about forms, not Ajax. We will dive into the details (and multiple options) of Ajax processing in Chapter 9, *Ajax Has Never Been So Easy*, on page 188. But there is a form-specific Ajax facility, which we'll look at quickly here. It appeared in Prototype 1.5.1 and aims at streamlining a very common use case: take a regular form, complete with method= and action=, and submit it over Ajax.

```
formElt.request([options]) → Ajax.Request
```

You'll have to refer to Section 9.2, *Options Common to All Ajax Objects*, on page 201 for all the details on the wealth of available options. Just know that this simple call (for example, $('myForm').request()) submits your form through Ajax, using its attributes to determine the HTTP verb

(GET or POST) and the target URL.[2] You can use this to unobtrusively turn your forms over to Ajax when JavaScript is enabled:

```
document.observe('contentloaded', function() {
  $$('form').invoke('observe', 'submit', function(e) {
    e.stop();
    $(Event.element(e)).request();
  });
});
```

This covers a common idiom but is certainly not sufficient for all situations. For more advanced needs, you will have to manually use Ajax.Request and its flock.

8.4 Keeping an Eye on Forms and Fields

In Section 6.3, *Reacting to Form-Related Content Changes*, on page 129, we discovered event-based observers for forms and fields. Whenever an event was triggered to herald a possible value change on a field (or somewhere in a form), these observers would verify that a change had indeed occurred and, if satisfied, would trigger a callback.

Such an approach is not always satisfactory: change-related events trigger late (usually when the field loses focus), too late for some uses (such as autocompletion or on-the-fly validation). Enter time-based observers:

```
new Form.Observer(formElt, interval, callback)
new Field.Observer(fieldElt, interval, callback)
observer.stop()
```

These observers work the same way, but they require an interval or period (expressed in seconds, with fractional numbers allowed), which determines how often they will check up on the data they're observing (field observers check up on a single field, and form observers check up on the whole set of values within the form, relying on serialization for it). Aside from this, the rules do not change. As soon as there actually is a new value (including, obviously, the first time it checks), the callback is triggered.

We used a time-based field observer in our consolidated example in Chapter 7, *Playing with the DOM Is Finally Fun!*, on page 132, in order to enable or disable buttons based on whether a text field was blank.

2. Starting with Prototype 1.6, if <*form*> features no action= attribute or an empty one, the current URL will be used.

Waiting for the field to lose its focus was inadequate. The user might click the enabled button only to find it suddenly disabled because by clicking, the text field would lose its focus. Visual feedback needed to take place earlier and be *live*. The resulting code was confoundingly simple:

```
new Field.Observer('edtName', 0.3, function() {
  $('btnSubmit').disabled = $F('edtName').blank();
});
```

Checking every 0.3" is definitely live enough (the user won't have to consciously wait after they type to see the UI get updated) but large enough an interval not to hog the processor.

Since Prototype 1.6, these observer classes descend from PeriodicalExecuter, so they inherit its stop() method, which lets you put the observer to rest. *1.6*

What We Just Learned

Here are the main take-away points about Prototype's form-related features:

- The features operate at two levels: full forms (the Form namespace) and individual fields (the Field namespace, which is an alias of Form.Element).
- Most methods in these namespaces appear as additional extensions on the relevant DOM elements.
- Most of the API deals with value retrieval and setting, either individually through getValue() and setValue() or at the form's level through such methods as serialize().
- Prototype smooths over cross-browser inconsistencies in field value retrieval (on such issues as no-value-attribute list elements or elements that can be toggled) and provides a powerful value-setting mechanism that helps implement dynamic form filling.
- Form serialization is handy for Ajax submission of the data, but common cases can be automated one step further using the request() method.
- Interval-based observers let us react quickly to changes in a form or individual field to implement dynamic behavior (such as enabling or disabling parts of the UI based on the current form data).

Neuron Workout

- How would you implement an equivalent of Field.Observer using PeriodicalExecuter[3] directly? What about Form.Observer?
- Write a method that takes all the radio buttons with a given field name and toggles their availability (enables or disables them, depending on their state).

3. For details on this class, see Section 10.3, *Periodical Execution Without Risk of Reentrance*, on page 232.

The Pragmatic Bookshelf

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

Visit Us Online

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

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

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

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

Buy the Book

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

Contact Us

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