YUI 3 Cookbook

Evan Goer

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

YUI 3 Cookbook
by Evan Goer

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Mary Treseler
Production Editor: Kristen Borg
Copyeditor: Rachel Monaghan
Proofreader: Kiel Van Horn

Indexer: BIM Indexing
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

May 2012: First Edition.

Revision History for the First Edition:
2012-05-29 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449304195* for release details.

ISBN: 978-1-449-30419-5

[LSI]

1337019481

Table of Contents

CHAPTER 1
Loading Modules

Consider the humble `<script>` element. Introduced in 1995, it is still the gateway for injecting JavaScript into the browser. Unfortunately, if you want to build sophisticated applications, `<script>` shows its age:

- `<script>` conflates the concepts of loading code and executing code. Programmers need fine-grained control over both phases.
- `<script>` is synchronous, blocking the browser's flow/paint cycle until the entire script downloads. This is why performance guides currently recommend moving `<script>` to the bottom of the page. The good news is that HTML now provides the `async` and `defer` attributes, so this issue might improve over time.
- `<script>` has a shared global context with no formal namespacing or security built in. This is bad enough when you're simply trying to protect your own code from your own mistakes, but becomes disastrous when your code must run alongside an unknown number of third-party scripts.
- `<script>` has no information about its relationships with other `<script>` elements. A script might require another script as a dependency, but there is no way to express this. If `<script>` elements are on the page in the wrong order, the application fails.

The root of the problem is that unlike nearly every programming environment on the planet, JavaScript in the browser has no built-in concept of modules (defined in Recipe 1.1). For small scripts, this is not necessarily a big deal. But small scripts have a way of growing into full-fledged applications.

To protect code from interference, many JavaScript libraries use a global object to contain all the library's methods. For example, the hypothetical "Code Ninja" library might instantiate a global object named `NINJA` that supplies methods such as `NINJA.throwShuriken()`. Here, `NINJA` serves as a kind of namespace. This is a reasonable first line of defense.

YUI 3 takes things one step further. There is a global `YUI` object, but you work with this object "inside out." Instead of using `YUI` just as a namespace, you call `YUI().use()` and then write all of your code *inside a callback function nested inside `use()` itself*. Within this scope is a private instance of the library named `Y`, which provides access to YUI methods such as `Y.one()` and objects such as `Y.AutoComplete`.

The disadvantage of YUI 3's approach is that at first glance, it looks profoundly weird.

The advantages of YUI 3's approach are:

- YUI can decouple loading into registration and execution phases. `YUI.add()` registers code as modules with the `YUI` global object, to be loaded on demand. `YUI().use()` provides access to those modules in a safe sandbox.
- YUI can load modules synchronously or asynchronously, since registration is now a separate phase from execution.
- Other than a few static methods, YUI avoids using the shared global context. The `Y` instance that carries the API is private, impossible to overwrite from outside the sandbox.
- YUI supports real dependency logic. When you register modules with `YUI.add()`, you can include metadata about other modules, CSS resources, and more. `YUI().use()` uses this information to build a dependency tree, fetching modules that are needed to complete the tree and skipping modules that are already present. YUI can even load modules conditionally based on browser capabilities. This frees you up to write code optimized for different environments, enabling you to support older, less capable browsers without serving unnecessary code to modern browsers.

Work on YUI's module and loader system began in the middle of 2007, and the system was revamped for the release of YUI 3 in 2009. In the years since, JavaScript modules have quite rightfully become a hot topic. Server-side JavaScript environments now provide native support for the CommonJS module format. The Dojo toolkit has adopted AMD modules as its native format. Future versions of the ECMAScript standard are likely to bake support for modules into JavaScript's core.

As mentioned in the Preface, there are many great JavaScript libraries available, each bringing its own philosophy and strengths. If you are looking for a single feature that captures YUI's design goals, the module system is an excellent place to start. The module system prioritizes code safety and encapsulation. It has intelligent defaults, but it also grants you a tremendous amount of fine-grained control. It works well for small page effects, but it really shines when you're assembling larger applications. You will see these principles expressed time and time again throughout the library.

Because the module and loader system is one of YUI's signature features, this chapter is extensive. If you are just starting out with YUI, you can get away with reading just the first or second recipe, but be sure to return later to learn how to load modules optimally and how to package your own code into modules for later reuse.

Most of the examples in this chapter make some visible change to the page in order to prove that the code works. The typical example uses `Y.one("#demo")` to grab the `<div>` with an `id` of `demo`, followed by `setHTML()` to change the `<div>`'s contents. If you haven't seen YUI's DOM manipulation API in action yet, please peek ahead at Recipes 2.1 and 2.3.

Recipe 1.1 defines the canonical way to load YUI onto the page. This is the most important recipe in the entire book.

Recipe 1.2 describes SimpleYUI, a convenient bundle of DOM manipulation, event façades, UI effects, and Ajax. Using SimpleYUI makes loading YUI more like loading other, more monolithic JavaScript libraries. This is a good alternative place to start if Recipe 1.1 is making your head spin.

Recipe 1.3 explains the concept of loading individual YUI modules, rather than larger rollups. For production-quality code, you can improve performance by identifying and loading only the modules you really need.

Recipe 1.4 introduces the YUI configuration object, which is important for defining your own modules and for gaining fine-grained control over the YUI Loader.

Recipes 1.5 and 1.6 describe loading different categories of modules. Recipe 1.5 explains how to load third-party modules from the YUI gallery, and Recipe 1.6 explains how to incorporate legacy YUI 2 widgets as YUI 3 modules.

Recipe 1.7 explains how to load the YUI core modules from your own servers rather than Yahoo! edge servers. You should strongly consider doing this if you are dealing with private user data over SSL, as loading third-party JavaScript from servers outside your control breaks the SSL security model.

Recipes 1.8, 1.9, 1.10, and 1.11 take you step-by-step through the process of creating your own modules. After Recipe 1.1, these four recipes are the ones that every serious YUI developer should know by heart. Understanding how to create modules is vital for being able to reuse your code effectively.

Recipe 1.12 introduces the `YUI_config` object, which makes it easier to share complex YUI configurations between pages and sites.

Recipe 1.13 demonstrates how to create your own custom rollups, similar to core rollups such as `node` and `io`.

Recipe 1.14 explains how to load jQuery and other third-party libraries into the YUI sandbox as if they were YUI modules. The YUI Loader and module system are flexible enough to wrap and asynchronously load just about anything you might want to use alongside YUI.

The next six recipes discuss more advanced loading scenarios. Recipe 1.15 covers the concept of conditional loading, where YUI fetches a module only if a browser capability

test passes. The YUI core libraries use this powerful technique to patch up old browsers without penalizing modern ones. Recipe 1.16 is a variation of Recipe 1.15 where instead of using conditional loading to patch old browsers, you use it to patch YUI itself.

Recipes 1.17 and 1.18 explain how to load modules in response to user actions, or even in anticipation of user actions. The ability to fetch additional modules after the initial page load provides you with great control over the perceived performance of your application.

Recipe 1.19 explains how to load YUI into an iframe while still maintaining control via the YUI instance in the parent document.

Finally, Recipe 1.20 discusses static loading. By default, YUI modules load asynchronously. Static loading is an advanced technique that trades flexibility and developer convenience for extra performance.

1.1 Loading Rollups and Modules

Problem

You want to load YUI on the page and run some code.

Solution

Load the YUI seed file, *yui-min.js*. Then call `YUI().use()`, passing in the name of a module or rollup you want to load, followed by an anonymous callback function that contains some code that exercises those modules.

Within the callback function, the `Y` object provides the tailored YUI API you just requested. Technically, you can name this object anything you like, but you should stick with the `Y` convention except for rare circumstances, such as Recipe 1.19.

Example 1-1 loads the YUI `Node` API, then uses that API to get a reference to the `<div>` with an `id` of `demo` and set its content. For more information about how to select and modify node instances, refer to Chapter 2.

Example 1-1. Loading the YUI Node API

```
<!DOCTYPE html>
<title>Loading the YUI Node API</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', function (Y) {
    Y.one('#demo').setHTML('Whoa.');
});
</script>
```

In YUI, you do not need to litter your pages with dozens of `<script>` elements. The Loader is specifically designed to kill this antipattern. As a corollary, you should *never* fetch the YUI seed file more than once.

Discussion

`YUI().use()` supports loading both modules and rollups.

A *module* in YUI is a named collection of reusable code. To learn how to create your own modules, start with Recipe 1.8 and related recipes.

A *rollup* is a kind of "supermodule" that represents multiple smaller modules. For example, `node` is a rollup that pulls in `node-base`, `node-style`, and several other modules for manipulating the DOM. Rollups exist for convenience, although sometimes it pays to be more selective and load individual modules, as described in Recipe 1.3.

But how does this even work? The line:

```
YUI().use('foo', function (Y) {...});
```

is pretty mystifying. To break this down step-by-step:

The first `<script>` element in Example 1-1 loads the YUI seed file, which defines the `YUI` global object. `YUI` is not just a namespace object; it is a module registry system. It contains just enough code to bootstrap your way to the rest of the library: some critical YUI utility functions, the Loader code that loads scripts onto the page, and Loader metadata that describes the core YUI modules and their dependencies.

The second `<script>` element calls `YUI().use()`. This call has two stages:

1. Calling `YUI()` creates a new YUI instance. A YUI instance is a host object for assembling a customized YUI API. The instance starts out fairly bare bones—it does not yet provide APIs for doing things like DOM manipulation or Ajax calls.
2. Calling `use()` then augments that instance with additional methods. `use()` takes one or more string parameters representing the names of modules and rollups to load, followed by a callback function (more on that a little later). Somewhat simplified, the `use()` method works in the following manner:
   a. The `use()` method determines which modules it actually needs to fetch. It calculates dependencies and builds a list of modules to load, excluding any modules already loaded and registered with the global `YUI` object.
   b. After resolving dependencies, `use()` constructs a "combo load" URL, and the Loader retrieves all the missing modules from Yahoo's fast edge servers with a single HTTP request. This happens asynchronously so as not to block the UI thread of the browser.
   c. When `use()` finishes loading modules, it decorates the YUI instance with the complete API you requested.

d. Finally, `use()` executes the callback function, passing in the YUI instance as the `Y` argument. Within the callback function, the `Y` object is a private handle to your own customized instance of the YUI library.

In other words, a YUI instance starts out small and relies on `use()` to carefully build up the API you requested. `YUI().use()` automatically handles dependencies and tailors its downloads for the browser you're running in. This is already a huge advantage over downloading libraries as giant monolithic blocks of code.

The `use()` callback function is referred to as the "YUI sandbox." It encapsulates all your code into a private scope, making it impossible for other scripts on the page to accidentally clobber one of your variables or functions. In fact, if you want to run multiple applications on the same page, you can even create multiple independent sandboxes. Once any sandbox loads a module, other sandboxes can use that module without interference and without having to fetch the code again.

Keep in mind that any code you write directly in a `use()` callback function is not actually a module itself, and is therefore not reusable. A `use()` callback should contain only the code required to wire modules into that particular page. Any code that might be reusable, you should bundle into a custom module using `YUI.add()`. For more information, refer to Recipe 1.8.

To improve performance, by default YUI loads the *minified* version of each module. The minified version has been run through YUI Compressor, a utility that shrinks the file size of each module by stripping out whitespace and comments, shortening variable names, and performing various other optimizations described in Recipe 12.12.

As shown in the next section, Recipe 1.2, it is possible to load YUI with the simpler pattern that other libraries use. SimpleYUI is great for learning purposes, but less appropriate for production code.

In addition to the `Y` instance, YUI passes an obscure second parameter to your `use()` callback. This object represents the response from the Loader, and includes a Boolean `success` field, a string `msg` field that holds a success or error message, and a `data` array that lists all modules that successfully loaded. Unfortunately, this reporting mechanism is not 100% reliable in all browsers.

1.2 Loading SimpleYUI

Problem

You want to load YUI onto the page like people loaded JavaScript libraries in the good old days, without all this newfangled module loading and sandboxing nonsense.

Solution

Instead of pointing `<script>` to *yui-min.js*, point it to *simpleyui-min.js*. SimpleYUI includes all modules in YUI's `node`, `event`, `io`, and `transition` rollups, flattened out into a single JavaScript file. These modules are more than enough to create interesting page effects and simple applications.

As shown in Example 1-2, loading SimpleYUI on the page automatically instantiates a global `Y` instance that provides access to the YUI API.

Example 1-2. Loading SimpleYUI

```
<!DOCTYPE html>
<title>Loading SimpleYUI</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/simpleyui/simpleyui-min.js"></script>
<script>
Y.one('#demo').setHTML('This message brought to you by SimpleYUI.');
</script>
```

Discussion

SimpleYUI provides the same functionality you would have received by loading these modules individually, as described in Recipe 1.1. So why use SimpleYUI at all? If you are new to YUI, SimpleYUI acts like jQuery and other popular JavaScript libraries: you simply load a script onto the page and start calling methods from a global object. SimpleYUI is a great way to try out YUI, particularly for people who are still getting used to YUI's idioms.

SimpleYUI is a starter kit that contains DOM, event, and Ajax functionality. However, SimpleYUI is in no way crippled or limited to just these modules; it also includes the Loader, so you are free to call `Y.use()` at any time to pull in additional modules such as `autocomplete` or `model`. For an example of calling `Y.use()` from within `YUI().use()`, refer to Example 1-22.

The disadvantages of using SimpleYUI are that it pulls in code that you might not need, and that it lacks a sandbox. You can address the latter issue by wrapping your code in an anonymous function and then immediately executing that function, as shown in Example 1-3.

Example 1-3. Loading SimpleYUI in a generic sandbox

```
<!DOCTYPE html>
<title>Loading SimpleYUI in a generic sandbox</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/simpleyui/simpleyui-min.js"></script>
```

```
<script>
var message = 'BOGUS MESSAGE';

(function () {
    var message = 'This message brought to you by sandboxed SimpleYUI.';
    Y.one('#demo').setHTML(message);
}());
</script>
```

JavaScript's scoping rules ensure that variables outside the function can be referenced from within the function. However, any variables redeclared inside the function will trump any values declared outside. Or, looking at this the other way around, code outside the sandbox cannot overwrite private variables inside the sandbox.

Experienced JavaScript developers often use this kind of generic sandbox with other libraries. It is a fine defensive pattern in general, but less common in YUI simply because the standard loading pattern shown in Example 1-1 provides a sandbox already.

If you search the Web, you'll find a popular alternative pattern that works just as well, but is a little less aesthetically pleasing:

```
(function(){})()
```

Yahoo! JavaScript architect Douglas Crockford refers to this as the "dogballs" pattern.

Strictly speaking, you don't need to resort to SimpleYUI to get a global `Y` object. `YUI().use()` returns a `Y` instance, so you can always do:

```
var Y = YUI().use(...);
```

In any case, these caveats about performance and sandboxing might not be important to you, depending on your situation. Some engineering groups use SimpleYUI as a way to segment different projects: critical pages and core pieces of infrastructure use the YUI sandbox, while prototypes and temporary marketing pages use SimpleYUI to make life easier for designers and prototypers. SimpleYUI is also a good tool for developers who are starting to transition code into the YUI "inside-out" sandbox pattern. Projects in transition can load SimpleYUI and leverage those APIs in existing legacy code, rather than having to immediately migrate large amounts of legacy JavaScript into YUI modules.

1.3 Identifying and Loading Individual Modules

Problem

You want to load the smallest possible amount of code necessary to accomplish a given task.

Solution

The YUI API documentation indicates which modules supply which individual methods and properties. As you write your code, consult the documentation and include only the specific modules you need in your `YUI().use()` call, in order to avoid loading code that contains unnecessary functionality.

Example 1-4 illustrates loading smaller, focused modules instead of larger rollups. As mentioned in Recipe 1.1, YUI passes a second parameter to the `use()` callback that represents the response from the Loader. Example 1-4 converts this object into a string with `Y.JSON.stringify()`, using `stringify()`'s extended signature to pretty-print the output, and then displays the string by inserting it into a `<pre>` element. You could do all of this by loading the `node` and `json` rollups, but it turns out that the script only really requires the smaller modules `node-base` and `json-stringify`.

Example 1-4. Using individual modules

```
<!DOCTYPE html>
<title>Using individual modules</title>

<pre id="demo"></pre>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('json-stringify', 'node-base', function (Y, loaderResponse) {
    var pre = Y.one('#demo');
    pre.set('text', Y.JSON.stringify(loaderResponse, null, 4));
});
</script>
```

The example uses `set('text')` rather than `setHTML()`. Methods like `setHTML()` and `set('innerHTML')` are insecure when used for non-HTML strings or strings whose actual content or origin is unknown.

Discussion

YUI is broken into small modules that enable you to define very tight sets of dependencies. For convenience, YUI users often load rollups, which represent a group of related modules. For example, the `node` rollup is an alias for loading a list of modules that includes `node-base`, `node-style`, `node-event-delegate`, and `nodelist`.

Likewise, the `json` rollup includes `json-parse` and `json-stringify`, on the assumption that most applications that work with JSON need to convert JSON in both directions. However, if your application only needs to convert objects into strings, you can load `json-stringify` and avoid loading deadweight code from `json-parse`.

If you understand exactly which modules your implementation needs, you can save bytes by loading just those modules instead of loading rollups. However, this does

require checking the YUI API documentation carefully for which methods and properties come from which modules, so that you're not caught off-guard by "missing" features.

One option is to use rollups when prototyping and developing, then replace them with a narrower list of modules when you are getting ready to release to production. The YUI Configurator is a handy tool for determining an exact list of dependencies. If you take this approach, be sure to have a test suite in place to verify that your application still works after narrowing down your requirements. For more information about testing YUI, refer to Chapter 12.

See Also

Recipe 1.13; the YUI Configurator (*http://yuilibrary.com/yui/configurator/*); the YUI JSON User Guide (*http://yuilibrary.com/yui/docs/json/*).

1.4 Loading a Different Default Skin

Problem

You want the Loader to load the `"night"` skin for all YUI widgets—a darker CSS skin that is designed to match themes that are popular on mobile devices.

Solution

Pass in a YUI configuration object that includes a `skin` property with an alternative `defaultSkin` name. Some modules provide one or more named CSS skins. By default, when the Loader loads a module with a skin, the Loader attempts to fetch the module's `"sam"` skin file. However, if you are loading modules that happen to have multiple skins, you can instruct the Loader to fetch a different skin across the board.

Example 1-5 loads and instantiates a `Calendar` widget with its alternative, darker `"night"` skin. By convention, all YUI skin styles are scoped within a class name of `yui3-skin-`*`skinname`*. This means that to actually *apply* the `night` skin once it has loaded on the page, you must add the class `yui3-skin-night` to the `<body>` or to a containing `<div>`.

Example 1-5. Changing YUI's default skin

```
<!DOCTYPE html>
<title>Changing YUI's default skin</title>

<div id="demo" class="yui3-skin-night"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI({
    skin: { defaultSkin: 'night' }
```

```
}).use('calendar', function (Y) {
    new Y.Calendar({ width: 300 }).render('#demo');
});
</script>
```

Discussion

YUI offers a great variety of configuration options that control the behavior of the Loader and certain properties of the YUI sandbox. For example, to prevent the Loader from dynamically loading any CSS, you can pass in a `fetchCSS: false`. This setting is useful if you plan to manually add all YUI CSS resources as static `<link>` elements, and you don't want the Loader to fetch the same CSS resources twice.

One of the most important use cases is configuring metadata for custom modules. The Loader already has metadata for core YUI modules included in the seed file, but to properly load any modules you have created, you must provide the Loader with your module names, dependencies, and more. For recipes that demonstrate how to do this, refer to Recipes 1.10 and 1.11.

See Also

More information about skins and loading CSS in Recipes 7.9 and 7.10; a variety of `Slider` skins shown side by side (*http://yuilibrary.com/yui/docs/slider/slider-skin.html*); the `YUI` Global Object User Guide (*http://yuilibrary.com/yui/docs/yui/*); YUI `config` API documentation (*http://yuilibrary.com/yui/docs/api/classes/config.html*); YUI Loader API documentation (*http://yuilibrary.com/yui/docs/api/classes/Loader.html*).

1.5 Loading Gallery Modules

Problem

You want to load a useful third-party module from the YUI gallery and use it alongside core YUI modules.

Solution

Load the gallery module from the Yahoo! content delivery network (CDN) with `YUI().use()` as you would with any other YUI module. Gallery module names all start with the prefix `gallery-`. Once loaded, gallery modules attach to the `Y` just like core YUI modules.

Example 1-6 loads the To Relative Time gallery module, which adds a `toRelativeTime()` method. This method converts `Date` objects to English strings that express a relative time value, such as `"3 hours ago"`.

To ensure that the example loads a specific snapshot of the gallery, the YUI configuration specifies a gallery build tag. For more information, refer to the Discussion.

Example 1-6. Using the To Relative Time gallery module with YUI Node

```
<!DOCTYPE html>
<title>Using the "To Relative Time" gallery module with YUI Node</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI({
    gallery: 'gallery-2010.08.25-19-45'
}).use('gallery-torelativetime', 'node', function (Y) {
    var entryTime = new Date(2011,10,1);
    Y.one('#demo').setHTML(Y.toRelativeTime(entryTime));
});
</script>
```

Discussion

The YUI gallery is a repository for sharing third-party modules. Modules in the gallery range from tiny standalone utilities to large families of related components.

YUI contributors can choose to serve their gallery modules from the Yahoo! CDN. Developers who want to take advantage of this feature must:

- Sign and submit a YUI Contributor License Agreement (CLA)
- Release their code under the open source BSD license, the same license YUI uses
- Host their source code on GitHub, the same repository where YUI is hosted

Some gallery modules have not gone through these steps and so are not served from the Yahoo! CDN. You can use non-CDN gallery modules by downloading and installing them on your own server. For more information about hosting modules locally, refer to Recipe 1.7.

The main difference between gallery modules and the core modules is that for the core modules, the YUI engineering team is fully responsible for fixing bugs, reviewing code, and testing changes. Gallery modules have whatever level of support the module's owner is willing to provide.

Updates to gallery modules get picked up on the CDN when the YUI team pushes out the gallery build, which occurs roughly every week. Each gallery build has a build tag, such as `gallery-2011.05.04-20-03`. If you omit the `gallery` configuration option, YUI falls back to loading a default gallery build tag associated with the particular version of core YUI you are using. Thus, the following code works:

```
YUI().use('gallery-torelativetime', 'node', function (Y) {
    var entryTime = new Date(2011,10,1);
    Y.one('#demo').setHTML(Y.toRelativeTime(entryTime));
});
```

However, it is better to declare an explicit, tested gallery build tag. Otherwise, upgrading your YUI version later on will silently change the gallery tag, which might not be what you want.

For gallery modules served from the Yahoo! CDN, the YUI engineering team lightly examines code changes for serious security issues (such as blatant malware) and glaring bugs. Beyond that, there is no guarantee of code quality. Non-CDN gallery modules are completely unreviewed. Before using any gallery module, be sure to carefully evaluate the module's functionality, source code, and license for yourself.

See Also

The YUI gallery (*http://yuilibrary.com/gallery/*); Luke Smith's To Relative Time gallery module (*http://yuilibrary.com/gallery/show/torelativetime*); the tutorial "Contribute Code to the YUI Gallery" (*http://yuilibrary.com/yui/docs/tutorials/gallery/*).

1.6 Loading a YUI 2 Widget

Problem

You want to use one of your favorite widgets from YUI 2, but it hasn't been ported over to YUI 3 yet.

Solution

Load the widget as a YUI 3 module using its YUI 2in3 wrappers, as shown in Example 1-7.

Example 1-7. Loading a YUI 2 TreeView in YUI 3

```
<!DOCTYPE html>
<title>Loading a YUI 2 TreeView in YUI 3</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('yui2-treeview', function (Y) {
    var YAHOO = Y.YUI2,
        tree = new YAHOO.widget.TreeView('demo', [
            {
                label: 'hats',
                children: [
                    { label: 'bowler' },
                    { label: 'fedora' }
                ]
            },
            {
                label: 'gloves'
            }
```

```
        ]);
    tree.render();
});
</script>
```

Discussion

With YUI 2in3, core YUI 2 widgets such as `ImageCropper`, `ColorPicker`, and `ProgressBar` are represented as first-class YUI 3 modules. Any YUI 2 widget you load this way attaches to the `Y` object as `Y.YUI2`. To make this look more like classic YUI 2–style code, you can rename `Y.YUI2` to `YAHOO`, as shown in Example 1-7.

Although you may freely intermix YUI 3 code with YUI 2 wrapped modules, keep in mind that just because it loads like YUI 3 doesn't mean it behaves like YUI 3. For example, new YUI 2 widgets take their container `<div>`'s `id` as a string, as in `'demo'`. For YUI 3 widgets, you pass in the *CSS selector* for the `<div>`, as in `'#demo'`.

By default, the version of YUI 2 you get is version 2.8.2. However, you can retrieve any previous version by setting the `yui2` field in the YUI object config:

```
YUI({ yui2: '2.7.0' }).use('yui2-treeview', function (Y) {
...
});
```

To load the absolute latest and greatest (and final!) version of YUI 2, use:

```
YUI({
    'yui2': '2.9.0',
    '2in3': '4'
}).use('yui2-treeview', function (Y) {
...
});
```

The `2in3` property configures the version of the YUI 2in3 wrapper to use, which must be at version 4 to load version 2.9.0.

See Also

YUI 2in3 project source (*https://github.com/yui/2in3/tree/master/dist/2.9.0/build*); YUI 2 `TreeView` documentation (*http://developer.yahoo.com/yui/treeview/*).

1.7 Loading Locally Hosted Builds

Problem

You want to load YUI from your own servers instead of from Yahoo! servers.

Solution

By default, the `YUI` object is configured to fetch from Yahoo! servers. You can change this by:

1. Downloading the latest stable YUI SDK zip file from yuilibrary.com.
2. Unzipping the zip file in some directory under your web server's web root.
3. Creating a `<script>` element that points to the *yui-min.js* file.

For example, if you unzipped the SDK under the top level directory */js* and pointed the first `<script>` element's `src` at the local seed file (as shown in Example 1-8), this automatically configures YUI to load all YUI core modules locally. This also disables combo loading (discussed shortly).

Example 1-8. Loading a local copy of YUI

```
<!DOCTYPE html>
<title>Loading a local copy of YUI</title>

<div id="demo"></div>

<script src="/js/yui/build/yui/yui-min.js"></script>
<script>
YUI().use('node', function (Y) {
    Y.one('#demo').setHTML('All politics is local.');
});
</script>
```

To verify that YUI is loading from your own site rather than `yui.yahooapis.com`, use your browser's component inspector (such as Firefox's Web Inspector pane or Chrome's Developer Tools pane).

Discussion

Yahoo! maintains a distributed collection of servers known as a *content delivery network* (CDN). A CDN is designed to serve files from systems that are physically close to the user who made the request. By default, YUI uses the Yahoo! CDN, which grants all YUI users free access to the same network that runs Yahoo's own high-traffic sites. This saves your own bandwidth, reduces your own server load, and greatly improves performance thanks to browser caching and improved geographical distribution.

However, there are plenty of reasons to go it alone. Perhaps your organization forbids loading resources from remote servers as a matter of policy. Or perhaps your pages use SSL, in which case loading remote resources is a bad idea, as it exposes your users' secure information to the remote site. In these cases, you can serve YUI from your own server.

Each release of YUI provides a full developer kit for download under *http://yuilibrary.com/downloads/*. The zip file contains the library, API documentation, and example files.

If you want the latest-and-greatest version of YUI's source, you can check it out by running:

```
git clone https://github.com/yui/yui3.git
```

For more information about how to send code to the upstream YUI project, refer to the tutorial "Contribute Code to YUI" (*http://yuilibrary.com/yui/docs/tutorials/contribute/*).

Download the zip file, unzip it into your preferred location under your web server's root, and then reference the local YUI seed file in your web page:

```
<script src="path/yui/yui-min.js"></script>
```

where *path* is the path under the web root in which the YUI module directories reside, such as */js/yui/build*. In addition to the core YUI 3 SDK, you can also download and serve up the latest build of the YUI gallery and the YUI 2in3 project from your own server.

Loading a local YUI seed file automatically reconfigures the Loader to work with local files. Under the covers, this is like instantiating a sandbox with a configuration of:

```
YUI({
    base: '/js/yui/build/',
    combine: false
}).use('node', function (Y) {
    Y.one('#demo').setHTML('All politics is local.');
});
```

The `base` field defines the server name and base filepath on the server for finding YUI modules. By default, this is `http://yui.yahooapis.com/version/build`. For alternative seed files, YUI inspects your seed file URL and resets `base` appropriately. This means you rarely have to set `base` yourself, at least at the top level. Sometimes you might need to override `base` within a module group, as described in Recipe 1.11.

The `combine` field selects whether YUI attempts to fetch all modules in one "combo load" HTTP request. A *combo loader* is a server-side script designed to accept a single HTTP request that represents a list of modules, decompose the request, and concatenate all the requested JavaScript into a single response.

Loading a seed file from `yui.yahooapis.com` sets the `combine` field to `true`. For seed files loaded from unknown domains, YUI changes `combine` to `false`, on the assumption that a random server does not have a combo loader installed. Setting `combine` to `false` is a safety measure that ensures that local installations of YUI "just work," at the cost of generating lots of HTTP requests. To set up a production-quality local YUI installation,

you should install your own local combo loader and set `combine` back to `true`. Implementations are available for a variety of server environments:

- PHP Combo Loader (*http://yuilibrary.com/projects/phploader/*), the reference implementation, written by the YUI team. Old and stable, but not under active development.
- Node.js Combo Loader (*https://github.com/rgrove/combohandler*), written and maintained by Ryan Grove.
- Perl Combo Loader (*https://github.com/brianjmiller/cgi-combo*), written and maintained by Brian Miller.
- ASP.NET Combo Loader (*https://github.com/gmoothart/NCombo*), written and maintained by Gabe Moothart.
- Python/WSGI Combo Loader (*https://github.com/chrisgeo/comboloader*), written and maintained by Chris George.
- Ruby on Rails Combo Loader (*https://github.com/sjungling/rails-yui_loader/*), written and maintained by Scott Jungling.

To install and operate a particular combo loader, refer to that combo loader's documentation.

See Also

YUI 3 SDK downloads (*http://yuilibrary.com/downloads/#yui3*); Brian Miller's article on locally served YUI3 (*http://blog.endpoint.com/2011/02/locally-served-yui3.html*), which includes a configuration for serving up local copies of the gallery and YUI 2in3.

1.8 Creating Your Own Modules

Problem

You want to bundle and reuse your own code as a YUI module.

Solution

Use `YUI.add()` to register your code as a module with the `YUI` global object. At minimum, `YUI.add()` takes:

- A name for your module. By convention, YUI module names are lowercase and use hyphens to separate words.
- A callback function that defines your actual module code. To expose a property or function in the module's public interface, you attach the component to the `Y` object.

Once `YUI.add()` executes, you can use your code like any other YUI module. In Example 1-9, `YUI().use()` immediately follows the module definition, loading the modules it needs and then executing module methods in a callback function.

Example 1-9. Creating and using a Hello World module

```
<!DOCTYPE html>
<title>Creating and using a Hello World module</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI.add('hello', function (Y) {
    Y.namespace('Hello');

    Y.Hello.sayHello = function () {
        return 'GREETINGS PROGRAMS';
    };
});

YUI().use('node-base', 'hello', function (Y) {
    Y.one('#demo').setHTML(Y.Hello.sayHello());
});
</script>
```

To help avoid naming collisions, you can use `Y.namespace()` to manufacture a `Hello` namespace for the `sayHello()` method. `Y.namespace()` is a handy utility, though in this simple example, the call is essentially equivalent to:

```
Y.Hello = {};
```

Example 1-9 represents only the most basic building block for creating modules. This example is not enough to create truly reusable code. Real-world modules declare dependencies and other metadata, and are defined in a separate file from where they are used. For more information, refer to Recipes 1.9 and 1.10.

Discussion

As mentioned in the introduction and in Recipe 1.1, YUI separates module *registration* from module *execution*. `YUI.add()` registers modules with the `YUI` global object, while `YUI().use()` attaches modules to a `Y` instance so that you can execute the module's functions. `YUI.add()` and `YUI().use()` are designed to work together; first you register some code, and then later you retrieve and execute it.

When designing your applications, always think about how to move as much code as possible out of `use()` and into `add()`. Code in an `add()` callback is reusable, while code in the `use()` callback is unreusable "glue" code designed to wire an application into a particular page.

If you compare `YUI().use()` and `YUI.add()` closely, you might notice the lack of parentheses on the `YUI` for `YUI.add()`. This is a key distinction:

- `YUI.add()` is a static method that registers module code with the `YUI` global object.
- `YUI().use()` is a factory method that creates YUI instances with the given configuration.

The `YUI` global object stores a common pool of available code. The `Y` object holds the particular subset of code that you want to actually register in a `YUI.add()` or use in a `YUI().use()`. Again, the name `Y` is just a strong convention. Within a sandbox, you can name the instance anything you like, but you should do this only if you are creating nested `use()` sandboxes, or if you need to inform other developers that this instance is "weird" in some way. For an example, refer to Recipe 1.19.

The heart of `YUI.add()` is the callback function that defines your module code. Any functions or objects that you attach to the `Y` in the `add()` callback function become available later on in the `use()` callback function. Anything you do not attach to the `Y` remains private. For an example of a private function in a module, refer to Example 1-10.

When attaching functions and objects, consider using a namespace rather than attaching directly to the `Y`, as this space is reserved for a small number of core YUI methods. You can either add namespaces manually by creating empty objects, or call the `Y.namespace()` utility method. `Y.namespace()` takes one or more strings and creates corresponding namespaces on the `Y` object. Any namespaces that already exist do not get overwritten. `Y.namespace()` is convenient for creating multiple namespaces at once and for creating nested namespaces such as `Y.Example.Hello`. `Y.namespace()` also returns the last namespace specified, so you can use it inline:

```
Y.namespace('Hello').sayHello = function () { ...
```

You might be wondering about the YUI core modules—do they use `YUI.add()`? In fact, YUI core modules all get wrapped in a `YUI.add()` at build time, thanks to the YUI Builder tool. If you download and unzip the YUI SDK, you will find the raw, unwrapped source files under the */src* directory, and the wrapped module files under the */build* directory. In other words, there's no magic here—the core YUI modules all register themselves with the same interface as your own modules.

See Also

Instructions for using YUI Builder (*http://yuilibrary.com/projects/builder*).

1.9 Creating a Module with Dependencies

Problem

You want to create a custom YUI module and ensure that it pulls in another YUI module as a dependency.

Solution

Use `YUI.add()` to register your code as a module with the `YUI` global object, and pass in a configuration object that includes your module's dependencies. After the module name and definition, `YUI.add()` takes two optional parameters:

- A string version number for your module. This is the version of your module, not the version of YUI your module is compatible with.
- A configuration object containing metadata about the module. By far the most common field in this configuration object is the `requires` array, which lists your module's dependencies. For each module name in the `requires` array, YUI pulls in the requirement wherever it is needed, loading it remotely if necessary.

Example 1-10 is a variation on Example 1-9. Instead of returning a string value, `Y.Hello.sayHello()` now changes the contents of a single `Y.Node`. The `hello` module now declares a dependency on `node-base` to ensure that `node.setHTML()` is always available wherever `hello` runs.

To make things a little more interesting, `sayHello()` uses a private helper function named `setNodeMessage()`. Users cannot call `setNodeMessage()` directly because it is not attached to `Y`. `setNodeMessage()` uses `Y.one()` to normalize the input to a YUI node, then sets the message text.

Example 1-10. Creating a module that depends on a YUI node

```
<!DOCTYPE html>
<title>Creating a module that depends on a YUI node</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI.add('hello', function (Y) {

    function setNodeMessage(node, html) {
        node = Y.one(node);
        if (node) {
            node.setHTML(html);
        }
    }

    Y.namespace('Hello').sayHello = function (node) {
        setNodeMessage(node, 'GREETINGS PROGRAMS');
    };

}, '0.0.1', {requires: ['node-base']});

YUI().use('hello', function (Y) {
    Y.Hello.sayHello(Y.one('#demo'));
});
</script>
```

Unlike Example 1-9, the `use()` call in Example 1-10 does not need to explicitly request `node-base`. The new, improved `hello` module now pulls in this requirement automatically.

Discussion

Example 1-10 lists the module `node-base` in the `requires` array for the `hello` module. This guarantees that `YUI().use()` loads and attaches `hello` to the `Y` *after* attaching `node-base` (or any other modules you add to that array).

When providing requirements, take care to avoid circular dependencies. For example, if `hello` declares that the `goodbye` module must be loaded before `hello`, but the `goodbye` module declares that `hello` must be loaded before `goodbye`, you have a problem. The Loader does have some logic to defend against metadata with circular dependencies, but you shouldn't count on your code running correctly.

For performance reasons, you should also provide your module's requirements in the Loader metadata, as described in Recipe 1.10.

As mentioned earlier, `requires` is the most important field. Some of the other fields for `YUI.add()` include:

`optional`
: An array of module names to automatically include with your module, but only if the YUI configuration value `loadOptional` is set to `true`. For example, `autocomplete-base` declares an optional dependency on `autocomplete-sources`, which contains extra functionality for populating an `AutoComplete` widget from YQL and other remote sources. `loadOptional` is `false` by default.

  Even if `loadOptional` is `false`, an optional dependency still causes a module to activate if the module's code happens to already be loaded on the page. Modules can be present on the page due to an earlier `YUI().use()` call, or by loading module code statically, as shown in Recipe 1.20.

`skinnable`
: A Boolean indicating whether your module has a CSS skin. If this field is `true`, YUI automatically creates a `<link>` element in the document and attempts to load a CSS file using a URL of:

  base/*module-name*/assets/skins/*skin-name*/*module-name*.css

  where *base* is the value of the `base` field (discussed in Recipe 1.11) and `skin-name` is the name of the skin, which defaults to the value `sam`. For more information about creating skins, refer to Recipe 7.10.

`use`
: **Deprecated.** An array of module names used to define "physical rollups," an older deprecated type of rollup. To create modern rollups, refer to Recipe 1.13.

In addition to module dependencies, Example 1-10 also illustrates a private function within a module. Since JavaScript lacks an explicit `private` keyword, many JavaScript developers signify private data with an underscore prefix, which warns other developers that the function or variable "should" be private. In many cases, this form of privacy is good enough.

However, the `setNodeMessage()` function in the example is truly private. Once YUI executes the `add()` callback, module users can call `sayHello()`, but they can never call `setNodeMessage()` directly, even though `sayHello()` maintains its internal reference to `setNodeMessage()`. In JavaScript, an inner function continues to have access to all the members of its outer function, even after the outer function executes. This important property of the language is called *closure*.

See Also

Recipe 7.10; Douglas Crockford on "Private Members in JavaScript" (*http://www.crockford.com/javascript/private.html*).

1.10 Creating Truly Reusable Modules

Problem

You want to create a custom YUI module by defining the module's code in a separate file, then reuse the module in multiple HTML pages.

Solution

Examples 1-9 and 1-10 each define a custom module, but then proceed to `use()` the module in the same `<script>` block on the same HTML page. Truly reusable modules are defined in a file separate from where they are used.

This creates a problem. For modules not yet on the page, Loader needs metadata about a module *before* attempting to load that module, such as where the module resides and what its dependencies are. Fortunately, you can provide this information by configuring the YUI object, as shown in Example 1-11.

Example 1-11. Creating a reusable module

add_reusable.html: Creates a YUI instance and passes in a configuration object that defines the `hello` module's full path and dependencies.

```
<!DOCTYPE html>
<title>Creating a reusable module</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI({
```

```
    modules: {
        'hello': {
            fullpath: 'hello.js',
            requires: ['node-base']
        }
    }
}).use('hello', function (Y) {
    Y.Hello.sayHello(Y.one('#demo'));
});
</script>
```

With this metadata, you do not need to manually add an extra `<script>` element to load the *hello.js* file. The `fullpath`, which can point to a local file or remote URL, is enough information for the YUI Loader to fetch the code. Declaring `node-base` as a dependency instructs the Loader to fetch `node-base` before fetching `hello`.

Since YUI module names often contain dashes, it is a YUI convention to always quote module names in configuration metadata, even if those quotes are not strictly necessary.

hello.js: Contains only the JavaScript for the `hello` module, identical to the version in Example 1-10. This file resides in the same directory as *add_reusable.html*.

```
YUI.add('hello', function (Y) {

    function setNodeMessage(node, html) {
        node = Y.one(node);
        if (node) {
            node.setHTML(html);
        }
    }

    Y.namespace('Hello').sayHello = function (node) {
        setNodeMessage(node, 'GREETINGS PROGRAMS');
    };

}, '0.0.1', {requires: ['node-base']});
```

Discussion

Example 1-11 is a minimal example of a single, simple module. The configuration object gets more complex as you add more modules and more dependencies, as shown shortly in Example 1-12.

So why doesn't YUI need a giant configuration object to load the core YUI modules? The answer is that YUI cheats—this information is included in the YUI seed. The default seed file includes both the Loader code and metadata for all the core YUI modules, but you can load more minimal seeds if need be. For more information about alternate seed files, refer to "YUI and Loader changes for 3.4.0" (*http://www.yuiblog.com/blog/2011/07/01/yui-and-loader-changes-for-3-4-0/*).

You might have noticed that the metadata `requires: ['node-base']` is provided twice: once in the YUI configuration that gets passed to the Loader, and again in the

`YUI.add()` that defines the module. If the Loader has this metadata, why bother repeating this information in `YUI.add()`?

The answer has to do with certain advanced use cases where the Loader is not present. For example, if you build your own combo load URL, load a minimal seed that lacks the Loader code, and then call `YUI().use('*')` as described in Recipe 1.20, the metadata in `YUI.add()` serves as a fallback for determining dependencies.

1.11 Defining Groups of Custom Modules

Problem

You want to define a group of related modules that all reside under the same path on the server.

Solution

In your YUI configuration, use the `groups` field to create a group of related modules that share the same `base` path and other characteristics.

Example 1-12 is configured to run from a real web server. If you prefer to open *add_group.html* as a local file, change the `base` configuration field to be a relative filepath such as *./js/local-modules/*.

Example 1-12. Defining a module group

add_group.html: Defines the `local-modules` module group, which contains four modules that reside under */js/local-modules*, plus a CSS skin file. The main module, `reptiles-core`, pulls in the `node` rollup for DOM manipulation and two more local modules for additional giant reptile-related functionality.

```
<!DOCTYPE html>
<title>Defining a module group</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI({
    groups: {
        'local-modules': {
            base: '/js/local-modules/',
            modules: {
                'reptiles-core': {
                    path: 'reptiles-core/reptiles-core.js',
                    requires: ['node', 'reptiles-stomp', 'reptiles-fiery-breath'],
                    skinnable: true
                },
```

```
                'reptiles-stomp': {
                    path: 'reptiles-stomp/reptiles-stomp.js'
                },
                'reptiles-fiery-breath': {
                    path: 'reptiles-fiery-breath/reptiles-fiery-breath.js'
                },
                'samurai': {
                    path: 'samurai/samurai.js'
                }
            }
        }
    }
}).use('reptiles-core', function (Y) {
    Y.Reptiles.info(Y.one('#demo'));
});
</script>
```

/js/reptiles/giant-reptiles.js: Defines the `reptiles-core` module, which pulls in three other modules and provides an `info()` method that appends a `<ul>` into the DOM.

```
YUI.add('reptiles-core', function (Y) {
    var reptiles = Y.namespace('Reptiles');

    reptiles.traits = [
        'dark eyes',
        'shiny teeth'
    ];

    reptiles.info = function (node) {
        var out = '', i;
        for (i = 0; i < reptiles.traits.length; i += 1) {
            out += '<li>' + reptiles.traits[i] + '</li>';
        };
        out += '<li>' + reptiles.breathe() + '</li>';
        out += '<li>' + reptiles.stomp() + '</li>';
        node.append('<ul class="reptile">' + out + '</ul>');
    };
}, '0.0.1', {requires: ['node', 'reptiles-stomp', 'reptiles-fiery-breath']});
```

/js/reptiles/stomp.js: Defines the `Y.Reptiles.stomp()` method.

```
YUI.add('reptiles-stomp', function (Y) {
    Y.namespace('Reptiles').stomp = function () {
        return 'STOMP!!';
    };
}, '0.0.1');
```

/js/reptiles/fiery-breath.js: Defines the `Y.Reptiles.breathe()` method.

```
YUI.add('reptiles-fiery-breath', function (Y) {
    Y.namespace('Reptiles').breathe = function () {
        return 'WHOOOSH!';
    };
}, '0.0.1');
```

/js/local-modules/reptiles-core/assets/skins/sam/reptiles-core.css: Defines the CSS skin for the `reptiles-core` module. YUI attempts to load this file because the `skinnable` field for `reptiles-core` is set to `true`. For more information about how this works, refer to the Discussion.

```
.reptile li { color: #060; }
```

The `samurai` module definition is empty. Feel free to make up your own definition.

Discussion

For multiple custom modules, consider using this convention for your module structure:

```
base/
  module-foo/
    module-foo.js
    assets/
      skins/
        sam/
          module-foo.css
          sprite.png
  module-bar/
    ...
```

that is, a base path with one directory per module. Each module directory contains at least one JavaScript file, possibly more if you include the **-min.js* or **-debug.js* versions of your modules. If the module has a skin, it should also contain an *assets/* directory, as shown in Recipe 7.10. If it has localized language resources, it should contain a *lang/* directory, as shown in Recipe 11.6.

Module groups create a configuration context where you can load modules from somewhere other than the Yahoo! CDN. You do not need to use module groups for logical groupings of your own modules ("all my widgets," "all my utility objects," and so on). For those kinds of logical groupings, it is more appropriate to create custom rollups, as described in Recipe 1.13. Module groups are for providing the Loader with a different set of metadata for loading modules from a *particular server and set of paths*: your own custom modules, third-party modules on some remote server, your own local copy of the core YUI library or YUI gallery, and so on.

In many cases, a module group is a necessity. Consider loading a local CSS skin. As described in Recipe 1.9, setting `skinnable` to `true` causes YUI to attempt to fetch a skin from:

base/*module-name*/assets/skins/*skin-name*/*module-name*.css

`base` defaults to the same prefix that you loaded the YUI seed file from, typically something like `http://yui.yahooapis.com/3.5.0/build`. So what happens if you try to load skin CSS from your own local server without using a module group?

```
<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI({
    modules: {
        'reptiles-core': {
            fullpath: '/js/local-modules/reptiles-core/reptiles-core.js',
            skinnable: true
        },
        ...
    }
}).use('reptiles-core', ...);
```

This configuration fails because YUI attempts to load your skin from *http://yui.yahooapis.com/3.5.0/build/reptiles-core/assets/skins/sam/reptiles-core.css*, instead of your local server.

What if you set `base` to act locally? For example:

```
YUI({
    base: '/js/local-modules/',
    modules: {
        'reptiles-core': {
            path: 'reptiles-core/reptiles-core.js',
            skinnable: true
        },
        ...
    }
}).use('reptiles-core', ...);
```

This is also undesirable because now YUI is configured to fetch *all* modules, including the YUI core and gallery modules, from this local path. Using a module group enables you to set the `base` path for all of your local modules without messing up the loader configuration for the core modules.

See Also

Recipe 1.13; Recipe 7.9; Recipe 7.10; Recipe 11.6; the YUI Loader section of the `YUI` Global Object User Guide (*http://yuilibrary.com/yui/docs/yui/#loader*).

1.12 Reusing a YUI Configuration

Problem

You want to reuse a complex configuration across multiple pages.

Solution

Before creating any YUI instances, load a separate script file containing a `YUI_config` object that stores all custom module configuration and other metadata you need. If the page contains a `YUI_config` object, YUI automatically applies this configuration to any YUI instances on the page.

Example 1-13 is a variation of Example 1-12, but with the module metadata broken out into its own reusable file.

Example 1-13 is configured to run from a real web server. If you prefer to open *add_yui_config.html* as a local file, change all */js* filepaths to relative filepaths such as *./js/*.

Example 1-13. Reusing a YUI configuration

add_yui_config.html: Loads and exercises the `reptiles-core` module using an implicit YUI configuration supplied by */js/yui_config.js*. The key word is "implicit"—you do not need to explicitly pass `YUI_config` into the `YUI()` constructor.

```
<!DOCTYPE html>
<title>Reusing a YUI configuration</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script src="/js/yui_config.js"></script>
<script>
YUI().use('reptiles-core', function (Y) {
    Y.Reptiles.info(Y.one('#demo'));
});
</script>
```

/js/yui_config.js: Provides the configuration data for loading a set of custom modules.

```
var YUI_config = {
    groups: {
        'local-modules': {
            base: '/js/local-modules/',
            modules: {
                'reptiles-core': {
                    path: 'reptiles-core/reptiles-core.js',
                    requires: ['node', 'reptiles-stomp', 'reptiles-fiery-breath'],
                    skinnable: true
                },
                'reptiles-stomp': {
                    path: 'reptiles-stomp/reptiles-stomp.js'
                },
                'reptiles-fiery-breath': {
                    path: 'reptiles-fiery-breath/reptiles-fiery-breath.js'
                },
                'samurai': {
                    path: 'samurai/samurai.js'
                }
            }
        }
    }
};
```

The other JavaScript files in this example are identical to the ones in Example 1-12.

Discussion

At construction time, each YUI instance attempts to merge the common `YUI_config` object into the configuration object you passed into the `YUI()` constructor. Thus, something like:

```
<script src="/js/yui_config.js"></script>
<script>
YUI({ lang: 'jp' }).use('reptiles-core', function (Y) {
    Y.Reptiles.info(Y.one('#demo'));
});
</script>
```

would safely add the `lang` property without clobbering the module metadata. Properties you supply to the constructor override properties in `YUI_config`.

If you're careful about how you merge configuration data, you can add new module groups or even new modules within an existing module group, as shown in Example 1-14.

Example 1-14. Merging common and page-specific YUI configuration

add_yui_config_merged.html: Loads and exercises the `reptiles-core` module using an implicit YUI configuration supplied by */js/yui_config_incomplete.js*, and merges some extra configuration information into the `YUI()` constructor.

```
<!DOCTYPE html>
<title>Merging common and page-specific YUI configuration</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script src="/js/yui_config_incomplete.js"></script>
<script>
YUI({
    groups: {
        'local-modules': {
            base: '/js/local-modules/',
            modules: {
                'reptiles-stomp': {
                    path: 'reptiles-stomp/reptiles-stomp.js'
                }
            }
        }
    }
}).use('reptiles-core', function (Y) {
    Y.Reptiles.info(Y.one('#demo'));
});
</script>
```

/js/yui_config.js: Provides some (intentionally incomplete) configuration data for loading a set of custom modules. The configuration is broken in two places: first, the `reptiles-stomp` module definition is missing, and second, the `base` path is incorrect. However, the configuration object provided in the HTML file fixes both problems.

```
// WARNING: Config intentionally incomplete/broken
var YUI_config = {
    groups: {
        'local-modules': {
            base: '/js/BOGUS_PATH',
            modules: {
                'reptiles-core': {
                    path: 'reptiles-core/reptiles-core.js',
                    requires: ['node', 'reptiles-stomp', 'reptiles-fiery-breath'],
                    skinnable: true
                },
                'reptiles-fiery-breath': {
                    path: 'reptiles-fiery-breath/reptiles-fiery-breath.js'
                },
                'samurai': {
                    path: 'samurai/samurai.js'
                }
            }
        }
    }
};
```

Example 1-14 supplies an incomplete `YUI_config` object in order to demonstrate that the merging actually works. More generally, you would use `YUI_config` to provide a complete, working configuration for everything that is common across your site, and then supply additional page-specific information either in the YUI instance constructor, or by modifying `YUI_config` (which would affect all instances on the page).

Once you're within a YUI instance, you can call `Y.applyConfig()` at any time to merge in additional configuration. You can even call `Y.applyConfig()` to load more module metadata, perhaps along with on-demand loading techniques such as those shown in Recipes 1.17 and 1.18.

1.13 Defining Your Own Rollups

Problem

You would like to define a particular stack of modules under a friendly alias for convenient reuse.

Solution

Define an empty module and provide it with a `use` field containing an array of other module or rollup names. Then load and use it as you would any other module.

Example 1-15 represents a simple rollup that serves as an alias for `node-base` and `json` (which is itself a rollup of `json-parse` and `json-stringify`). The custom `my-stack` rollup behaves like any of the other popular core YUI rollups, such as `node`, `io`, `json`, or `transition`.

Example 1-15. Defining your own rollups

```
<!DOCTYPE html>
<title>Defining your own rollups</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI({
    modules: {
        'my-stack': {
            use: ['node-base', 'json']
        }
    }
}).use('my-stack', function (Y) {
    var dataStr = '{ "rollups": "are neat" }',
        data    = Y.JSON.parse(dataStr);

    Y.one('#demo').setHTML(data.rollups);
});
</script>
```

Discussion

As Example 1-15 demonstrates, a rollup is just an alias for a list of other rollups and modules. The example uses core YUI modules, but you can also include gallery modules, your own custom modules, or anything else.

Rollups are great for logically grouping modules that represent major components of your application stack, or for grouping modules that are closely related, but don't strictly depend on each other. For example, `json-parse` and `json-stringify` are completely independent modules, but applications often end up using both anyway.

Another benefit of rollups is that they free you up to encapsulate your code into even smaller chunks than you otherwise might have. You can use rollups to bundle very tiny modules into larger units, making it easier for others to use your code without having to worry about the fiddly details of what to include.

See Also

Recipe 1.1; Recipe 1.10.

1.14 Loading jQuery as a YUI Module

Problem

You want to load jQuery and some jQuery plugins into the sandbox alongside YUI, just like any YUI module.

Solution

Create a module group that defines module metadata for the main jQuery library and any other jQuery-related code that you want to load as well. Use `base` and `path` (or `fullpath`) to point to the remote files.

If you need to load multiple jQuery files in a particular order, use `requires` to specify the dependency tree, and set `async: false` for the overall module group. Setting `async: false` is necessary for loading any code that is *not* wrapped in a `YUI.add()`—it ensures that third-party code loads synchronously, in the correct file order.

After defining jQuery files as YUI modules, you can then `use()` them alongside any ordinary YUI modules you like. Example 1-16 pulls in the YUI `calendar` module along with jQuery and jQuery UI, which includes the jQuery Datepicker plugin. Unlike YUI core widgets, the jQuery Datepicker's CSS does not get loaded automatically, so you must load it as a separate CSS module. For more information about loading arbitrary CSS as a YUI module, refer to Recipe 7.9.

Experienced jQuery developers might have noticed that the example simply renders the Datepicker without bothering to wrap it in a `$(document).ready()`. The standard YUI loading pattern with JavaScript at the bottom of the page usually makes DOM readiness a nonissue. However, if you modify elements that occur after your `<script>` element or load YUI in an unusual way, you might need to wait for DOM readiness. For YUI's equivalent of jQuery's `ready()`, refer to Recipe 4.2.

Example 1-16. Loading jQuery as a YUI module

```
<!DOCTYPE html>
<title>Loading jQuery as a YUI module</title>
<style>
h4 { margin: 25px 0px 10px 0px; }
div.container { width: 300px; }
</style>

<body class="yui3-skin-sam">

<h4>YUI 3 Calendar Widget</h4>
<div class="container" id="ycalendar"></div>

<h4>jQuery UI Calendar Plugin</h4>
<div class="container" id="datepicker"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI({
    groups: {
        'jquery': {
            base: 'http://ajax.googleapis.com/ajax/libs/',
            async: false,
```

```
            modules: {
                'jquery': {
                    path: 'jquery/1.7/jquery.min.js'
                },
                'jquery-ui': {
                    path: 'jqueryui/1.8/jquery-ui.min.js',
                    requires: ['jquery', 'jquery-ui-css']
                },
                'jquery-ui-css': {
                    path: 'jqueryui/1.8/themes/base/jquery-ui.css',
                    type: 'css'
                }
            }
        }
    }
}).use('calendar', 'jquery-ui', function (Y) {
    new Y.Calendar().render('#ycalendar');
    $('#datepicker').datepicker();
    Y.one('body').append('<p>YUI and jQuery, living together, mass hysteria!</p>');
});
</script>
</body>
```

As with any module, it's critical to define your dependencies correctly. Here, the `jquery-ui` module declares a dependency on `jquery` and `jquery-ui-css`, which ensures that YUI adds jQuery's code to the page *above* jQuery UI's code. If you somehow got the dependencies backward and declared that `jquery` depended on `jquery-ui`, then YUI would add jQuery *below* jQuery UI, which would break the Datepicker plugin.

Of course, you're not restricted to just core jQuery and jQuery UI. As long as you declare your paths and dependencies correctly, you can load any third-party jQuery plugin (or any other library code, for that matter).

Discussion

Loading jQuery, Dojo, Scriptaculous, or any other major framework into a YUI sandbox is not exactly a recipe for great efficiency. If you've loaded the code necessary to do both `Y.one('#demo')` and `$('#demo')` in the same page, you've loaded an awful lot of duplicate code for rummaging around the DOM.

That said, the YUI Loader is an excellent standalone script and CSS loader. It can load any third-party JS or CSS file you like, in any order you like, as long as you provide the correct metadata. Some reasons you might want to do this include:

- Easy code reuse. You have found some critical feature or component that is available only in some other library.
- Better collaboration. You are working primarily in YUI, but you have teammates or contractors who have written non-YUI code that you need to quickly integrate, or vice versa.

- Improving perceived performance. Your non-YUI pages are currently littered with blocking `<script>` and `<link>` elements at the top of the document. You're looking for a quick way to migrate over to a more advanced loading pattern, and perhaps even take advantage of some advanced YUI Loader tricks such as those covered in Recipes 1.15 and 1.17.

In fact, if you want to use the Loader to load non-YUI scripts only, and you are *sure* that you don't need to load any core YUI modules, consider loading the `yui-base-min.js` seed rather than the `yui-min.js` seed:

```
<script src="http://yui.yahooapis.com/3.5.0/build/yui-base/yui-base-min.js"></script>
```

The `yui-base-min.js` seed includes the YUI module registry and the YUI Loader, but leaves out all the metadata for the core YUI modules. This makes it a little more efficient to load the YUI seed solely for loading and managing third-party scripts.

YUI is designed to be compatible with most major libraries, although you might run into strange conflicts here and there. The most common reason for bugs is when the other library modifies the prototype of a native JavaScript or native DOM object. YUI provides solid abstraction layers around native objects, but these abstractions can break if the other library changes object behavior at a deep level.

The other thing to watch out for is forgetting that different libraries use different abstractions. For example, you can't pass a YUI `Node` instance directly into some other library for further DOM manipulation. If you are building some kind of Frankenstein's Monster application that does some DOM manipulation with YUI and some in Dojo, keep a close eye on each point where the two libraries communicate.

See Also

jQuery (*http://docs.jquery.com*); jQuery UI.Datepicker (*http://docs.jquery.com/UI/Datepicker*); jCarousel (*http://sorgalla.com/projects/jcarousel/*); the jQuery–YUI 3 Rosetta Stone (*http://www.jsrosettastone.com/*); an explanation of the different seed files in YUI and Loader changes for 3.4.0 (*http://www.yuiblog.com/blog/2011/07/01/yui-and-loader-changes-for-3-4-0/*).

1.15 Loading Modules Based on Browser Capabilities

Problem

You want YUI to supply additional fallback code to support users who have legacy browsers, but without penalizing users who have modern browsers. (This is called *capability-based loading*.)

Solution

In your YUI configuration, use the `condition` field to flag a module as conditional. A conditional module loads only if some other module specified by `trigger` is present, and then only if the `test` function returns `true`.

Example 1-17 demonstrates a simple `suitcase` module that can store data on the client. By default, the module tries to use `localStorage`, but if the browser is too old to support this feature natively, YUI loads an extra module that stores data using cookies instead.

Example 1-17. Loading modules based on browser capabilities

add_capability.html: Creates a YUI instance and passes in a configuration object that defines metadata for the `suitcase` module and for the `suitcase-legacy` conditional module.

```
<!DOCTYPE html>
<title>Loading modules based on browser capabilities</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI({
    modules: {
        'suitcase': {
            fullpath: 'suitcase.js'
        },
        'suitcase-legacy': {
            fullpath: 'suitcase-legacy.js',
            condition: {
                trigger: 'suitcase',
                test: function () {
                    try {
                        return window.localStorage ? false : true;
                    } catch(ex) {
                        return true;
                    }
                }
            },
            requires: ['suitcase', 'cookie']
        }
    }
}).use('node', 'suitcase', function (Y) {
    var type = Y.Cookie ? 'battered, legacy' : 'sleek, ultra-modern';
    Y.Suitcase.set('foo', 'bar');
    Y.one('#demo').setHTML('In your ' + type + ' suitcase: ' + Y.Suitcase.get('foo'));
});
</script>
```

The `suitcase-legacy` module has a trigger condition. If the `suitcase` module is passed into `use()`, YUI executes `suitcase-legacy`'s test function. If the browser does *not* support `localStorage`, the function returns `true`, which causes YUI to also fetch `suitcase-legacy` and all its dependencies. If the function does support `localStorage`, YUI skips fetching `suitcase-legacy`.

Within the `use()` callback, the presence of `Y.Cookie` is a quick way to check whether `suitcase-legacy` was successfully triggered.

suitcase.js: Defines a simple get/set API for storing data on the client using `localStorage`. Note that the `suitcase` module is written without any "knowledge" of the `suitcase-legacy` API. Capability-based loading is designed to help you avoid having to include extra conditionals or other unnecessary code in your main modules.

```
YUI.add('suitcase', function (Y) {
    Y.Suitcase = {
        get: function (name) {
            return localStorage.getItem(name);
        },
        set: function (name, value) {
            localStorage.setItem(name, value);
        }
    };
}, '0.0.1');
```

suitcase-legacy.js: Defines the legacy cookie-based get/set API. Because of dependency ordering, YUI must load `suitcase-legacy` after `suitcase`, which means that the `get()` and `set()` methods from `suitcase-legacy` always overwrite the `get()` and `set()` methods from `suitcase`. In other words, if both modules are loaded on the page, calling `Y.Suitcase.get()` will use `cookies`, not `localStorage`.

```
YUI.add('suitcase-legacy', function (Y) {
    Y.Suitcase = {
        get: function (name) {
            return Y.Cookie.get(name);
        },
        set: function (name, value) {
            Y.Cookie.set(name, value);
        }
    };
}, '0.0.1', { requires: ['suitcase', 'cookie'] });
```

Fortunately for users (but unfortunately for demonstration purposes), `localStorage` is widely available in most browsers. If you don't have a really old browser available that can show the legacy module in action, feel free to hack the example and change the test function to just return `true`.

The `Suitcase` object is a toy example. YUI already provides more professional storage APIs called `Cache` and `CacheOffline`. Like `Suitcase`, `CacheOffline` is able to use `localStorage` when that feature is available.

Discussion

Supporting older, less capable browsers often requires supplying extra JavaScript to correct for bugs and to emulate more advanced native features. After writing and testing code to correct older browsers, the last thing you want to do is penalize cutting-edge users by forcing them to download extra code.

YUI's capability-based loading solves this problem by enabling you to break legacy code out into separate modules. Older browsers can load and execute the extra code they need, while newer browsers suffer only the small performance hit of evaluating a few conditionals.

The core YUI library uses capability-based loading to do things like:

- Avoid loading support for physical keyboard events on iPhones
- Make DOM-ready events safer on old versions of Internet Explorer, without penalizing other browsers
- Seamlessly use the best graphics feature available for the given browser: SVG, Canvas, or VML

While capability-based loading was originally designed for patching up legacy browsers, you can also flip this idea around and serve up extra code that unlocks features in a *more* capable browser. For example, let's say your application must perform an expensive calculation. Older browsers run the calculation directly and suffer an annoying UI freeze. However, if the browser supports the Web Worker API, YUI could trigger a conditional module that uses workers to run the calculation in the background. Usually you want to avoid "penalizing" newer browsers with an extra download, but if the benefits are high enough, it might be worth doing.

Most conditional modules should be abstracted behind another API. In Example 1-17, the modules are designed so that developers can call `Y.Suitcase.get()` and `Y.Suitcase.set()` without knowing whether the legacy implementation was in effect. Of course, this abstraction can be slower than the native implementation, or break down at the edges in some other way. For example, anyone who tries to store a 3 MB object in `Y.Suitcase` using a legacy browser will be sorely disappointed.

For obvious reasons, capability test functions should execute quickly. A typical capability test either checks for the existence of an object property, or creates a new DOM element and runs some subsequent operation on that element. Unfortunately, touching the DOM is expensive, and even more unfortunately, sometimes capability tests need to do substantial work, since just because a browser exposes a certain property or method doesn't mean that the feature works properly. As an example, the test function in Example 1-17 needs a try/catch statement in order to work around an edge-case bug in older versions of Firefox.

Capability testing can be a surprisingly deep rabbit hole. In extreme cases where capability testing has become hopelessly complex or slow, you might consider using the

`Y.UA` object. `Y.UA` performs user-agent sniffing, which many web developers regard as evil. Still, `Y.UA` is there, just in case you really do need to use the Dark Side of the Force. `Y.UA` can also be useful when capability testing isn't helpful for answering the question, such as when you need to detect certain CSS or rendering quirks.

See Also

The W3C standard for web storage (*http://www.w3.org/TR/webstorage/*); the YUI `Cookie` API (*http://yuilibrary.com/yui/docs/cookie/*).

1.16 Monkeypatching YUI

Problem

You want to conditionally load extra code at runtime to patch a YUI bug or hack new behavior into YUI.

Solution

In your YUI configuration, define one or more patch modules, using the `condition` field to flag those modules as conditional. Set the `trigger` field to the name of the module to patch, and create a test function that simply returns `true`.

Example 1-18 loads a module that patches `node-base`, changing the behavior of `setHTML()`. Ordinarily, `setHTML()` is a safer version of setting `innerHTML`; before blowing away the node's internal contents, `setHTML()` walks the DOM and cleanly detaches any event listeners. For whatever reason, you've decided this safer behavior is undesirable. The "patch" clobbers `setHTML()`, turning it into a simple alias for setting `innerHTML`.

Example 1-18 is configured to run from a real web server. If you prefer to open *add_monkeypatching.html* as a local file, change the `base` configuration field to be a relative filepath such as *./js/patches/*.

Example 1-18. Monkeypatching YUI

add_monkeypatching.html: Creates a YUI instance and passes in a configuration object that defines metadata for the `node-patches` conditional module.

```
<!DOCTYPE html>
<title>Monkeypatching YUI</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI({
    groups: {
```

```
        patches: {
            base: '/js/patches/',
            modules: {
                'node-patches' : {
                    path: 'node-patches/node-patches.js',
                    condition: {
                        name: 'node-patches',
                        trigger : 'node-base',
                        test : function () { return true; }
                    }
                }
            }
        }
    }
}).use('node-base', function (Y) {
    Y.one('#demo').setHTML("Hmmm, setHTML() is unusually fast these days.");
});
</script>
```

/js/patches/node-patches/node-patches.js: Provides additional code that overrides `Node`'s `setHTML()` method. The patch module loads only if `node-base` is loaded.

```
YUI.add('node-patches', function (Y) {
    Y.Node.prototype.setHTML = function (content) {
        this.set('innerHTML', content);
    }
});
```

Discussion

Monkeypatching refers to modifying the behavior of a program at runtime without altering the upstream source. Monkeypatching can be useful for implementing quick fixes, but as the name implies, it isn't necessarily the best approach for long-term stability.

Example 1-18 represents a somewhat contrived behavior change. More generally, you could use monkeypatching to temporarily address a serious bug in the YUI library, or to inject behavior that you need in a development or staging environment, but not in production.

When patching someone else's code, you can use `Y.Do.before()` and `Y.Do.after()` to cleanly inject behavior into a program without clobbering an existing method. For more information, refer to Recipe 4.12.

See Also

Recipe 1.15; YUI Tutorial: "Report a Bug" (*http://yuilibrary.com/yui/docs/tutorials/report-bugs/*).

1.17 Loading Modules on Demand

Problem

You have a feature that your application needs only some of the time. You want to load this code only for users who need it, without affecting the initial page load.

Solution

Instead of loading the optional code up front, call `Y.use()` *within* the top-level `YUI().use()` sandbox to load the optional code on demand.

For example, suppose you need to display a confirmation pane when the user clicks a button. The straightforward approach is to load the `overlay` module with `YUI().use()`, create a new `Overlay` instance, and then bind a `click` event to the button that will `show()` the overlay. For examples of using overlays, refer to Recipe 8.2.

Although there's nothing wrong with that approach, users still have to load the `overlay` module and its dependencies even if they never click the button. You can improve the performance of the initial page view by deferring loading and executing code until the moment the user needs it, as shown in Example 1-19:

1. Create a top-level `showOverlay()` function.
2. Within `showOverlay()`, call `Y.use()` to load the `overlay` module.
3. Within the `Y.use()` callback function:
   a. Create a new `Overlay` instance, initially set to be invisible.
   b. Redefine the `showOverlay()` function to do something else. The next time `showOverlay()` is called, it will simply show the hidden overlay instance.
   c. Call the newly redefined `showOverlay()` from within `showOverlay()` to make the overlay instance visible.
4. Bind "hide" and "show" callback functions as `click` events for the two respective buttons:
   - The "hide" callback first checks whether the overlay has been created.
   - The "show" callback calls `showOverlay()`. The first button click invokes the "heavy" version of `showOverlay()`, the version that loads the `overlay` module, instantiates an overlay, and then redefines itself. Subsequent clicks invoke the "light" version of `showOverlay()`, which flips the overlay into the visible state.

Example 1-19. Loading the overlay module on demand

```
<!DOCTYPE html>
<title>Loading the overlay module on demand</title>
<style>
.yui3-overlay-content {
    padding: 2px;
    border: 1px solid #000;
```

```
    border-radius: 6px;
    background-color: #afa;
}
</style>

<button id="show">Show Overlay</button>
<button id="hide">Hide Overlay</button>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node-base', function (Y) {
    var overlay;

    var showOverlay = function () {
        Y.use('overlay', function () {
            overlay = new Y.Overlay({
                bodyContent: 'Hello!',
                centered: true,
                height: 100,
                render: true,
                visible: false,
                width: 200,
                zIndex: 2
            });

            showOverlay = function () {
                overlay.show();
            };

            showOverlay();
        });
    };

    Y.one('#hide').on('click', function () {
        if (overlay) {
            overlay.hide();
        }
    });

    Y.one('#show').on('click', function () {
        showOverlay();
    });
});
</script>
```

Discussion

Example 1-19 illustrates two concepts. The first is the ability of functions in JavaScript to redefine themselves. A function calling itself (*recursion*) is common enough, but a function that redefines and then calls itself is less common. This pattern is useful if you have a function that needs to do one thing the first time it is called, and something else on subsequent calls. Use this technique sparingly, as there's a good chance you'll confuse people who read your code later on—including, possibly, yourself.

The second concept is the difference between the exterior `YUI().use()`, which creates a new YUI sandbox, and the interior `Y.use()`, which loads modules into the existing sandbox that's referenced by the `Y` variable. `Y.use()` enables you to load and attach additional modules at any time, for any reason. This is sometimes called *lazy loading*.

Lazy-loading modules can greatly improve your application's perceived performance. Native applications have a great advantage in that they start out with most or all of their code preloaded, while web applications have to bootstrap themselves over the network.

To compensate for this, you can divide your application into two pieces: a minimal *interactivity core* that provides just enough functionality to render the application, and additional components that you can lazy-load in the background as the user starts poking around. Example 1-19 attempts to be "smart" by loading extra code only if it is needed, but your application doesn't have to be this fancy. You could wait for your interactivity core to finish loading and then start loading all secondary components in the background, in order of priority.

Loading modules in response to user actions can cause a delay at the moment when the user triggers the loading. If this becomes a problem, you can just lazy-load all modules in the background regardless of whether they are needed, or alternatively, you can try to improve performance with *predictive loading*, as described in Recipe 1.18.

See Also

Eric Ferraiuolo's `gallery-base-componentmgr` module (*http://yuilibrary.com/gallery/show/base-componentmgr*), which makes it easy to lazy-load `Y.Base`-derived objects and their dependencies.

1.18 Enabling Predictive Module Loading on User Interaction

Problem

You have a feature that your application needs only some of the time, but that requires a lot of extra code to run. You want to load this code only for users who need it, without impacting the initial page load. You want to minimize any delay that occurs if a user does invoke the feature.

Solution

Use predictive loading to load the necessary code after the initial page load, but just before the user tries to invoke the feature.

In Example 1-19, the application defers loading the `overlay` module until the user clicks the button, which improves the inital page load time. However, this could cause an annoying delay when the user makes the first click.

Example 1-20 adds a refinement to the previous example. It calls `Y.use()` to load the `overlay` module in the background, but only if the user's mouse hovers over the Show Overlay button or if the button acquires focus. If the user then clicks on the button and the module has not yet loaded, the `click` event gets queued up until the `Overlay` widget is ready. To do this, the example separates loading from execution by creating a `loadOverlay()` function and a `showOverlay()` function.

1. The `loadOverlay()` function has different behavior depending on whether the overlay has already been instantiated, the `overlay` module is currently loading, or the `overlay` module needs to start loading.
   a. `loadOverlay()` takes a callback function, which turns out to be `showOverlay()`. If the overlay has already been instantiated, `loadOverlay()` executes the callback and returns immediately.
   b. If the `overlay` module is currently loading, this means the overlay is not yet ready to show. `loadOverlay()` queues the callback up in the `callbacks` array and returns immediately.
   c. If both of these conditions fail, this means the `loadOverlay()` function has been invoked for the first time. It is therefore time to start loading the `overlay` module. `loadOverlay()` calls `Y.use()` to load the `overlay` module on the fly.
   d. The `Y.use()` callback instantiates the overlay, sets `overlayLoading` to `false` (indicating that it is permissible to show the overlay), and finally executes any `showOverlay()` callbacks that have queued up while the code was loading.
2. The `showOverlay()` function is considerably simpler. If the overlay is already instantiated, the function shows the overlay. Otherwise, `showOverlay()` calls `loadOverlay()` with itself as the callback, which guarantees that `loadOverlay()` has at least one instance of `showOverlay()` queued up and ready to fire as soon as the overlay is instantiated.
3. The `hideOverlay()` function is simpler still. If the overlay is already instantiated, the function shows the overlay.
4. Finally, the script attaches event handlers to the Show Button and Hide Button. The `on()` method attaches an event handler, while the `once()` method attaches an event listener that automatically detaches itself the first time it is called.

Example 1-20. Loading the overlay module predictively

```
<!DOCTYPE html>
<title>Loading the overlay module predictively</title>
<style>
.yui3-overlay-content {
    padding: 2px;
    border: 1px solid #000;
    border-radius: 6px;
    background-color: #afa;
}
</style>
```

```
<button id="show">Show Overlay</button>
<button id="hide">Hide Overlay</button>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node-base', function (Y) {
    var callbacks = [],
        overlay,
        overlayLoading,
        showButton = Y.one('#show'),
        hideButton = Y.one('#hide');

    var loadOverlay = function (callback) {
        if (overlay) {
            if (callback) {
                callback();
            }
            return;
        }

        if (callback) {
            callbacks.push(callback);
        }

        if (overlayLoading) {
            return;
        }

        overlayLoading = true;

        Y.use('overlay', function () {
            var callback;

            overlay = new Y.Overlay({
                bodyContent: 'Hello!',
                centered: true,
                visible: false,
                height: 100,
                width: 200,
                zIndex: 2
            }).render();

            overlayLoading = false;

            while (callback = callbacks.shift()) {
                if (Y.Lang.isFunction(callback)) {
                    callback();
                }
            }
        });
    };
```

```
    var showOverlay = function () {
        if (overlay) {
            overlay.show();
        } else {
            loadOverlay(showOverlay);
        }
    };

    var hideOverlay = function () {
        if (overlay) {
            overlay.hide();
            callbacks = [];
        }
    };

    showButton.once('focus', loadOverlay);
    showButton.once('mouseover', loadOverlay);
    showButton.on('click', showOverlay);
    hideButton.on('click', hideOverlay);
});
</script>
```

For more information about the `Overlay` widget, refer to Recipe 8.2.

Discussion

While on-demand loading modules can help reduce initial load times, it can cause a delay when the user triggers the main event that requires the extra code. The goal of predictive loading is to start the loading a little earlier by using some other, related browser event that signals the user's possible intent to use the feature.

A reasonable way to predict that the user is likely to click a button is to listen for `mouseover` or `focus` events on the button or its container. You must listen for both events, since some users may use the mouse while others may use the keyboard. To get an even earlier indication of the user's intent, you could attach the `focus` and `mouseover` listeners to the button's container. For more information about using `on()` and `once()` to attach event handlers, refer to Chapter 4.

Thanks to these event handlers, the `loadOverlay()` function is called when the user is about to click the Show Overlay button. Since dynamic script loading is an asynchronous operation, `loadOverlay()` accepts an optional callback function as an argument, and calls that function once the overlay is ready to use.

To ensure that user clicks don't get lost while the `overlay` module is loading, multiple calls to `loadOverlay()` just add more callbacks to the queue, and all queued callbacks will be executed in order as soon as the overlay is ready. By the time the user actually clicks, the overlay should be ready to go, but if the user does manage to click while the code is loading, the overlay still appears as expected.

1.19 Binding a YUI Instance to an iframe

Problem

You want to manipulate an iframe using JavaScript in the parent document, without actually having to directly load YUI into the iframe.

Solution

Create a child YUI instance within your main YUI instance and bind the child instance to the iframe, as shown in Example 1-21. Every YUI instance has a `win` and a `doc` configuration value. By default, these values point to the native DOM window and document that are hosting YUI, but you can change them to point to the window and document of a different frame.

To set `win` and `doc`, use `document.getElementById()` to get a DOM reference to the iframe, then set `win` to the frame's `contentWindow` and `doc` to the frame's `contentWindow.document`. Note that `win` and `doc` are core configuration values and cannot be set to be YUI `Node` objects, as this would presuppose that every YUI instance has the `node` rollup loaded and available.

Example 1-21. Binding a YUI instance to an iframe

```
<!DOCTYPE html>
<title>Binding a YUI instance to an iframe</title>

<iframe src="iframe.html" id="frame"></iframe>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', function (Y) {
    var frame = document.getElementById('frame'),
        win = frame.contentWindow,
        doc = frame.contentWindow.document;

    YUI({ win: win, doc: doc }).use('node', function (innerY) {
        var innerBody = innerY.one('body');
        innerBody.addClass('foo');
        innerBody.append('<p>YUI3 was loaded!</p>');
    });
});
</script>
```

Nested instances are one of the few reasons to name the callback something other than `Y`. `innerY` is a fully functional YUI instance bound to the iframe's window and document. It has all the capabilities of a conventional `Y` instance, but scoped to the iframe. For example, calling `innerY.one('body')` gets the iframe's body, not the parent's body.

For security reasons, modern browsers prevent a parent document from manipulating a framed document with JavaScript unless the URLs of both documents have the same domain, protocol, and port. For this reason, be sure to host your iframes on the same server as the parent document.

If you try out Example 1-21 on your local filesystem using Chrome, the example fails due to Chrome's strict security policies around local files and JavaScript. In this case, just copy the example files to a real web server.

Discussion

If `win` and `doc` are not configured properly, iframes can be tricky to work with. For instance, the following code fails:

```
var frame = Y.one('#foo');
var h1 = frame.one('h1');
```

The first line is just fine: it retrieves a `Y.Node` instance for the iframe with an `id` of `foo`. But a naive call to `frame.one()` or `frame.all()` fails because YUI is scoped to work on the parent document.

One approach would be to add `<script>` markup and JavaScript code directly in the iframe, but this is clunky. The better strategy is to bind the iframe's window and document objects to a nested YUI instance. Within that instance, the YUI `Node` API works as expected on the iframe's content. Driving the iframe from the parent keeps all your code in one place and avoids having to fetch all your JavaScript code a second time from within the iframe. The iframe also has access to the `Y` instance for easy communication with the parent document.

If the iframe needs additional modules, you can first load them into the parent instance with a `Y.use()`, and then in the `Y.use()` callback, call `innerY.use()` to attach the module to the inner YUI instance. Example 1-22 is identical to Example 1-21, except that it also pulls in the `event` rollup in order to set a `click` event on the body of the iframe.

Example 1-22. Loading additional modules into an iframe

```
<!DOCTYPE html>
<title>Loading additional modules into an iframe</title>

<iframe src="iframe.html" id="frame"></iframe>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', function (Y) {
    var frame = document.getElementById('frame'),
        win = frame.contentWindow,
        doc = frame.contentWindow.document;

    YUI({ win: win, doc: doc }).use('node', function (innerY) {
```

```
        var innerBody = innerY.one('body');
        innerBody.addClass('foo');
        innerBody.append('<p>YUI3 was loaded!</p>');

        Y.use('event', function () {
            innerY.use('event', function () {
                innerBody.on('click', function () {
                    innerBody.replaceClass('foo', 'bar');
                });
            });
        });
    });
});
</script>
```

If your application makes heavy use of iframes, consider using `Y.Frame`, a utility included in the YUI Rich Text Editor widget.

See Also

"Security in Depth: Local Web Pages" (*http://blog.chromium.org/2008/12/security-in-depth-local-web-pages.html*) and Chromium Issue 47416 (*http://code.google.com/p/chromium/issues/detail?id=47416*), which describe the Chrome team's security concerns around local files, JavaScript, and frames; Andrew Wooldridge's "Hidden YUI Gem—Frame" (*http://andrewwooldridge.com/blog/2011/04/14/hidden-yui-gem-frame/*), which discusses a handy utility for working with iframes.

1.20 Implementing Static Loading

Problem

You want to improve YUI's initial load time by first loading all the modules you need in a single HTTP request, then attaching all modules to the `Y` instance at once.

Solution

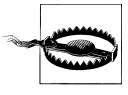
Use the YUI Configurator (*http://yuilibrary.com/yui/configurator/*) to handcraft a combo load URL for the YUI seed file and the exact list of modules you need. Then use this URL to fetch all YUI code in a single HTTP request. Once the code has downloaded, call `use('*')` to attach all YUI modules in the registry.

Ordinarily, the callback function passed into `use()` executes asynchronously after YUI calculates dependencies and fetches any missing resources. However, if you know that you have already loaded all modules you need onto the page, you can provide the special value `'*'` to `use()`, as shown in Example 1-23. This special value means that all necessary modules have already been loaded statically, and instructs YUI to simply attach every module in the registry to the `Y`. Even conditional modules, described in Recipe 1.15, get attached right away—regardless of the results of their `test` function.

Example 1-23. Loading node-base and dependencies statically

```
<!DOCTYPE html>
<title>Loading node-base and dependencies statically</title>

<div id="demo"></div>

<script type="text/javascript" src="http://yui.yahooapis.com/combo?
3.5.0/build/yui-base/yui-base-min.js&3.5.0/build/oop/oop-min.js&
3.5.0/build/event-custom-base/event-custom-base-min.js&
3.5.0/build/features/features-min.js&3.5.0/build/dom-core/dom-core-min.js&
3.5.0/build/dom-base/dom-base-min.js&3.5.0/build/selector-native/selector-native-min.js&
3.5.0/build/selector/selector-min.js&3.5.0/build/node-core/node-core-min.js&
3.5.0/build/node-base/node-base-min.js&3.5.0/build/event-base/event-base-min.js"></script>
<script>
YUI({
    bootstrap: false,
}).use('*', function (Y) {
    Y.one('#demo').setHTML('Real Programmers manage their dependencies manually.');
});
</script>
```

For good measure, the example sets `bootstrap` to `false`, which prevents the Loader from filling in any missing dependencies.



This technique can improve performance, but not without tradeoffs. For more information, refer to this recipe's Discussion.

Discussion

Static loading is yet another tool in your toolbox for managing application performance.

The YUI module system is designed to break large frameworks into tiny, digestible chunks that can be loaded asynchronously. This flexibility provides a huge performance advantage over monolithic libraries that force you to download the entire API whether you need it or not.

However, while dynamically constructing a custom library improves performance tremendously, it brings its own performance cost. First, calculating dependencies does not come for free. It is reasonably fast when done on the client side and very fast when done on the server side, but the cost is not zero. Second, loading YUI requires a minimum of two HTTP requests: one call to load the YUI seed file and one call to fetch the combo-loaded YUI modules.

If you are willing to throw the flexibility of the module system away, it is possible to squeeze a little extra performance from YUI. By listing all modules in the combo load URL, you can fetch everything in a single HTTP request and eliminate the need to calculate dependencies.

The disadvantage of this technique is that you are now responsible for managing your own dependencies across your entire application. If you want to upgrade to a new YUI minor version, add a YUI module to support a new feature, or remove a module that is no longer needed, you must recalculate your dependencies and update all your combo URLs yourself. If different pages might have different module requirements, you will have to maintain multiple distinct combo URLs. Static loading also makes it harder to take advantage of capability-based loading and other advanced techniques. If you are considering static loading, be sure to measure the real-world performance difference and weigh it against these increased maintenance costs.

CHAPTER 2
DOM Manipulation

The *document object model* (DOM) is not a particularly pleasant API to program against.

The main reason for this is that historically, browser DOM implementations have been incredibly buggy and inconsistent. Although JavaScript itself has its share of design flaws, many common complaints about JavaScript are actually complaints about the DOM.

A perhaps less appreciated reason is that the DOM is a low-level API that exposes only basic capabilities. By design, low-level APIs avoid making too many assumptions about how developers might want to use the underlying objects. Certain popular DOM extensions such as `innerHTML` and `querySelector` could be considered more mid-level, as they evolved based on what developers were actually doing.

JavaScript libraries have the advantage of being free to provide higher-level APIs that are more intuitive and terse than the lower-level DOM. However, each library comes with a strong mental model for how to work with the DOM. It would be a mistake to bake those models deeply into the DOM itself. (Imagine how unhappy jQuery developers would be if the only way to work with the DOM was the YUI way, or vice versa.)

In any case, the rise of JavaScript libraries has made it far easier to manipulate the DOM. A good DOM abstraction layer can:

- Correct for bugs and implementation differences in specific browsers. YUI accomplishes this using *feature detection* (testing for the existence of a feature) and *capability detection* (verifying whether the feature works properly). If a behavior is missing or incorrect, YUI corrects the problem. YUI's sophisticated Loader can fetch extra code to correct bugs, if and only if that code is needed.
- Enable you to use advanced features from newer specifications, even if the browser doesn't implement those features natively. If the feature is present, YUI uses the fast native implementation. If not, YUI implements the feature in JavaScript, providing you with a uniform interface.

- Provide a much more pleasant and capable API. Although stock DOM methods can get the job done, YUI and other frameworks offer friendly façades and helper methods that provide powerful capabilities with only a small amount of code.

Before JavaScript libraries, most web developers would learn about browser bugs the hard way, slowly building up their own personal bag of tricks. Individual browser bugs are not always difficult to work around, but some bugs are nastier than others, and it takes a special kind of thick-headedness to want to spend your time solving the cross-browser problem in general. Fortunately, YUI and its cousins all bake in years of hard-won experience around writing portable code, freeing up your time for the fun stuff—actually writing your application.

Recipe 2.1 describes how to retrieve a single element reference using CSS selector syntax. This recipe introduces the `Node` object, a façade that provides a consistent, easy-to-use API for working with the DOM. Whenever this book refers to a `Node` object, it is referring to a YUI node as opposed to a native DOM node (unless explicitly stated otherwise).

Recipe 2.2 explains how to manipulate CSS classes. For this common operation, YUI supplies sugar methods that properly handle elements with multiple classes.

Recipe 2.3 demonstrates how to exercise the `Node` API to get and set element attributes.

Recipe 2.4 explains how to use the browser's internal HTML parser to serialize and deserialize string content in and out of the DOM.

Recipe 2.5 covers CSS selectors that return multiple nodes as a `NodeList` object. You can iterate through a `NodeList` and operate on individual `Node` objects, or you can use the `NodeList` to perform bulk operations on every member.

Recipe 2.6 describes how to create new elements in the DOM. This is one of the fundamental operations that enable you to construct and display complex widgets and views.

Finally, Recipes 2.7 and 2.8 describe how to augment the `Node` API itself. This is something of a power-user feature, but it is straightforward enough to consider using in your own applications.

2.1 Getting Element References

Problem

You want to retrieve a reference to an element so that you can manipulate the element further.

Solution

Use a CSS selector with the `Y.one()` method to retrieve a single `Node` reference.

Loading Example 2-1 in a browser displays an unimpressive blank page. The code retrieves a `Node` reference to the demo `<div>` element, but doesn't actually do any work with that reference. Don't worry—in Recipe 2.2, the example actually starts calling `Node` methods, and things will get a little more interesting.

Example 2-1. Getting an element by ID

```
<!DOCTYPE html>
<title>Getting an element by ID</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', function (Y) {
    var demo = Y.one('#demo');
});
</script>
```

Discussion

`Y.one()` is your standard entry point for manipulating the DOM with YUI. Example 2-1 illustrates one of the most common patterns, retrieving a single node by its unique ID. This is like the workhorse DOM method `document.getElementById()`, with two key differences:

- `Y.one()` is shorter.
- `Y.one()` returns a YUI `Node` façade object that wraps the underlying native DOM `Element` object.

However, `Y.one()` doesn't just mimic `document.getElementsById()`. As shown in Example 2-2, `Y.one()` takes any CSS selector, returning the first node that matches. If the selector fails to match any elements, `Y.one()` returns `null`. This enables you to sift through the DOM using familiar CSS selector syntax. This technique wasn't conceived of when the DOM was originally designed, but it has proven to be so useful that most browsers now offer native support.

There is a counterpart to `Y.one()` named `Y.all()`, which returns all nodes that match the selector. For more information, refer to Recipe 2.5.

Example 2-2. Getting an element with various selectors

```
<!DOCTYPE html>
<title>Getting an element with various selectors</title>

<div id="demo" dir="rtl"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', function (Y) {
    var aDiv1 = Y.one('#demo');         // the demo <div>
    var aDiv2 = Y.one('div');           // the demo <div> (the first and only <div>)
```

```
    var aDiv3 = Y.one('body div.bar');  // null; there's no <div class="bar">
    var aScript = Y.one('body script'); // the first <script> in the <body>
    var aDiv4 = Y.one('html > div');    // null; there's no <div> direct child of <html>
    var aDiv5 = Y.one('body > div');    // the demo <div> again
    var aDiv6 = Y.one('div[dir=rtl]');  // the demo <div> one last time
});
</script>
```

If you pass in a CSS selector that matches multiple elements, `Y.one()` returns the first match. By default, YUI keeps its selector engine light by using CSS 2.1 as its baseline. If you need the extra power of CSS3 selectors, load the optional `selector-css3` module.

`Y.one()` returns a YUI `Node` instance, which smooths over browser inconsistencies and offers a more capable interface than the native DOM `Node` and DOM `Element`. Once you have a YUI `Node` reference, you can:

- Add classes to the node by calling `addClass()`, as described in Recipe 2.2
- Hide the node by calling `hide()`, as described in Recipe 3.1
- Remove the node from its parent entirely by calling `remove()`
- Destroy the node, all its children, and remove all its plugins and event listeners by calling `destroy(true)`
- Change the node's properties by calling `set()`, as described in Recipe 2.3
- Move the node on the page by calling `setXY()`, which normalizes element positions to use YUI's unified, cross-browser coordinate system

and perform *many* other operations, as described in the `Node` API.

You can often chain methods when working with `Node`. For example, instead of doing:

```
var demo = Y.one('#demo');
demo.remove();
```

to retrieve the `Node` reference and then remove the node from the document, you can chain these operations:

```
Y.one('#demo').remove();
```

From any given node, you can walk down the DOM tree by calling `one()` or `all()` on the node itself. This returns the first child node (or a list of multiple child nodes) that matches the selector. To walk up the tree, call `ancestor()` or `ancestors()` on the node; this returns the first ancestor node (or a list of multiple ancestor nodes) that matches the selector. To walk sideways, call `next()` or `previous()`. The `next()` and `previous()` methods are like native DOM `nextSibling()` and `previousSibling()`, but they always return a sibling element as a YUI node, and they ignore adjacent text nodes in all browsers.

Alternatively, you can walk the DOM by successively calling `get("parentNode")` or `get("children")`. For more information about `get()`, refer to Recipe 2.3.

The input for `Y.one()` can take a CSS selector string or a native `HTMLElement`. You can use this to create methods that accept flexible input by filtering arguments through `Y.one()`:

```
function foo(node, bar) {
    node = Y.one(node);
    if (node) { ...
```

In addition to the `Node` API methods, a `Node` is also a YUI `EventTarget`. Among other things, this means you can attach event listeners to the element by calling the `on()` method. Chapters 3 and 4 discuss events in detail.

Every YUI node wraps a native DOM object, which you can retrieve by calling `getDOMNode()`, as shown in Recipe 3.6. YUI uses this pattern of wrapping native objects in façade objects throughout the library, in Recipe 2.5, in Chapter 4, and elsewhere.

Use caution when mixing the YUI `Node` API with native DOM operations, particularly destructive native DOM operations. For example, if an event handler holds a YUI `Node` reference, and you destroy the underlying native DOM node with a native `innerHTML` assignment or similar operation, this can lead to memory leaks.

See Also

The `Node` User Guide (*http://yuilibrary.com/yui/docs/node/*).

2.2 Manipulating CSS Classes

Problem

You want to dynamically change one or more classes on an element.

Solution

Call `Node`'s `addClass()` and `removeClass()` methods to add and remove classes without affecting other classes on the element.

Example 2-3 correctly adds and removes classes without clobbering the original `garish` class. Note that you do not need to wrap `addClass()` or `removeClass()` in a `hasClass()` check, as these methods perform this check for you internally.

Example 2-3. Manipulating CSS classes

```
<!DOCTYPE html>
<title>Manipulating CSS classes</title>
<style>
.garish { color: #f00; }
.moregarish { background: #0f0; }
```

```
.ohpleasegodno { text-decoration: blink; overflow-x: -webkit-marquee; }
</style>

<div id="demo" class="garish ohpleasegodno">Things could always be worse...</div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', function (Y) {
    var div = Y.one('#demo');

    div.addClass('moregarish');
    if (div.hasClass('moregarish')) {
        Y.log('Lime green FTW!');
    }
    div.removeClass('ohpleasegodno');
});
</script>
```

Discussion

The native DOM attribute `className` enables you to easily set an element's class to whatever value you like. However, since elements may have multiple classes in any order, blindly getting and setting `className` is usually a bad idea. `hasClass()`, `addClass()`, and `removeClass()` all correctly handle multiple class names on an element.

In addition to those methods, `Node` provides `replaceClass()` for swapping one class for another, and `toggleClass()` for alternately adding and removing a class. All five methods are also available on the `NodeList` API, enabling you to manipulate classes in bulk. For more information, refer to Recipe 2.5.

Alternatively, you can change an element's appearance by calling `setStyle()` to set an individual CSS declaration, or do a mass string assignment of all CSS declarations by calling `setStyles()`. However, `addClass()` is a more powerful technique because it avoids hardcoding presentation information in JavaScript. `setStyle()` is arguably useful as a quick way to toggle an element between `display:block` and `display:none`, but YUI provides sugar methods for hiding elements, as described in Recipe 3.1.

For most widgets and views, you will want to make a distinction between "core" styles and "skin" styles. For example, in a floating overlay or lightbox, `position:absolute` is a core style, and `background-color:silver` is a skin style. Both types of styles should be encapsulated in classes. `setStyle()` is not ideal for manipulating skin styles. In some cases you can use `setStyle()` with core styles, but often it is better to use higher-level methods such as `hide()`, `show()`, and `setXY()`.

When it comes to controlling presentation, you can get even more abstract than calling `addClass()`. For example, all YUI widgets have a `visible` attribute that, when set to `false`, adds the class `yui3-`*`widget name`*`-hidden` to the bounding box. However, this class doesn't include any CSS rules by default; it simply flags the widget as "hidden" and enables you, the designer of the widget, to choose what that means. It could mean `display:none`, an animated fade, minimizing the widget into an icon, or anything else. For more information about widgets, refer to Chapter 8.

2.3 Getting and Setting DOM Properties

Problem

You want to change a link on the fly to point to a new location.

Solution

Use `Node`'s `set()` method to set the link's `href` property, as shown in Example 2-4.

Example 2-4. Setting DOM properties

```
<!DOCTYPE html>
<title>Setting DOM properties</title>

<a id="demo">Quo vadimus?</a>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', function (Y) {
    Y.one('#demo').set('href', 'http://yuilibrary.com');
});
</script>
```

`Node` has an equivalent `get()` method that retrieves the value of a DOM property as a string.

Discussion

The `get()` and `set()` methods are generic YUI methods for viewing and modifying DOM properties. A DOM property is a JavaScript concept that is related to, but not quite the same thing as, an HTML attribute. For example, the native DOM property `src`:

```
someImg.src = 'http://example.com/foo.gif';
```

is related to the HTML attribute `src`:

```
<img src="http://example.com/foo.gif">
```

However, in general the relationship is not one-to-one. An image's DOM properties include (but are *definitely* not limited to): `align`, `alt`, `border`, `className`, `height`, `hspace`, `id`, `innerHTML`, `isMap`, `parentNode`, `src`, `tagName`, `useMap`, `vspace`, and `width`. From this list, you can see that:

- Many properties, such as `src` and `border`, correspond directly to an HTML attribute name
- Some properties, such as `className`, map to an HTML attribute but under a different name
- Other properties, such as `parentNode` and `innerHTML`, have no corresponding HTML attribute
- Some HTML attributes, such `data-*` attributes, currently have no corresponding DOM property

To make matters more confusing, although most JavaScript developers refer to `img.src` and `img.innerHTML` as DOM "properties," the official W3C terminology for `img.src` and `img.innerHTML` is in fact DOM "attributes" (*http://www.w3.org/TR/DOM-Level-2-HTML/html.html#ID-642250288*) .This book refers to them as "properties."

If a property represents a `Node` or a `NodeList`, `get()` retrieves the YUI wrapper object (`Node` or `NodeList`) rather than a native DOM object. Unless you explicitly call `getDOMNode()` to get the underlying native object, YUI always maintains the `Node` façade.

Unfortunately, the exact property list varies from element to element and from browser to browser. Most browsers support a large set of properties defined by the W3C, plus a few useful nonstandard ones. Recent W3C specifications have folded in some popular extensions, such as `innerHTML`.

In addition to `get()` and `set()` for manipulating DOM properties, `Node` also provides `getAttribute()`, `setAttribute()`, and `removeAttribute()` for manipulating HTML attributes. These YUI methods are thin wrappers around the native methods of the same name, but have slightly different semantics:

- Calling `get()` on a DOM property that does not exist returns `undefined`. Calling `set()` on a made-up property has no effect. `get()` and `set()` also support several useful properties that cannot be accessed via `getAttribute()` and `setAttribute()`, such as `text`, `children`, and `options`.
- Calling `getAttribute()` on an HTML attribute that does not exist returns an empty string. This adheres to the W3C DOM specification, correcting for the fact that native browser implementations of `getAttribute()` actually return `null` in this case. Calling `setAttribute()` on a made-up attribute works just fine. This means you can use `getAttribute()` and `setAttribute()` to store string metadata on nodes. However, a better approach would be to use `getData()` and `setData()`, which stores

data on the YUI façade object and avoids disturbing the DOM. You can also use `getAttribute()` and `setAttribute()` (but *not* `get()` and `set()`) to access the new `data-*` HTML5 attributes.

See Also

Recipes 8.7 and 8.8 for working with `data-*` attributes; Recipe 11.4 to see the `setAttrs()` sugar method and the `removeAttribute()` method setting up and tearing down multiple attributes at once; Marko Dugonjić's blog post, "The Difference Between *href* and *getAttribute('href')* in JavaScript" (*http://www.maratz.com/blog/archives/2005/08/29/the-difference-between-href-and-getattributehref-in-javascript/*).

2.4 Changing an Element's Inner Content

Problem

You want to retrieve an element's content and conditionally replace that content with something else.

Solution

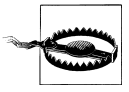
Use `Y.one()` to get a node reference. Then call the `getHTML()` method to inspect the element's inner content as a string, followed by `setHTML()` to change the element's contents. If the demo `<div>` is empty, the script immediately replaces the contents with a different string. See Example 2-5.

Example 2-5. Changing an element's inner content

```
<!DOCTYPE html>
<title>Changing an element's inner content</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', function (Y) {
    var demo = Y.one('#demo'),
        hi = '<em>HELLO</em> from the ' + demo.get('tagName')
            + ' with id=' + demo.get('id');

    if (demo.getHTML() === '') {
        demo.setHTML(hi);
    }
});
</script>
```

The `<em>` and `</em>` tags in the string get parsed and written into the DOM as an `<em>` element. As an alternative to calling `setHTML()`, you can get and set the `innerHTML` DOM property:

```
if (demo.get('innerHTML') === '') {
    demo.set('innerHTML', hi);
}
```



Methods like `setHTML()` and `set('innerHTML')` are insecure when used for non-HTML strings or strings whose actual content or origin is unknown. When you need to guard against unknown content, you can use `set('text')`. See also `Y.Escape`, discussed in Recipe 9.13.

Discussion

First introduced by Internet Explorer, `innerHTML` has long been standard equipment in browsers, and has recently been codified as a standard. `innerHTML` is a powerful feature that grants you direct access to the browser's fast HTML parser, serializing and deserializing strings in and out of the DOM.

Some browsers have buggy implementations of `innerHTML` that behave strangely in cases where HTML has implicit "wrapper" elements. For example, some browsers might fail if you use `innerHTML` to insert a `<tr>` string directly into a `<table>` without a `<tbody>` wrapper. YUI uses feature detection to make `innerHTML` safer to use, adding wrapper elements for browsers that require it.

The `setHTML()` method is a YUI sugar method. It has the same semantics as `innerHTML`, but it first walks old child nodes and cleanly detaches them from the parent. The more aggressive `innerHTML` simply destroys and replaces the old elements. Using `setHTML()` is a bit slower, but it avoids breaking references to the old nodes and prevents memory leaks in old versions of Internet Explorer.

An alternative approach for creating elements is to use `Y.Node.create()` to create individual `Node` objects. For a comparison of `setHTML()`, `Y.Node.create()`, and other methods for modifying the DOM, refer to Recipe 2.6.

2.5 Working with Element Collections

Problem

You want to add the class `highlight` to all list elements within the demo `<div>`.

Solution

Use `Y.all()` to retrieve a `NodeList` containing all list elements that match the criteria. Then call `addClass()` on the `NodeList` to add the class to each member `Node`, as shown in Example 2-6.

Example 2-6. Operating on a collection of elements

```
<!DOCTYPE html>
<title>Operating on a collection of elements</title>
```

```
<style>
.highlight { background: #c66; }
</style>

<div id="demo">
    <ul>
        <li>Apples</li>
        <li>Bananas</li>
        <li>Cherries</li>
    </ul>
    <ol>
        <li>Strawberries</li>
        <li>Tomatoes</li>
    </ol>
</div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', function (Y) {
    var listItems = Y.all('#demo li');
    listItems.addClass('highlight');
});
</script>
```

If you didn't need to reuse the reference to the `NodeList` later on, an even more compact solution would be to replace the two lines of JavaScript with simply:

```
Y.all('#demo li').addClass('highlight');
```

Discussion

In Example 2-1, `Y.one()` retrieved a `Node` instance representing the **one** element matched by the `"#demo"` CSS selector:

```
var demo = Y.one('#demo');
```

Similarly, Example 2-6 uses `Y.all()` to retrieve a `NodeList` instance representing **all** the elements matched by the `"#demo li"` CSS selector:

```
var listItems = Y.all('#demo li');
```

If the selector fails to match any elements, `Y.all()` returns an empty `NodeList`. This enables you to safely do things like `Y.all('script').remove()` to remove all `<script>` elements—if there are no `<script>` elements on the page, nothing happens.

This is deliberately different behavior from `Y.one()`, which returns `null` if no match is found. `Y.one()` is designed to make it easy to perform node existence tests, while `Y.all()` is designed to make it easy to do bulk operations. YUI is interesting in that it has two abstractions for fetching DOM nodes. Some libraries rely on a single abstraction (reaching into the DOM always returns a collection), but the choice of `Y.one()` and `Y.all()` enables you to write cleaner code—you know ahead of time whether you will receive a single node or a collection.

As the suffix "List" implies, `NodeList` is an arraylike collection, providing methods such as `pop()`, `shift()`, `push()`, `indexOf()`, and `slice()`. `NodeList` also includes a subset of the more popular `Node` methods, such as `addClass()`, `on()`, and `remove()`. This enables you to perform bulk operations on every member node in only a few lines of code, without having to manually loop over the `NodeList`'s contents. For example:

- `Y.all(`*`selector`*`).on(`*`type, fn`*`)` attaches an event listener to every element matched by the selector. However, it is often better to use event delegation, described in Recipe 4.5.
- `Y.all(`*`selector`*`).transition(`*`config`*`)` runs a CSS Transitions-based animation on every element matched by the selector. For more information about the YUI `Transition` API, refer to Chapter 3.
- `Y.all(`*`selector`*`).each(`*`fn`*`)` executes an arbitrary function on each element matched by the selector.

The `one()` and `all()` methods are also available on each `Node` object. The difference is that calling `Y.all(`*`selector`*`)` queries the entire document for matches, while calling *`node`*`.all(`*`selector`*`)` restricts the query to search only through that node's descendants. For an example of this in action, refer to Example 2-8.

See Also

Recipe 9.2, which describes some useful static `Y.Array` methods that also work with `NodeLists`.

2.6 Creating New Elements

Problem

You want to create a new element and add it to the document.

Solution

Retrieve a `Node` instance and call `append(`*`child`*`)` to add the new node to the document as a child of the selected node, as shown in Example 2-7.

Example 2-7. Creating a new element and adding it to the DOM

```
<!DOCTYPE html>
<title>Creating a new element and adding it to the DOM</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', function (Y) {
    Y.one('#demo').append("<h1>Don't Forget the Heading!</h1>");
```

```
});
</script>
```

Alternatively, create the node and then call `appendTo(selector)` on the new node:

```
var heading = Y.Node.create("<h1>Don't Forget the Heading!</h1>");
heading.appendTo('#demo');
```

Discussion

Creating new elements is a key operation in any web application. For example, if you inject a visible widget onto a page, the widget is responsible for bootstrapping itself into existence by creating, modifying, and appending elements into the document as needed.

Many developers use a design strategy called *progressive enhancement* to ensure that their pages provide at least basic functionality when JavaScript is turned off or broken. The idea is to first provide a working skeleton of static markup and only *then* enhance the page's behavior with JavaScript. Certain types of projects such as games, bookmarklets, or internal business applications might not need progressive enhancement, but in general, failing to follow this strategy can lead to costly errors. YUI includes several patterns that directly or indirectly support progressive enhancement, such as feature detection and `Widget`'s `HTML_PARSER` attribute, described in Recipe 7.5.

Even today, many older tutorials and scripts rely on the `document.write()` method, which compiles strings into elements and writes those elements into the DOM as the document is loading. Calling `document.write()` after the document has loaded wipes out and replaces the entire page content, which can lead to surprising bugs. Calling `document.write()` before the document has loaded makes it difficult for the browser to optimize how it fetches resources and renders the page.

YUI's `Node` API provides much better approaches than `document.write()` for creating new elements. These approaches fall into several families:

- The static `Y.Node.create()` method, which creates a new node disconnected from the document. This is the workhorse method for creating `Y.Node` objects in YUI.
- `cloneNode()`, which can create a shallow copy of a `Y.Node` (only copy the open and close tags) or a deep copy (copy all attributes and internal contents). Cloning is a useful optimization when you need to create several similar nodes: use `Y.Node.create()` to create a template node, and then clone the template. Like `Y.Node.create()`, cloned nodes are created outside the document.
- `setHTML()` and the `innerHTML` DOM property, discussed in Recipe 2.4. These methods use the browser's HTML parser to compile a string into elements and insert those elements into the DOM all in one step, completely replacing the element's inner contents.

Although `setHTML()` and `innerHTML` might seem superficially similar to `document.write()`, these approaches are scoped to an individual element and are fine to use after the document has loaded.

- `appendChild()`, `insertBefore()`, and `replaceChild()`. These are YUI DOM façade methods. They act like the similarly named native DOM methods, but return YUI `Node` objects. Use these methods if you feel more comfortable working with an API that looks more like the DOM.
- `append()`, `prepend()`, `insert()`, and `replace()`. These are YUI sugar methods. In addition to having shorter names, they are also chainable. These methods can either attach existing `Node` objects, or compile strings into objects and then attach the results.

`Y.Node.create()` and `cloneNode()` involve a two-step process: first you create the `Node` objects you want, and then you assemble them into a tree and add them to the document with `append()` or a similar method. Appending a tree of `Node` objects into the document makes them visible, but requires the browser to run an expensive reflow and repaint operation. It is therefore best to use `append()` "off document" to completely assemble a `Node` structure, then perform a final `append()` to add the entire structure into the document in one operation.

The other approach is to pass raw strings of HTML into `innerHTML`, `setHTML()`, or `append()` and its cousins. It can be very efficient to serialize strings directly into the DOM without needing to mess with intermediate `Node` objects. However, if you want to manipulate the nodes later, you must then incur the overhead of flagging the markup to be locatable (by adding classes and IDs) and calling `Y.one()` or `Y.all()` to get node references. The more methodical `Y.Node.create()` ensures that you already have references to everything you need. Often, the choice between compiling and parsing HTML strings versus assembling nodes as objects boils down to which approach yields the cleanest code.

See Also

Mike Davies on the costs of ignoring progressive enhancement (*http://isolani.co.uk/blog/javascript/BreakingTheWebWithHashBangs*).

2.7 Adding Custom Methods to Nodes

Problem

You want to be able to determine whether an individual node contains one of the new elements that were added in the HTML5 specification.

Solution

Use `Y.Node.addMethod()` to add a `hasHTML5()` method to all `Node` objects, as shown in Example 2-8. `addMethod()` takes three arguments: the string name of the method to bind to `Node`, a function that actually becomes the method, and an optional context with which to call the method.

To determine whether an element is new in HTML5:

1. Define a string that lists all HTML5 element names. This string serves as a CSS selector.
2. Use `Node.all()` to return a `NodeList` of all child elements that match this selector. Within `addMethod()`, the `this` object refers to the YUI node where the method is operating, and the `domNode` parameter represents the native DOM object that underlies the YUI node.
3. Return `true` if any HTML5 elements were found; `false` otherwise.

Example 2-8. Adding the hasHTML5() method

```
<!DOCTYPE html>
<title>Adding the hasHTML5() method</title>

<article id="demo"><p>Hello</p></article>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', function (Y) {
    Y.Node.addMethod('hasHTML5', function(node) {
        var html5elements = 'article, aside, audio, bdi, canvas, command, ' +
            'datalist, details, figcaption, figure, footer, header, ' +
            'hgroup, keygen, mark, meter, nav, output, progress, rp, rt, ' +
            'ruby, section, source, summary, time, video, wbr';

        return (this.one(html5elements) !== null);
    });

    Y.log(Y.one('#demo').hasHTML5());
    Y.log(Y.one('body').hasHTML5());
});
</script>
```

`Y.log()` logs debug messages to the browser console. For more information, refer to Recipe 12.1.

Strictly speaking, most HTML 4.01 elements are also HTML5 elements, so perhaps this method should have been named `hasNewInHTML5()` or the even more horrible `hasElementNewInHTML5()`.

Discussion

Having a `Node` façade makes it straightforward to augment the DOM. The YUI team uses `Node` to make it easy to fix browser bugs, normalize browser behaviors, and add features. There is no reason why you can't also use this abstraction layer for your own purposes.

When you use `addMethod()`, any values you return from your function automatically get normalized to maintain the façade:

- If you return a native DOM node, `addMethod()` wraps it as a YUI node.
- If you return a native DOM collection or array, `addMethod()` wraps it as a YUI `NodeList`.
- If you return some other value (other than `undefined`), the value passes through unaltered.
- If you declare no return value, `addMethod()` returns the underlying `Node` instance, which enables your method to be chained.

There is also an equivalent `Y.NodeList.addMethod()` for augmenting `NodeList`s. Any method you add in this manner will automatically get iterated over the `NodeList`'s members when that method is called.

Augmenting `Node` with new methods is probably the kind of thing you should bundle into a module for reuse. For more information, refer to Recipe 1.8.

2.8 Adding Custom Properties to Nodes

Problem

To celebrate International Talk Like a Pirate Day (September 19), you want to create a custom property that provides the pirate-speak version of the element's text.

Solution

Add the property to `Y.Node.ATTRS`. A custom property is an object that contains a `getter` function, and optionally a `setter` function if the property is writable.

In Example 2-9, the `getter` function uses a simple object as a map for replacing English words with pirate-speak. It acts by:

1. Getting the `Node`'s `text` property, which represents the plain-text content. The `text` property is a YUI abstraction over browser native properties such as `innerText` and `textContent`. (Within `Y.Node.ATTRS`, the `this` object refers to the node in question.)
2. Creating an object to serve as a mapping between English words and pirate words. The mapping in Example 2-9 is pretty short; you should feel free to expand it.

3. Performing a `String.replace()` on the normal text. The regular expression matches all words in the string. For each word matched, `replace()` calls a function that replaces the word with a pirate word or leaves the word alone, depending on the contents of the map.
4. Returning the pirate text, with a bonus "Arrrr!" thrown in.

To prove that the property works, the example sets the demo `<div>`'s content to an English sentence, then immediately turns around and uses the `pirate` property to change the `<div>` to the pirate-speak equivalent.

Example 2-9. Adding a read-only pirate property

```
<!DOCTYPE html>
<title>Adding a read-only pirate property</title>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', function (Y) {
    var demo = Y.one('#demo');

    Y.Node.ATTRS.pirate = {
        getter: function() {
            var normalText = this.get('text'),
                pirateMap = {'hello':'ahoy', 'my':'me', 'the':"th'",
                    'you':'ye', 'to':"t'", 'is':'be', 'talk':"be talkin'"},
                pirateText = normalText.replace(/\b\w*\b/g, function (word) {
                    return pirateMap[word] || word;
                });

            return pirateText + ' Arrrr!';
        }
    };

    demo.setHTML('It is fun to talk like a pirate!');
    demo.setHTML(demo.get('pirate'));
});
</script>
```

Discussion

Although converting an element's text to pirate-speak is an admittedly silly example, there are all sorts of other simple text transformations that you might want to implement as custom properties. For example, the ROT13 transformation replaces each letter with the letter 13 places further in the alphabet: *a* (letter 1) becomes *n* (letter 14), *u* (letter 21) wraps around to *h* (letter 8), and so on. ROT13 is useful for obscuring joke punchlines, plot spoilers, and answers to puzzles. Other possibly interesting transformations on node text include disemvowelling, reversing text, pretty printing code in `<pre>` elements, and more.

Custom properties also don't necessarily have to revolve around the element's text content. For example, you could have designed the `hasHTML5()` method from Recipe 2.7 as a custom property with a `getter` function.

As with the `hasHTML5()` method described in Recipe 2.7, you should probably register the `pirate` property in a custom YUI module using `YUI.add()`. For more information, refer to Recipe 1.8.

Keep in mind that custom YUI node properties do not end up as custom properties in the DOM, so you can access them only by calling `get()` and `set()` on the YUI node. You cannot retrieve custom properties by accessing `myNode.myProperty`, either on the YUI node or the underlying native DOM node.

CHAPTER 3
UI Effects and Interactions

Originally, JavaScript provided only small niceties like the occasional animation, fade, or rollover. Today's browsers have advanced to the point where it is possible to build and maintain sophisticated applications. But sometimes, the small niceties are all you need.

Many YUI developers are frontend engineers who tend to take an application-centric view of the code they are writing. The amount of work that goes into JavaScript dominates all else, and the HTML and CSS is something to tweak later or (hopefully) hand off to a designer.

But if you're a designer who codes, or you're working on a very content-heavy page, it might be the HTML and CSS that dominates. In this more page-centric view, the HTML page needs only some snippets of JavaScript to sprinkle in some user interface effects.

To support this kind of use case, YUI enables you to add interesting UI effects with little overhead. You can make an element draggable in one line of code. You can fade an element in response to a click in just three lines of code. You can perform an interesting sequence of animations in only a few lines of code.

In some ways, YUI is actually better suited for the page-centric world than you might expect. If the JavaScript is meant to be a light cosmetic addition to the page, then the worst thing you can do is load a huge monolithic library just to do a fade. YUI's flexible module system enables you to be far more selective about which components you load to create a particular effect.

Recipe 3.1 demonstrates how to hide and show an element immediately.

Recipe 3.2 introduces a slightly fancier approach to hiding and showing, by explaining how to gracefully fade an element in and out of visibility.

Recipe 3.3 builds on these concepts by introducing the YUI `Transition` API, which makes it easy to do basic DOM animations.

Recipe 3.4 demonstrates a more complicated transition that animates multiple properties on independent timers, and chains a second transition after the first.

Recipe 3.5 describes how to register slide effects and other custom transitions with YUI under a string name.

Recipe 3.6 uses the handy `Y.DOM.inViewportRegion()` to create a simple "infinite scroll" interaction.

Recipe 3.7 explains how to make an element draggable and several variations on this behavior. It also covers the concept of plugins, which are discussed in more detail in Chapter 8.

Recipe 3.8 introduces the `resize` module, an extension of the Drag and Drop (`DD`) API that makes elements and widgets resizable.

Recipe 3.9 provides a complete working example of a table with rows that you can reorder by dragging. In addition to showing how to use drop targets, this example introduces some new event-related concepts, such as inspecting the event object and using a central event manager for event handlers.

3.1 Hiding an Element

Problem

When the user clicks a button, you want to hide an element from view.

Solution

Use `Y.one()` to get the `<button>` and the `<div>` as a YUI node. Then add a `click` event listener to the button using `Node`'s `on()` method. When the user clicks the button, the callback function executes and calls `hide()` on the demo `<div>`. See Example 3-1.

Example 3-1. Hiding an element in response to a click

```
<!DOCTYPE html>
<title>Hiding an element in response to a click</title>
<style>
#demo { width: 100px; height: 100px; border: 1px #000 solid; background: #d72; }
</style>

<button id="hide">Hide</button>
<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', function (Y) {
    var hideButton = Y.one('#hide'),
        demo = Y.one('#demo');

    hideButton.on('click', function () {
        demo.hide();
    });
```

```
});
</script>
```

If you use `Y.one()` inline, the code is even more compact:

```
YUI().use('node', function (Y) {
    Y.one('#hide').on('click', function () {
        Y.one('#demo').hide();
    });
});
```

Discussion

Not surprisingly, YUI offers a `show()` method as a counterpart to `hide()`. If you need to hide *and* show the `<div>`, add a second HTML `<button>` with an `id` of `show`. Then add another event listener:

```
YUI().use('node', function (Y) {
    Y.one('#hide').on('click', function () {
        Y.one('#demo').hide();
    });

    Y.one('#show').on('click', function () {
        Y.one('#demo').show();
    });
});
```

Technically speaking, you can always set `display: none` yourself by setting the style, as described in Chapter 2. Calling `hide()` is more elegant than setting the style manually, and it also offers some extra functionality as described in Recipe 3.2.

Nearly all page effects are triggered by some sort of *event*—a button click, a mouseover, or some other action by the user or the system. Most of the examples in this chapter use the `on()` method for setting an event listener on that node. This method takes a string representing the type of the event (such as `'click'` or `'mouseover'`) and a function to execute when the event occurs.

The `node` rollup exposes a small amount of event functionality via `on()`, but to get the full power of the YUI `Event` API, you must use `event-base` and related modules, or pull in the `event` rollup. For much more information about how the YUI event system works in general, refer to Chapter 4.

3.2 Fading an Element

Problem

You want to make an element disappear a little more gracefully, as having it disappear immediately is kind of jarring.

Solution

Use the `hide()` method as before, but load the `transition` module as well. Among other features, the `transition` module augments `hide()` and `show()` with additional functionality, enabling you to fade the element in and out by passing `true` to `hide()` and `show()`. See Example 3-2.

Example 3-2. Fading an element in response to a click

```
<!DOCTYPE html>
<title>Fading an element in response to a click</title>
<style>
#demo { width: 100px; height: 100px; border: 1px #000 solid; background: #d72; }
</style>

<button id="hide">Hide</button> <button id="show">Show</button>
<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', 'transition', function (Y) {
    Y.one('#hide').on('click', function () {
        Y.one('#demo').hide(true);
    });

    Y.one('#show').on('click', function () {
        Y.one('#demo').show(true);
    });
});
</script>
```

Passing `true` to `hide()` and `show()` without loading the `transition` module has no effect.

Discussion

The `transition` module enables you to perform simple animations. It relies on the API defined by the CSS3 Transitions specification, which describes how to change CSS values over time. The YUI `transition` module presents you with the same API regardless of whether the browser supports CSS3 Transitions natively.

With the `transition` module enabled, `hide()` and `show()` support three optional arguments. You can invoke a fade by passing in `true` as the first argument to `hide()`. This activates a default transition that fades the element over a period of 0.5 seconds.

You can also change the behavior of the default fade. For example, you can change the duration by passing in a configuration object instead of `true`:

```
YUI().use('node', 'transition', function (Y) {
    Y.one('#hide').on('click', function () {
        Y.one('#demo').hide({ duration: 2.0 });
    });
```

```
        Y.one('#show').on('click', function () {
            Y.one('#demo').show({ duration: 1.5 });
        });
    });
```

This stretches the duration to 2.0 seconds for the `hide()`, and 1.5 seconds for the `show()`. In general, you can pass in:

- `true`, as in `hide(true)`. This triggers `hide()`'s default transition. It is sugar for `hide('fadeOut')`.
- A string name for a predefined transition, as in `hide('fadeOut')`. In addition to `fadeIn` and `fadeOut`, YUI also ships with a `sizeIn` and `sizeOut` transition, which means you can call `hide('sizeOut')` to shrink an element to oblivion. For more information about how to register your own custom transitions, refer to Recipe 3.5.
- An arbitrary transition object. For examples, refer to Recipes 3.3 and 3.4.

You can also provide a callback function to execute when the transition completes, as shown in Example 3-3. For example, when the `hide()` completes, you can remove the element from the DOM entirely by calling `remove()` on the node. As long as you save the node reference, you can still reverse the `hide()` operation by inserting the node back into the DOM and then calling `show()`.

Example 3-3. Fading and removing an element in response to a click

```
YUI().use('node', 'transition', function (Y) {
    var demo = Y.one('#demo');

    Y.one('#hide').on('click', function () {
        demo.hide({ duration: 2.0 }, function () {
            demo.remove();
        });
    });

    Y.one('#show').on('click', function () {
        Y.one('#show').insert(demo, 'after');
        demo.show({ duration: 1.5 });
    });
});
```

The code is not symmetric—the node gets removed in a callback for `demo.hide()`, but it gets reinserted just before calling `demo.show()`. If you tried to make the `show()` code mirror the `hide()` code, then the `<div>` would appear to pop into existence after 1.5 seconds, which is not the desired effect.

See Also

Recipe 3.5; the CSS3 Transitions specification (*http://www.w3.org/TR/css3-transitions/*).

3.3 Moving an Element

Problem

You want to animate an element and move it across the page.

Solution

Set the element's CSS `position` property to `absolute`. Then load the `transition` module and call the node's `transition()` method, passing in a configuration object. As you can see in Example 3-4, the configuration object includes these properties:

`delay`
: An optional delay in seconds before starting the transition.

`duration`
: The time in seconds to run the transition.

`easing`
: The optional name of a predefined mathematical function for controlling the element's acceleration.

`left`
: The final state of the element's `left` CSS property. You can animate a large number of CSS properties, including the size, position, text color, and more.

Example 3-4. Moving an element across the screen

```
<!DOCTYPE html>
<title>Moving an element across the screen</title>
<style>
#demo {
    width: 100px; height: 100px; border: 1px #000 solid; background: #d72;
    position: absolute;
}
</style>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', 'transition', function (Y) {
    Y.one('#demo').transition({
        delay: 1.0,
        duration: 2.0,
        easing: 'ease-in',
        left: '500px'
    });
});
</script>
```

Discussion

Calling `hide(true)` is convenient for simply fading an element. For more general access to CSS3 Transitions, use the `transition()` method.

JavaScript has always had timers and DOM manipulation, so it's easy to think about creating basic animations by changing an element in many small steps over a certain timeframe. But the devil is in the details. JavaScript timers are unreliable over short time slices. Constant DOM repaints are expensive and compete with myriad other tasks the browser might be trying to do. In short, creating robust, nonjittery animations in older browsers is not easy. CSS3 Transitions greatly simplifies DOM animations by natively handling many of these fiddly details for you.

Naturally, the YUI `Transition` API has you covered either way. For all browsers, YUI presents a consistent, friendly interface for configuring transitions. If the browser does not support transitions natively, YUI loads additional fallback code that implements the API in pure JavaScript.

The basic concept of CSS transitions is that over a certain duration, using a certain *easing function*, the `transition()` method transitions an element from one CSS state to another. The easing function, also known as a *transition timing function*, controls how the element accelerates from one CSS state to another over the specified time period. Here, the concept of "acceleration" doesn't just apply to the element's position. An element transitioning from red to green could stay red for most of the transition, then quickly accelerate into green—or vice versa. For a complete list of available timing functions and CSS properties you can animate, refer to the CSS3 Transitions specification.

Example 3-4 starts the animation one second after the page loads. You can, of course, trigger this from an event instead:

```
YUI().use('node', 'transition', function (Y) {
    var demo = Y.one('#demo');

    demo.on('mouseover', function () {
        demo.transition({
            duration: 2.0,
            easing: 'ease-in',
            left: '500px'
        });
    });
});
```

The `transition()` method can also animate different CSS properties independently. It is even possible to chain transitions together, as shown in Recipe 3.4.

To simply jump an element to a new location without any animation, use the `setXY()` method. `setXY()` works on elements regardless of whether you remembered to set the CSS `position`.

See Also

The CSS3 Transitions specification's sections on transition timing functions (*http://www.w3.org/TR/css3-transitions/#transition-timing-function_tag*) and animatable properties (*http://www.w3.org/TR/css3-transitions/#animatable-properties-*).

3.4 Creating a Series of Transitions

Problem

You want to perform a series of transitions that work together to create an effect.

Solution

Use durations and delays to animate CSS properties independently. In the configuration object, you can specify CSS properties as simple values. But if you specify a CSS property as an object, each CSS property can have its own delay, duration, and easing function that overrides the default.

In addition to playing tricks with durations and delays, Example 3-5 chains a second transition after the first one by creating an `on` object with an `end` function. YUI calls this event handler function after all animations in the first transition finish.

Example 3-5. A series of transitions

```
<!DOCTYPE html>
<title>A series of transitions</title>
<style>
#demo {
    width: 200px; height: 200px; border: 1px #000 solid; background: #d72;
    position: absolute;
    text-align: center;
    opacity: 0.3;
}
</style>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('node', 'transition', function (Y) {
    var demo = Y.one('#demo');

    demo.transition({
        duration: 2.5,
        width: '100px',
        height: '100px',
        left: {
            easing: 'ease-in',
            value: '500px'
        },
```

```
        opacity: {
            delay: 1.0,
            duration: 1.75,
            value: 1.0
        },
        on: {
            start: function () {
                demo.setHTML("It's just a jump to the left...");
            },
            end: function () {
                demo.setHTML('And then a step to the riiight!');
                demo.transition({
                    duration: 2.0,
                    left: '0px',
                    easing: 'linear'
                });
            }
        }
    });
});
</script>
```

Discussion

Moving the `<div>` back and forth across the screen requires changing the `left` CSS property twice. Since you can't define the same property twice in the same configuration object, this requires calling another `transition()` function.

Naively, you might try chaining the second `transition()` immediately after the first, as in: `node.transition({...}).transition({...})`. Here the second `transition()` function gets called almost immediately after the first, so the two animations clobber each other. Instead, set the second `transition()` in an `end` callback, as shown in the example. This ensures that the second `transition()` picks up properly where the first one leaves off.

You can also use `start` and `end` to set and remove extra CSS properties that you need for the transition, such as `overflow` or `position`. For an example of this, refer to Recipe 3.5.

Use caution when trying out complex transitions in older browsers. The fallback code handles simple transitions smoothly, but more complex series of transitions can cause jumps and jitters. As mentioned in Recipe 3.3, emulating CSS transitions in pure JavaScript is inherently less precise than the real thing.

3.5 Defining Your Own Canned Transitions

Problem

You have a standard animation configuration that you want to use over and over, but passing the full config to `transition()` each time is cumbersome.

Solution

Append your named transition to the `Y.Transition.fx` object. Example 3-6 adds two named transitions: a `slideFadeIn` for hiding elements, and a `slideFadeOut` for reversing the operation. This registers the transition and enables you to refer to it by name, like `fadeIn` and `fadeOut`.

For this particular slide effect, the element must have its CSS `position` set to `relative`. The example uses the `start` callback to just clobber the element's `position` property, whatever it might be. A more sophisticated transition could be more careful about applying and removing this property.

Example 3-6. Defining a named slideFadeOut transition

```
<!DOCTYPE html>
<title>Defining a named slideFadeOut transition</title>
<style>
#demo { width: 100px; height: 100px; border: 1px #000 solid; background: #d72; }
</style>

<button id="hide">Hide</button>
<button id="show">Show</button>
<div id="demo"></div>

<script src='http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js'></script>
<script>
YUI().use('node', 'transition', function (Y) {
    function setRelativePosition() {
        this.setStyle('position', 'relative');
    }

    Y.Transition.fx.slideFadeOut = {
        opacity: 0,
        right: '-100px',
        easing: 'ease-out',
        on: { start: setRelativePosition }
    };

    Y.Transition.fx.slideFadeIn = {
        opacity: 1.0,
        right: '0px',
        easing: 'ease-in',
        on: { start: setRelativePosition }
    };

    Y.one('#hide').on('click', function () {
        Y.one('#demo').hide('slideFadeOut');
    });

    Y.one('#show').on('click', function () {
        Y.one('#demo').show('slideFadeIn');
    });
});
</script>
```

Discussion

Registering a named transition makes it easy to reuse that code, particularly for people with less JavaScript expertise than you have. If you are a frontend engineer or lead prototyper, `Y.Transition.fx` makes it easy to register a whole host of canned transitions for other designers and prototypers on your team to use.

Example 3-6 demonstrates using named transitions in `hide()` and `show()`, but you can always use named transitions in a general `transition()` call as well.

To simplify things even further, you can redefine the default transition behavior of `hide()` and `show()`, as Example 3-7 illustrates. This enables your team to simply call `hide(true)` without having to care whether this causes a fade, resize, slide, or something more complex.

Example 3-7. Redefining the default hide and show transition

```
YUI().use('node', 'transition', function (Y) {
    function setRelativePosition() {
        this.setStyle('position', 'relative');
    }

    Y.Transition.fx.slideFadeOut = {
        opacity: 0,
        right: '-100px',
        easing: 'ease-out',
        on: { start: setRelativePosition }
    };

    Y.Transition.fx.slideFadeIn = {
        opacity: 1.0,
        right: '0px',
        easing: 'ease-in',
        on: { start: setRelativePosition }
    };

    Y.Transition.HIDE_TRANSITION = 'slideFadeOut';
    Y.Transition.SHOW_TRANSITION = 'slideFadeIn';

    Y.one('#hide').on('click', function () {
        Y.one('#demo').hide(true);
    });

    Y.one('#show').on('click', function () {
        Y.one('#demo').show(true);
    });
});
```

3.6 Creating an Infinite Scroll Effect

Problem

You want to create an "infinite scroll" interaction that appends new results as the user scrolls down the page.

Solution

Load the `dom` module, which provides the `Y.DOM.inViewportRegion()` method. Then define two functions: `addContent()`, which is responsible for adding new content to the page, and `fillToBelowViewport()`, which is responsible for calling `addContent()` until the last paragraph is no longer in the viewport.

Then add a `scroll` event listener that calls `fillToBelowViewport()` as the user scrolls. Finally, call `addContent()` to initially populate the page, followed by `fillToBelowViewport()` to guarantee that the viewport starts out overfilled. The initial `fillToBelowViewport()` might do nothing, depending on the size of the user's screen.

One slightly tricky aspect to `inViewportRegion()` is that `Y.DOM` is designed to work independently of YUI `Node`, which means its methods all operate on native `HTMLElement` objects. For convenience, Example 3-8 loads the `Node` API anyway. The `scroll` listener uses `Y.one()` to fetch a `Node` instance, and then calls `getDOMNode()` to get the underlying native `HTMLElement` object, to be passed into `Y.DOM.inViewportRegion()`.

The YUI `Node` API also has a handy `generateID()` method, which the example uses to generate a unique ID on the last paragraph. Every time new content gets added, a new ID gets saved as a handle for use in the `scroll` listener.

Example 3-8. Creating an infinite scroll effect

```
<!DOCTYPE html>
<title>Creating an infinite scroll effect</title>
<style>
p { font-family: courier; color: #333; }
</style>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('dom-core', 'node', function (Y) {
    var lastParaId;

    function addContent(numParas) {
        var i, content = '',
            para = '<p>All work and no play makes Jack a dull boy.</p>';

        for (i = 0; i < numParas; i += 1) {
            content += para;
        }
```

```
        Y.one('#demo').append(content);
        return Y.one('#demo p:last-child').generateID();
    }

    function fillToBelowViewport() {
        var lastPara = Y.one('#' + lastParaId).getDOMNode();
        if (Y.DOM.inViewportRegion(lastPara)) {
            lastParaId = addContent(10);
        }
    }

    Y.on('scroll', fillToBelowViewport);

    lastParaId = addContent(20);
    fillToBelowViewport();
});
</script>
```

Discussion

Example 3-8 is the skeleton of an infinite scroll interaction. Most real-world infinite scrolls use Ajax to fetch new content. Since Ajax requests can take a noticeable amount of time, you could add a spinner or some other animation to indicate that the page is fetching more data. You could also improve perceived performance by fetching Ajax data a bit earlier, perhaps by triggering off of an element a few positions *above* the last paragraph or by tracking scroll velocity.

`Y.DOM` contains a few methods for creating elements and manipulating classes, which means that in a pinch, you can use it as a lightweight substitute for the full YUI `Node` API. However, it is really more useful for doing things like checking whether an element is in a certain region or whether two elements intersect.

See Also

The `ImageLoader` User Guide (*http://yuilibrary.com/yui/docs/imageloader/*); YUI DOM API documentation (*http://yuilibrary.com/yui/docs/api/classes/DOM.html*).

3.7 Dragging an Element

Problem

You want to enable users to drag an element around the screen.

Solution

The easiest way to make an element draggable is to load the `dd-drag` module, create a new `Y.DD.Drag` instance, and configure that instance to work on a particular node, as shown in Example 3-9.

Example 3-9. Creating a draggable node

```
<!DOCTYPE html>
<title>Creating a draggable node</title>
<style>
#demo { width: 100px; height: 100px; border: 1px #000 solid; background: #d72; }
</style>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('dd-drag', function (Y) {
    var dd = new Y.DD.Drag({ node: '#demo' });
});
</script>
```

Alternatively, you can load the `dd-plugin` module and plug the `Y.Plugin.Drag` plugin into the `Node` instance, as shown in Example 3-10. Every node exposes a method named `plug()` that can augment that node with additional behavior. Plugins enable you to add behavior to a YUI object in a reversible, nondestructive way.

Example 3-10. Creating a draggable node using a plugin

```
YUI().use('dd-plugin', function(Y) {
    Y.one('#demo').plug(Y.Plugin.Drag);
});
```

In YUI, a *plugin* is a specialized object designed to augment or change the behavior of another object. YUI has a specific interface for consuming plugins (the `plug()` and `unplug()` methods), and a dedicated API for writing plugins. For more information, refer to Recipes 7.7 and 7.8.

Discussion

When you create a `DD.Drag` instance, you can configure the drag behavior by passing a configuration object into the constructor. For example, if you want the element to be draggable only by a `<p>` handle within the `<div>`, you can configure that by setting the `handles` attribute, as Example 3-11 shows.

Example 3-11. Creating a draggable node with a handle

```
<!DOCTYPE html>
<title>Creating a draggable node with a handle</title>
<style>
#demo { width: 100px; height: 100px; border: 1px #000 solid; background: #d72; }
#demo p { margin: 0px; padding 3px; border-bottom: 1px #000 solid; background: #e9e; }
</style>

<div id="demo"><p>handle</p></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
```

```
<script>
YUI().use('dd-drag', function (Y) {
    var dd = new Y.DD.Drag({
        node: '#demo',
        handles: ['p']
    });
});
</script>
```

In addition to drag functionality, a `DD.Drag` instance gains new methods such as `addHandle()` and `stopDrag()`. For example, an equivalent to Example 3-11 would be to create the `DD.Drag` instance, then call the `dd.addHandle()` method:

```
YUI().use('dd-drag', function (Y) {
    var dd = new Y.DD.Drag({ node: '#demo' });
    dd.addHandle('p');
});
```

While `DD.Drag` defines a particular set of dragging functionality, you can change its behavior by loading yet more modules and plugging plugins into the drag instance.

For example, by default the dragged element follows your mouse or finger around the screen. To change the behavior so that the element stays in place and a "ghost" proxy element follows the pointer around instead, load the `dd-proxy` module and plug the drag instance with `Plugin.DDProxy`, as shown in Example 3-12.

Example 3-12. Creating a draggable-by-proxy node

```
<!DOCTYPE html>
<title>Creating a draggable-by-proxy node</title>
<style>
#demo { width: 100px; height: 100px; border: 1px #000 solid; background: #d72; }
</style>

<div id="demo"></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('dd-drag', 'dd-proxy', function (Y) {
    var dd = new Y.DD.Drag({ node: '#demo' });
    dd.plug(Y.Plugin.DDProxy);
});
</script>
```

You can also use a plugin to constrain the draggable area, as shown in Example 3-13. (By default, the user can drag the element anywhere on the screen.) To constrain a draggable element inside a container element, load the `dd-constrain` module, plug the instance with the `Plugin.DDConstrained` plugin, and configure `Plugin.DDConstrained` to use the `box` `<div>` as the container.

Example 3-13. Creating a constrained draggable node

```
<!DOCTYPE html>
<title>Creating a constrained draggable node</title>
<style>
#demo { width: 100px; height: 100px; border: 1px #000 solid; background: #d72; }
#box  { width: 400px; height: 300px; border: 1px #000 dashed; background: #ccc; }
</style>

<div id="box"><div id="demo"></div></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('dd-drag', 'dd-constrain', function (Y) {
    var dd = new Y.DD.Drag({ node: '#demo' });
    dd.plug(Y.Plugin.DDConstrained, { constrain2node: '#box' });
});
</script>
```

Plugins are powerful because you can mix and match them for different situations. Example 3-14 combines the functionality of Examples 3-12 and 3-13 to create a constrained draggable-by-proxy node.

Example 3-14. Creating a constrained draggable-by-proxy node

```
<!DOCTYPE html>
<title>Creating a constrained draggable-by-proxy node</title>
<style>
#demo { width: 100px; height: 100px; border: 1px #000 solid; background: #d72; }
#box  { width: 400px; height: 300px; border: 1px #000 dashed; background: #ccc; }
</style>

<div id="box"><div id="demo"></div></div>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('dd-drag', 'dd-proxy', 'dd-constrain', function (Y) {
    var dd = new Y.DD.Drag({ node: '#demo' });
    dd.plug(Y.Plugin.DDProxy);
    dd.plug(Y.Plugin.DDConstrained, { constrain2node: '#box' });
});
</script>
```

3.8 Creating a Resizable Node

Problem

You want to enable users to resize a node by dragging its edges and corners.

Solution

Make sure the node has a CSS `position` of `relative`, then plug it with `Y.Plugin.Resize`, as shown in Example 3-15.

Example 3-15. Making an element resizable

```
<!DOCTYPE html>
<title>Making an element resizable</title>
<style>
#demo {
    width: 100px; height: 100px; border: 1px #000 solid; background: #d72;
    position: relative;
}
</style>

<div id="demo"></div>

<script src='http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js'></script>
<script>
YUI().use('resize', function (Y) {
    var resize = new Y.Resize({ node: '#demo' });
});
</script>
```

Similar to Drag and Drop, an alternative to using the plugin approach is to create a new `Resize` instance and configure it to work on a particular node:

```
YUI().use('resize-plugin', function (Y) {
    Y.one('#demo').plug(Y.Plugin.Resize);
});
```

Discussion

The `Resize` API uses the Drag and Drop API under the hood and has similar semantics. You can use `Resize` as a plugin to a node or widget, or use it as a standalone instance. Also like Drag and Drop, `Resize` supports resize constraints and resizing by proxy. For instance, Example 3-16 uses a "plug the plugin" approach to constrain the resize to a width between 50 and 200 pixels. The height is unconstrained. (That's right—plugins are themselves pluggable.)

Example 3-16. Creating a constrained resizable node

```
YUI().use('resize-plugin', 'resize-constrain', function (Y) {
    var demo = Y.one('#demo');
    demo.plug(Y.Plugin.Resize);
    demo.resize.plug(Y.Plugin.ResizeConstrained, {
        minWidth: 50,
        maxWidth: 200
    });
});
```

When a user resizes an element, you can also listen for resize events that bubble up to the `Resize` instance (not the node the resize is acting on). Here, it's a little more convenient to create an explicit `Resize` instance rather than plugging the node. Example 3-17 illustrates how to toggle the node's appearance when the user starts and stops the resize.

Example 3-17. Responding to resize events

```
<!DOCTYPE html>
<title>Responding to resize events</title>
<style>
#demo {
    width: 100px; height: 100px; border: 1px #000 solid; background: #d72;
    position: relative;
}
#demo.resizing { background: #27d; }
</style>

<div id="demo"></div>

<script src='http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js'></script>
<script>
YUI().use('resize', function (Y) {
    var resize = new Y.Resize({ node: '#demo' });
    resize.on('resize:start', function () {
        this.get('node').addClass('resizing');
    });
    resize.on('resize:end', function () {
        this.get('node').removeClass('resizing');
    });
});
</script>
```

Conveniently, the `Resize` instance stores a handle to the node it is acting on, which you can retrieve by calling `get('node')`. This handle is actually a YUI attribute, not to be confused with an HTML attribute. For more information about the `Attribute` API, refer to Recipe 7.1.

See Also

The `Resize` User Guide (*http://yuilibrary.com/yui/docs/resize/*); `Resize` API documentation (*http://yuilibrary.com/yui/docs/api/modules/resize.html*).

3.9 Implementing a Reorderable Drag-and-Drop Table

Problem

You want to enable the user to reorganize a table's rows using Drag and Drop.

Solution

Use `Y.all()` and `each()` to configure each row in the table body as a draggable node and as a drop target, as shown in Example 3-18. Constrain each row to the interior of the table, and set each row not only to be draggable by proxy, but to stay in place when the user drops the proxy on the target.

This means that there are three main elements of concern:

The dragged element
: The row the user is trying to drag, which stays in place

The proxy element
: A "ghost" row that follows the user's mouse or finger

The drop target
: The row that the proxy is hovering over, or that has been dropped on

After configuring drag and drop targets, use the Drag and Drop Manager, `Y.DD.DDM`, to handle events that bubble up from dragged elements and drop targets. The most important event is the `drop:hit` event, which fires when the user drops the element over a drop target. Here the handler function checks whether the proxy's midpoint was above or below the drop target's midpoint. Based on this check, it inserts the dragged element either before or after the drop target. The proxy automatically disappears, and the DOM change causes the browser to slide the dragged row into its new position. Other events such as `drag:start` and `drag:end` need listeners only for cosmetic reasons.

Example 3-18. Reorderable drag-and-drop table

```
<!DOCTYPE html>
<title>Reorderable drag-and-drop table</title>
<style>
table.dd {
    border: 1px #000 solid; border-spacing: 1px;
    background: #844; width: 25em;
}
table.dd th { background: #999; padding: 0.2em; }
table.dd td { background: #ddd; padding: 0.2em; }
table.dd td.over { background: #9c9; }
table.dd tr.being-dragged { opacity: 0.5; }
</style>

<table class="dd">
<thead>
    <tr><th>Type</th><th>From</th><th>Weaknesses</th></tr>
</thead>
<tbody>
    <tr><td>Vampires</td><td>Transylvania</td><td>Crosses, Garlic</td></tr>
    <tr><td>Werewolves</td><td>The Forest</td><td>Silver, Teen Angst</td></tr>
    <tr><td>Zombies</td><td>Unwise Experiments</td><td>Headshots</td></tr>
    <tr><td>Robots</td><td>The Distant Future</td><td>Illogic</td></tr>
    <tr><td>Ninjas</td><td>Feudal Japan</td><td>Dishonor</td></tr>
```

```
    <tr><td>Pirates</td><td>The High Seas</td><td>Rum</td></tr>
    <tr><td>Bob</td><td>Human Resources</td><td>None Known</td></tr>
</tbody>
</table>

<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use('dd-drag', 'dd-drop', 'dd-proxy', 'dd-constrain', function (Y) {
    var rows = Y.all('table.dd tbody tr');
    rows.each(function (row) {
        var rowDrop = new Y.DD.Drop({ node: row }),
            rowDrag = new Y.DD.Drag({ node: row });

        rowDrag.plug(Y.Plugin.DDConstrained, { constrain2node: 'table.dd' });
        rowDrag.plug(Y.Plugin.DDProxy, { moveOnEnd: false });
    });

    function midpoint(node) {
        return node.getY() + (node.get('offsetHeight') / 2);
    }

    Y.DD.DDM.on('drop:hit', function (ev) {
        var drop = ev.drop.get('node'),
            drag = ev.drag.get('node'),
            proxy = ev.drag.get('dragNode');

        if (midpoint(proxy) >= midpoint(drop)) {
            drop.insert(drag, 'after');
        }
        else {
            drop.insert(drag, 'before');
        }
        drop.all('td').removeClass('over');
    });

    Y.DD.DDM.on('drag:start', function (ev) {
        ev.target.get('node').addClass('being-dragged');
    });

    Y.DD.DDM.on('drag:end', function (ev) {
        ev.target.get('node').removeClass('being-dragged');
    });

    Y.DD.DDM.on('drop:over', function (ev) {
        ev.drop.get('node').all('td').addClass('over');
    });

    Y.DD.DDM.on('drop:exit', function (ev) {
        ev.target.get('node').all('td').removeClass('over');
    });
});
</script>
```

Discussion

If you've read Recipe 2.5, you should be familiar with using `Y.all()` and `NodeList` to work with a collection of nodes. The `each()` method applies a function to each node in the `NodeList`. Conveniently, the `<table>` markup supplies an explicit `<thead>` and `<tbody>`, making it easy to exclude the header rows in the `Y.all()`.

Chapter 4 discusses events in much more detail, but the key concept in Example 3-18 is `Y.DD.DDM`, which listens for all Drag and Drop custom events, signified with the prefix `drag:`. The Drag and Drop Manager provides a central point of control for handling Drag and Drop events. For more information about how to configure custom events to bubble up to a particular event target, refer to Recipe 4.7.

Each event handler function receives an event object representing the drag event. The event object provides a `target` object representing the node that is being acted upon, and the Drag and Drop API may further decorate the event object with a `drag` object, a `drop` object, and even a `dragNode` object (which can represent the proxy). This enables you to modify the relevant nodes as Drag and Drop events occur.

As mentioned in the solution, `drop:hit` is the core event handler that is actually responsible for inserting the row into a new location in the DOM. Keep in mind that if you want to implement a reorderable table, list, or anything else with YUI, you must use a Drag and Drop proxy and set `moveOnEnd` to `false`. When Drag and Drop moves a dragged node, it changes the node's `position` to be absolute and animates its *xy* coordinates appropriately. In a reorderable list or table, this is undesirable for two reasons.

First, as soon as the drag begins, the table will try to close on the missing row. You can solve this by using a proxy, preventing the table from closing on the row.

Second, when the user drops the row, the row continues to float at its current *xy* coordinates and will *look* incorrect, even if your code inserts the row into the correct DOM location. You can solve this by setting `moveOnEnd` to `false`, which prevents Drag and Drop from artificially changing the row's `position` and *xy* coordinates, and by listening for `drop:hit` as the signal to change the structure of the table. When the row drops, the browser simply reflows and displays all table rows in their natural, correct position.

The other event handlers are there to improve aesthetics and usability. For example, the `drag:start` handler clarifies which row is being dragged, while the `drop:over` handler highlights the current target to help the user see where the row will be dropped.

Some variations you could make to this recipe include:

- Instead of inserting the row into the DOM on a `drop:hit`, insert it into the DOM on every `drop:over` event. In this implementation, the dragged row appears to slide its way through its neighbors as the user drags the row around.
- The current implementation is a bit touchy when the user is trying to drag and insert an element at the top or bottom. You can make this action a little easier by expanding the possible drop targets beyond just the rows containing table data.