SPRY 1.4
DEVELOPER GUIDE

© 2007 Adobe Systems Incorporated. All rights reserved.

Spry framework for Ajax User Guide for Windows® and Mac OS

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Dreamweaver and Flash are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Apple and Mac OS are trademarks of Apple Inc., registered in the United States and other countries. All other trademarks are the property of their respective owners.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Notice to U.S. Government End Users. The Software and Documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contents

Chapter 1: Spry 1.4 Overview
About the Spry 1.4 framework 1

Chapter 2: Working with Spry widgets
About Spry widgets 2
Before you begin 3
Working with the Accordion widget 4
Working with the Collapsible Panel widget 14
Working with the Tabbed Panels widget 23
Working with the Menu Bar widget 33
Working with the Validation Text Field widget 44
Working with the Validation Text Area widget 65
Working with the Validation Select widget 75
Working with the Validation Checkbox widget 83

Chapter 3: Working with Spry XML Data Sets
About Spry XML Data Sets and dynamic regions 91
Building dynamic pages with Spry 110
Getting and manipulating data 121
Working with dynamic regions 128

Chapter 4: Working with Spry Effects
About Spry effects 141
Before you begin 142
Attach Effects 142

Index 152

Chapter 1: Spry 1.4 Overview

The Spry 1.4 framework for Ajax is a JavaScript library that provides web designers with the ability to build richer, more interesting, and more interactive web pages.

About the Spry 1.4 framework

About the Spry 1.4 framework

The Spry 1.4 framework for Ajax is a JavaScript library that provides web designers with the ability to build web pages that offer richer experiences to their site visitors. With Spry, you can use HTML code, CSS code, and a minimal amount of JavaScript to incorporate XML data into your HTML documents, create widgets such as accordions and menu bars, and add different kinds of effects to various page elements. The Spry framework is designed so that the code is simple and easy to use for those who have basic knowledge of HTML, CSS, and JavaScript.

The Spry framework is meant primarily for users who are web design professionals or advanced nonprofessional web designers. It is not intended as a full web application framework for enterprise-level web development (though it can be used in conjunction with other enterprise-level pages).

The Spry 1.4 framework provides three large components that you can create dynamic pages with: widgets, XML data sets, and effects. Widgets are page elements, such as accordions and tabbed panels, that make your page more interesting and interactive; XML data sets let you display data from an XML data source on your web page, much as a traditional web application lets you display data from a database; and Spry effects, such as Fade and Squish, let you improve your user's experience by adding motion to the page. You can display XML data inside a widget and add effects to widgets to create much richer pages than static HTML allows. The sections that follow provide more information on how to use widgets, data sets, and effects individually.

For examples of how to use the Spry framework, including examples that combine the use of widgets, data sets, and effects, visit the Spry framework home page on Adobe Labs at http://labs.adobe.com/technologies/spry/. This page also includes the latest updates for Spry.

About Spry 1.4 and Dreamweaver

Spry 1.4 is the version of Spry that's included with Dreamweaver CS3, so this documentation is designed to work with the Spry assets that Dreamweaver includes. This version of the documentation, however, does not discuss the visual tools that help you build Spry pages in Dreamweaver CS3. For information on how to use Spry tools in Dreamweaver, see Dreamweaver Help.

For the most recent version of the Spry Framework help (i.e., Spry 1.5 and later), visit www.adobe.com/go/learn_spry. Future versions of Spry Help might not always be compatible with the current Spry assets that Dreamweaver includes. If you are a Dreamweaver CS3 user and want to use later versions of the Spry framework, make sure you download the latest Spry assets from Adobe Labs as well.

Chapter 2: Working with Spry widgets

A Spry widget is a page element that combines HTML, CSS and JavaScript data to enable user interaction. The Spry framework for Ajax supports a set of reusable widgets written in standard HTML, CSS, and JavaScript code.

About Spry widgets

About Spry widgets

A *Spry widget* is a page element that combines HTML, CSS and JavaScript code to enable user interaction. A Spry widget is made up of the following parts:

Widget structure An HTML code block that defines the structural composition of the widget.

Widget behavior JavaScript code that controls how the widget responds to user-initiated events.

Widget styling CSS code that specifies the appearance of the widget.

The Spry framework supports a set of reusable widgets written in standard HTML, CSS, and JavaScript code. You can easily insert these widgets—the code is HTML and CSS at its simplest—and then style the widget. The behaviors in the framework include functionality that lets users show or hide content on the page, change the appearance (such as color) of the page, interact with menu items, and much more.

Each widget in the Spry framework is associated with unique CSS and JavaScript files, available on Adobe Labs. The CSS file contains everything necessary for styling the widget, and the JavaScript file gives the widget its functionality. The CSS and JavaScript files associated with a given widget are named after the widget, so it's easy for you to know which files correspond to which widgets. (For example, the files associated with the Accordion widget are called SpryAccordion.css and SpryAccordion.js).

About Spry widget accessibility and JavaScript degradation

It is critical to the usability of the widget that it be accessible when following established web navigation conventions. You can't assume that the user is using a mouse, and therefore Adobe has taken steps to ensure that all aspects of the currently available widgets are accessible through the keyboard. In the Accordion widget, for example, you can use up and down arrow keys to open content panels. Adobe encourages all widget developers to build in this kind of functionality.

It's also impossible to control the end user's environment. Adobe develops widgets to ensure that when JavaScript is turned off, all the content of the widget is still available on the screen. While this will most likely affect the page layout, it is more important that the content of the widget still be available, especially when working with disclosure widgets. Adobe ensures that default CSS states do not set visibility to `hidden`, and that HTML code is not positioned off screen.

Coding guidelines for developing Spry widgets

One of the goals of Spry is to enable the user community to build and share widgets. Adobe has a set of guidelines to use when authoring widgets for public distribution. Adobe is providing these guidelines with the hope that all widgets will have a consistent base functionality.

- Use standard HTML code for structure
- Don't require CSS code unless necessary
- If you require CSS code for functionality, clearly document the requirements
- Use a single line (if possible) of JavaScript to enable the widget functionality
- Write keyboard navigation options in a function by default
- When JavaScript is turned off, the content should still appear on the page
- Widgets should be self-contained. Everything needed for the widget is provided in the HTML, JavaScript, and CSS files.

Keeping to these guidelines will help ensure that widgets are easy to understand and use, plus consistency strengthens the framework for everyone.

Using standard code is important because there is less to learn for the common user. It also makes it easy to use these widgets in WYSIWYG editors.

CSS code is used in some widgets to show and hide content by switching the visibility rule in CSS code. This is an example of a required use of CSS. Such a use is acceptable because the CSS code is the obvious mechanism for showing and hiding content. CSS code that is pure styling, however, should not be required. The widget should always function without styling. Document required CSS rules with comments in the CSS file, and if you're providing further documentation, mention it there as well.

Most widgets are activated with a single line of JavaScript code just after the actual widget code. Try to keep the JavaScript arguments to a minimum. Widths and heights of widgets should be set in CSS code, not in JavaScript, unless there are no other options.

Keyboard navigation and accessibility are important to users and to Spry. Write keyboard navigation so that users can use common workflow keys (arrow keys, space bar) to access all parts of your widget. Use things like tab order where appropriate.

It's vital that content not be hidden in non-scripting environments. Ensure that when JavaScript is turned off, your content is not hidden because of CSS visibility being turned off or content being positioned off screen.

Before you begin

Prepare your files

Before you add Spry widgets to your web pages, download and link the appropriate files.

1 Locate the Spry ZIP file on the Adobe® Labs website.

2 Download and unzip the Spry ZIP file to your hard drive.

3 Open the unzipped Spry folder and locate the widgets folder. This folder contains all of the files necessary for adding and styling Spry widgets.

4 Add widget files to your website by doing one of the following:

- Copy the widgets folder and paste or drag a copy of it to the root directory of your web site. If you do this, you will have all of the files necessary for all of the widgets that Spry supports.
- Create a folder in your website (for example, a folder called *SpryAssets*), open the widgets folder, and copy only the files or folders that pertain to the widgets you want to add. For example, to add only an Accordion widget to your web pages, copy the accordion folder, or the SpryAccordion.js and SpryAccordion.css files.

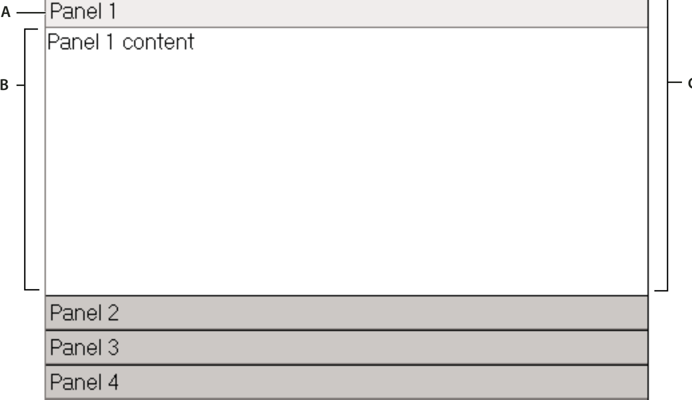
***Note:** If you drag the original widgets folder or individual files out of the unzipped Spry folder, the demos in the Spry folder won't work properly. Be sure to copy and paste to your website instead of dragging.*

5 When the correct widget JavaScript and CSS files are part of your website, you are ready to link them and add Spry widgets to your pages. For specific instructions on adding each widget to a page, see the individual widget sections.

Working with the Accordion widget

Accordion widget overview and structure

An Accordion widget is a set of collapsible panels that can store a large amount of content in a compact space. Site visitors hide or reveal the content stored in the accordion by clicking the tab of the panel. The panels of the accordion expand or contract accordingly as the visitor clicks different tabs. In an accordion, only one content panel is open and visible at a given time. The following example shows an Accordion widget, with the second panel expanded:



***A.** Accordion panel tab **B.** Accordion panel content **C.** Accordion panel (open)*

The default HTML code for the Accordion widget comprises an outer `div` tag that contains all of the panels, a `div` tag for each panel, and a header `div` and content `div` within the tag for each panel. An Accordion widget can contain any number of individual panels. The HTML code for the Accordion widget also includes `script` tags in the head of the document and after the accordion's HTML code.

The `script` tag in the head of the document defines all of the JavaScript functions related to the Accordion widget. The `script` tag after the Accordion widget code creates a JavaScript object that makes the accordion interactive. Following is the HTML code for an Accordion widget:

```
<head>
...
    <!--Link the CSS style  sheet that styles the accordion-->
    <link href="SpryAssets/SpryAccordion.css" rel="stylesheet" type="text/css" />
    <!--Link the Spry Accordion JavaScript library-->
    <script src="SpryAssets/SpryAccordion.js" type="text/javascript"></script>
</head>
<body>
    <!--Create the Accordion widget and assign classes to each element-->
    <div id="Accordion1" class="Accordion">
        <div class="AccordionPanel">
            <div class="AccordionPanelTab">Panel 1</div>
            <div class="AccordionPanelContent">
                Panel 1 Content<br/>
                Panel 1 Content<br/>
                Panel 1 Content<br/>
            </div>
        </div>
        <div class="AccordionPanel">
            <div class="AccordionPanelTab">Panel 2</div>
            <div class="AccordionPanelContent">
                Panel 2 Content<br/>
                Panel 2 Content<br/>
                Panel 2 Content<br/>
            </div>
        </div>
        <div class="AccordionPanel">
            <div class="AccordionPanelTab">Panel 3</div>
            <div class="AccordionPanelContent">
                Panel 3 Content<br/>
                Panel 3 Content<br/>
                Panel 3 Content<br/>
            </div>
        </div>
        <div class="AccordionPanel">
            <div class="AccordionPanelTab">Panel 4</div>
            <div class="AccordionPanelContent">
                Panel 4 Content<br/>
                Panel 4 Content<br/>
                Panel 4 Content<br/>
            </div>
        </div>
    </div>
<script type="text/javascript">
    var Accordion1 = new Spry.Widget.Accordion("Accordion1");
</script>
</body>
```

In the code, the `new` JavaScript operator initializes the Accordion widget object, and transforms the `div` content with the ID of `Accordion1` from static HTML code into an interactive page element. The `Spry.Widget.Accordion` method is a constructor in the Spry framework that creates accordion objects, and the information necessary to initialize the object is contained in the SpryAccordion.js JavaScript library that you linked to in the head of the document.

Each of the `div` tags in the Accordion widget contains a CSS class. These classes control the style of the Accordion widget, and exist in the accompanying SpryAccordion.css file.

You can change the appearance of any given part of the Accordion widget by editing the CSS code that corresponds to the class names assigned to it in the HTML code. For example, to change the background color of the accordion's tabs, edit the `AccordionPanelTab` rule in the SpryAccordion.css file. Keep in mind that changing the CSS code in the SpryAccordion.css file will affect all accordions that are linked to that file.

In addition to the classes shown in the HTML code, the Accordion widget includes certain default behaviors that are attached to the widget. These behaviors are a built-in part of the Spry framework, and are in the SpryAccordion.js JavaScript library file. The Accordion library includes behaviors related to hovering, tab clicking (to open panels), panel focus, and keyboard navigation.

You can change the look of the accordion as it relates to these behaviors by editing the appropriate classes in the SpryAccordion.css file. If for some reason you want to remove a given behavior, you can delete the CSS rules that correspond to that behavior.

***Note:** While you can change the look of the accordion as it relates to a certain behavior, you cannot alter the built-in behaviors themselves. For example, Spry still places an AccordionFocused class on an accordion in focus, even if no properties are set for the AccordionFocused class in the SpryAccordion.css file.*

Variation on tags used for Accordion widget structure

In the preceding example, `div` tags create a nested tag structure for the widget:

```
Container div
    Panel div
        Tab div
        Content div
```

This 3-level, 4-container structure is essential for the Accordion widget to work properly; the structure, however, is more important than the actual tags you decide to use. Spry reads the structure (not `div` tags necessarily) and builds the widget accordingly. As long as the 3-level, 4-container structure is in place, you can use any block-level element to create the widget:

```
Container div
    Panel div
        H3 tag
        P tag
```

The `div` tags in the example are flexible and can contain other block-level elements. A `p` tag (or any other inline tag), however, cannot contain other block-level elements, so you cannot use it as a container or panel tag for the accordion.

CSS code for the Accordion widget

The SpryAccordion.css file contains the rules that style the Accordion widget. You can edit these rules to style the accordion's look and feel. The names of the rules in the CSS file correspond directly to the names of the classes specified in the Accordion widget's HTML code.

***Note:** You can replace style-related class names in the SpryAccordion.css file and HTML code with class names of your own. Doing so does not affect the functionality of the widget, as CSS code is not required for widget functionality.*

The default functionality for the behaviors classes at the end of the style sheet are defined in the JavaScript library file SpryAccordion.js. You can change the default functionality by adding properties and values to the behavior rules in the style sheet.

The following is the CSS code for the SpryAccordion.css file:

```
/*Accordion styling classes*/
.Accordion {
    border-left: solid 1px gray;
    border-right: solid 1px black;
    border-bottom: solid 1px gray;
    overflow: hidden;
}
.AccordionPanel {
    margin: 0px;
    padding: 0px;
}
.AccordionPanelTab {
    background-color: #CCCCCC;
    border-top: solid 1px black;
    border-bottom: solid 1px gray;
    margin: 0px;
    padding: 2px;
    cursor: pointer;
}
.AccordionPanelContent {
    overflow: auto;
    margin: 0px;
    padding: 0px;
    height: 200px;
}
.AccordionPanelOpen .AccordionPanelTab {
    background-color: #EEEEEE;
}
.AccordionPanelClosed .AccordionPanelTab {
}
/*Accordion behaviors  classes*/
.AccordionPanelTabHover {
    color: #555555;
}
.AccordionPanelOpen .AccordionPanelTabHover {
    color: #555555;
}
.AccordionFocused .AccordionPanelTab {
    background-color: #3399FF;
}
.AccordionFocused .AccordionPanelOpen .AccordionPanelTab {
    background-color: #33CCFF;
}
```

The SpryAccordion.css file contains extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Insert the Accordion widget

1 Locate the SpryAccordion.js file and add it to your web site. You can find the SpryAccordion.js file in the widgets/accordion directory, located in the Spry directory that you downloaded from Adobe Labs. For more information, see “Prepare your files” on page 3.

For example, create a folder called *SpryAssets* in the root folder of your web site, and move the SpryAccordion.js file to it. The SpryAccordion.js file contains all of the information necessary for making the Accordion widget interactive.

2 Locate the SpryAccordion.css file and add it to your web site. You might choose to add it to the main directory, a SpryAssets directory, or to a CSS directory if you have one.

3 Open the web page to add the Accordion widget to and link the SpryAccordion.js file to the page by inserting the following `script` tag in the page's head tag:

```
<script src="SpryAssets/SpryAccordion.js" type="text/javascript"></script>
```

Make sure that the file path to the SpryAccordion.js file is correct. This path varies depending on where you've placed the file in your web site.

4 Link the SpryAccordion.css file to your web page by inserting the following `link` tag in the page's head tag:

```
<link href="SpryAssets/SpryAccordion.css" rel="stylesheet" type="text/css" />
```

Make sure that the file path to the SpryAccordion.css file is correct. This path varies depending on where you've placed the file in your web site.

5 Add the accordion to your web page by inserting the following `div` tag where you want the accordion to appear on the page:

```
<div id="Accordion1" class="Accordion">
</div>
```

6 Add panels to the accordion by inserting `<div class="AccordionPanel">` tags inside the `<div id="Accordion1"...>` tag, as follows:

```
<div id="Accordion1" class="Accordion">
    <div class="AccordionPanel">
    </div>
    <div class="AccordionPanel">
    </div>
</div>
```

The preceding code adds two panels to the accordion. You can add unlimited panels.

7 To add tabs to the panels, insert `div class="AccordionPanelTab"` tags inside each `div class="AccordionPanel"` tag, as follows:

```
<div class="AccordionPanel">
    <div class="AccordionPanelTab">Panel 1 Title</div>
</div>
```

8 To add a content area to the panels, insert `div class="AccordionPanelContent"` tags in each `div class="AccordionPanel"` tag, as follows:

```
<div class="AccordionPanel">
    <div class="AccordionPanelTab">Panel 1 Title</div>
    <div class="AccordionPanelContent">Panel 1 Content</div>
</div>
```

Insert the content between the opening and closing `AccordionPanelContent` tags. For example, `Panel 1 Content`, as in the preceding example. The content can be any valid HTML code, including HTML form elements.

9 To initialize the Spry accordion object, insert the following script block after the HTML code for the Accordion widget:

```
<div id="Accordion1" class="Accordion">
. . .
</div>
<script type="text/javascript">
    var Accordion1 = new Spry.Widget.Accordion("Accordion1");
</script>
```

The `new` JavaScript operator initializes the Accordion widget object, and transforms the `div` content with the ID of `Accordion1` from static HTML code into an interactive accordion object. The `Spry.Widget.Accordion` method is a constructor in the Spry framework that creates accordion objects. The information necessary to initialize the object is contained in the SpryAccordion.js JavaScript library that you linked to at the beginning of this procedure.

Make sure that the ID of the accordion's `div` tag matches the ID parameter you specified in the `Spry.Widgets.Accordion` method. Make sure that the JavaScript call comes after the HTML code for the widget.

10 Save the page.

The complete code looks as follows:

```
<head>
...
<link href="SpryAssets/SpryAccordion.css" rel="stylesheet" type="text/css" />
<script src="SpryAssets/SpryAccordion.js" type="text/javascript"></script>
</head>
<body>
    <div id="Accordion1" class="Accordion">
        <div class="AccordionPanel">
            <div class="AccordionPanelTab">Panel 1</div>
            <div class="AccordionPanelContent">
                Panel 1 Content<br/>
                Panel 1 Content<br/>
                Panel 1 Content<br/>
            </div>
        </div>
        <div class="AccordionPanel">
            <div class="AccordionPanelTab">Panel 2</div>
            <div class="AccordionPanelContent">
                Panel 2 Content<br/>
                Panel 2 Content<br/>
                Panel 2 Content<br/>
            </div>
        </div>
        <div class="AccordionPanel">
```

```
            <div class="AccordionPanelTab">Panel 3</div>
            <div class="AccordionPanelContent">
                Panel 3 Content<br/>
                Panel 3 Content<br/>
                Panel 3 Content<br/>
            </div>
        </div>
        <div class="AccordionPanel">
            <div class="AccordionPanelTab">Panel 4</div>
            <div class="AccordionPanelContent">
                Panel 4 Content<br/>
                Panel 4 Content<br/>
                Panel 4 Content<br/>
            </div>
        </div>
    </div>
<script type="text/javascript">
    var Accordion1 = new Spry.Widget.Accordion("Accordion1");
</script>
</body>
```

Add a panel to an Accordion widget

❖ Insert a `div class="AccordionPanel"` tag (along with tags for a panel tab and a panel content area) inside the container `div` tag for the accordion. Do not forget to add the closing `</div>` tag when you add the code. For example:

```
<div id="Accordion1" class="Accordion">
    <div class="AccordionPanel">
        <div class="AccordionPanelTab"></div>
        <div class="AccordionPanelContent"></div>
    </div>
</div>
```

The preceding code adds a panel to the Accordion widget. You can add unlimited panels.

Delete a panel from an Accordion widget

❖ Delete the desired `div class="AccordionPanel"` tag (and its content or child tags) from the container `div` tag for the accordion. Do not forget to delete the closing `/div` tag when you delete the code.

Enable keyboard navigation

Making widgets accessible for keyboard navigation is an important part of every widget. Keyboard navigation lets the user control the widget with arrow keys and custom keys.

The foundation of keyboard navigation is the `tabIndex` attribute. This attribute tells the browser how to navigate through the document.

❖ To enable keyboard navigation on the accordion, add a `TabIndex` value to the accordion container tag, as follows:

```
<div id="Acc1" class="Accordion" tabIndex="0">
```

If the `tabIndex` attribute has a value of zero (0), the browser determines the order. If the `tabIndex` attribute has a positive integer value, that order value is used.

***Note:** Using tabIndex on a div tag is not XHTML 1.0 compliant.*

You can also set custom keys for keyboard navigation. Custom keys are set as arguments of the accordion constructor script:

```
<script type="text/javascript">
var acc3= new Spry.Widget.Accordion("Acc3", { nextPanelKeyCode: 78 /* n key */,
previousPanelKeyCode: 80 /* p key */ });
</script>
```

Set the default open panel

You can set a certain panel to be open when the page containing the Accordion widgets loads in a browser.

❖ Set the `defaultPanel` option in the constructor as follows:

```
<script type="text/javascript">
    var acc8 = new Spry.Widget.Accordion("Accordion1", { defaultPanel: 2 });
</script>
```

***Note:** The accordion panels use a zero-based counting system, so setting the value to 2 opens the third panel.*

Open panels programatically

You can programatically open different panels by using JavaScript functions. For example, you might have a button on your page that opens a particular accordion panel when the user clicks the button.

❖ Use the following JavaScript functions to open accordion panels:

```
<input type="button" onclick="acc10.openFirstPanel()" >open first panel</input>
<input type="button" onclick="acc10.openNextPanel()" >open next panel</input>
<input type="button" onclick="acc10.openPreviousPanel()" >open previous panel</input>
<input type="button" onclick="acc10.openLastPanel()" >open last panel</input>
<script type="text/javascript">
    var acc10 = new Spry.Widget.Accordion("Accordion1");
</script>
```

Customize the Accordion widget

The SpryAccordion.css file provides the default styling for the Accordion widget. You can, however, easily customize the widget by changing the appropriate CSS. The CSS rules in the SpryAccordion.css file use the same class names as the related elements in the accordion's HTML code, so it's easy for you to know which CSS rules correspond to the different sections of the Accordion widget. Additionally, the SpryAccordion.css file contains class names for behaviors that are related to the widget (for example, hovering and clicking behaviors).

The SpryAccordion.css file should already be included in your website before you start customizing. For more information, see "Prepare your files" on page 3.

***Note:** Internet Explorer up to and including version 6 does not support sibling and child contextual selectors (for example, `.AccordionPanel + .AccordionPanel` or `.Accordion > .AccordionPanel`).*

Style an Accordion widget (general instructions)

Set properties for the entire Accordion widget container, or set properties for the components of the widget individually.

1 Open the SpryAccordion.css file.

2 Locate the CSS rule for the part of the accordion to change. For example, to change the background color of the accordion's tabs, edit the `AccordionPanelTab` rule in the SpryAccordion.css file.

3 Make your changes to the CSS and save the file.

You can replace style-related class names in the SpryAccordion.css file and HTML code with class names of your own. Doing so does not affect the functionality of the widget.

The SpryAccordion.css file contains extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Style Accordion widget text

Set properties for the entire Accordion widget container, or set properties for the components of the widget individually.

❖ To change the text styling of an Accordion widget, use the following table to locate the appropriate CSS rule in the SpryAccordion.css file, and then add your own text styling properties and values.

Text to change	Relevant CSS rule	Example of properties and values to add
Text in the entire accordion (includes both tab and content panel)	.Accordion or .AccordionPanel	font: Arial; font-size:medium;
Text in accordion panel tabs only	.AccordionPanelTab	font: Arial; font-size:medium;
Text in accordion content panels only	.AccordionPanelContent	font: Arial; font-size:medium;

Change Accordion widget background colors

❖ Use the following table to locate the appropriate CSS rule in the SpryAccordion.css file, and then add or change background color properties and values.

Part of widget to change	Relevant CSS rule	Example of property and value to add or change
Background color of accordion panel tabs	.AccordionPanelTab	background-color: #CCCCCC; (This is the default value.)
Background color of accordion content panels	.AccordionPanelContent	background-color: #CCCCCC;
Background color of the open accordion panel	.AccordionPanelOpen .AccordionPanelTab	background-color: #EEEEEE; (This is the default value.)
Background color of panel tabs on hover	.AccordionPanelTabHover	color: #555555; (This is the default value.)
Background color of open panel tab on hover	.AccordionPanelOpen .AccordionPanelTabHover	color: #555555; (This is the default value.)

Constrain the width of an accordion

By default, the Accordion widget expands to fill available space. To constrain the width of an Accordion widget, set a width property for the accordion container.

1 Locate the `.Accordion` CSS rule in the SpryAccordion.css file. This rule defines properties for the main container element of the Accordion widget.

2 Add a width property and value to the rule, for example `width: 300px;`.

Change accordion panel height

By default, the height of Accordion widget panels is set to 200 pixels. To change the height of panels, change the height property in the `.AccordionPanelContent` rule.

1 Locate the `.AccordionPanelContent` CSS rule in the SpryAccordion.css file.

2 Change the height property to a dimension of your choosing.

***Note:** Always set this value to ensure that accordion panel sizes are the same.*

Change accordion panel behaviors

The Accordion widget includes some predefined behaviors. These behaviors consist of changing CSS classes when a particular event occurs. For example, when a mouse pointer hovers over an accordion panel tab, Spry applies the AccordionPanelTabHover class to the widget. (This behavior is similar to `a:hover` for links.) The behaviors are blank by default; to change them, add properties and values to the rules.

1 Open the SpryAccordion.css file and locate the CSS rule for the accordion behavior to change. The Accordion widget includes four built-in rules for behaviors.

Behavior	Description
.AccordionPanelTabHover	Activates when hovering over the panel tab
.AccordionPanelOpen	Activates on the tab of the open panel
.AccordionPanelClosed	Activates on the closed panels
.AccordionFocused	Activates when the entire accordion gets focus.

For examples, see the Accordion sample file located in the samples directory of the Spry directory that you downloaded from Adobe Labs. For more information, see “Prepare your files” on page 3.

2 Add CSS rules for any of the behaviors you want to enable.

***Note:** You cannot replace behavior-related class names in the SpryAccordion.css file with class names of your own because the behaviors are hard coded as part of the Spry framework. To override the default class with your class names, send the new values as arguments to the accordion constructor, as the following example shows:*

```
<script type="text/javascript">
    var acc2 = new Spry.Widget.Accordion("Acc2", { hoverClass: "hover", openClass: "open",
closedClass: "closed", focusedClass: "focused" });
</script>
```

Turn off panel animation

By default, accordion panels use animation to smoothly open and close. You can turn this animation off, however, so that the panels instantly open and close.

❖ To turn off animation, send an argument in the accordion constructor, as follows:

```
<script type="text/javascript">
    var acc5 = new Spry.Widget.Accordion("Acc5", { enableAnimation: false });
</script>
```

Set panel animation timing

You can change the time it takes for a panel to open. Set the time in milliseconds (1000 = 1 second). By default, Spry uses 500 milliseconds.

❖ Add the duration option to the constructor:

```
<script type="text/javascript">
    var acc9 = new Spry.Widget.Accordion("Acc9", { duration: 100 });
</script>
```

Set variable panel heights

By default, when animation is enabled, Spry forces all accordion content panels to use the same height. Spry derives this height from the height of the default open panel of the accordion, which is determined by CSS or by the height of the content in the panel.

The accordion also supports variable height panels. To turn this support on, pass a `useFixedPanelHeights: false` option to the accordion's constructor.

❖ Pass a `useFixedPanelHeights:false` option as follows:

```
<script type="text/javascript">
    var acc7 = new Spry.Widget.Accordion("Acc7", { useFixedPanelHeights: false });
</script>
```

For an example, see the Accordion sample file located in the samples directory of the Spry directory that you downloaded from Adobe Labs. For more information, see “Prepare your files” on page 3.

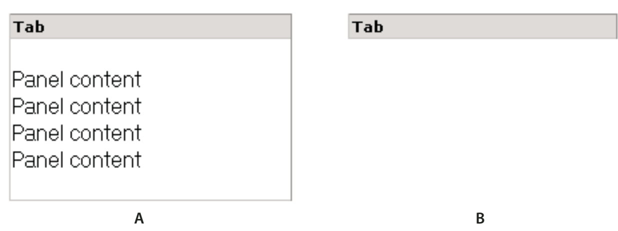
To have Spry set the panel height to a set value in JavaScript (rather than CSS code), pass the `fixedPanelHeight` option that programatically sets the content panel heights. Pixels are used for this option.

```
<script type="text/javascript">
    var acc7 = new Spry.Widget.Accordion("Acc7", { fixedPanelHeight: "300px" });
</script>
```

Working with the Collapsible Panel widget

Collapsible Panel widget overview and structure

A Collapsible Panel widget is a panel that can store content in a compact space. Users hide or reveal the content stored in the Collapsible Panel by clicking the tab of the widget. The following example shows a Collapsible Panel widget, expanded and collapsed.



***A.** Expanded* ***B.** Collapsed*

The HTML code for the Collapsible Panel widget is made up of an outer `div` tag that contains the content `div` tag and the tab container `div` tag. The HTML code for the Collapsible Panel widget also includes `script` tags in the head of the document and after the Collapsible Panel's HTML code.

The `script` tag in the head of the document defines all of the JavaScript functions related to the Collapsible Panel widget. The `script` tag after the Collapsible Panel widget code creates a JavaScript object that makes the Collapsible Panel interactive. Following is the HTML code for an Collapsible Panel widget:

```
<head>
...
    <!--Link the CSS style  sheet that styles the Collapsible Panel-->
    <link href="SpryAssets/SpryCollapsiblePanel.css" rel="stylesheet"
        type="text/css" />
    <!--Link the Spry Collapsible Panel JavaScript library-->
    <script src="SpryAssets/SpryCollapsiblePanel.js" type="text/javascript"></script>
</head>
<body>
    <!--Create the Collapsible Panel widget and assign classes to each element-->
    <div id="CollapsiblePanel1" class="CollapsiblePanel">
        <div class="CollapsiblePanelTab">Tab</div>
        <div class="CollapsiblePanelContent">Content</div>
    </div>
    <!--Initialize the Collapsible Panel widget object-->
    <script type="text/javascript">
        var CollapsiblePanel1 = new Spry.Widget.CollapsiblePanel("CollapsiblePanel1");
    </script>
</body>
```

In the code, the `new` JavaScript operator initializes the Collapsible Panel widget object, and transforms the `div` content with the ID of `CollapsiblePanel1` from static HTML code into an interactive page element. The `Spry.Widget.CollapsiblePanel` method is a constructor in the Spry framework that creates Collapsible Panel objects, and the information necessary to initialize the object is contained in the SpryCollapsiblePanel.js JavaScript library that you linked to in the head of the document.

Each of the elements in the Collapsible Panel widget contains a CSS class. These classes control the style of the Collapsible Panel widget, and exist in the accompanying SpryCollapsiblePanel.css file.

You can change the appearance of any given part of the Collapsible Panel widget by editing the CSS code that corresponds to the class names assigned to it in the HTML code. For example, to change the background color of the Collapsible Panel's tabs, edit the `CollapsiblePanelTab` rule in the SpryCollapsiblePanel.css file. Keep in mind that changing the CSS code in the SpryCollapsiblePanel.css file will affect all collapsible panels that are linked to that file.

In addition to the classes shown in the HTML code, the Collapsible Panel widget includes certain default behaviors that are attached to the widget. These behaviors are a built-in part of the Spry framework, and are in the SpryCollapsiblePanel.js JavaScript library file. The Collapsible Panel library includes behaviors related to hovering, clicking (to open and close the panel), panel focus, and keyboard navigation.

You can change the look of the Collapsible Panel as it relates to these behaviors by editing the appropriate classes in the SpryCollapsiblePanel.css file. If for some reason you want to remove a given behavior, you can delete the CSS rules that correspond to that behavior.

***Note:** While you can change the look of the Collapsible Panel as it relates to a certain behavior, you cannot alter the built-in behaviors themselves. For example, Spry still places a CollapsiblePanelFocused class on the currently open panel, even if no properties are set for the CollapsiblePanelFocused class in the SpryCollapsiblePanel.css file.*

Variation on tags used for Collapsible Panel widget structure

In the preceding example, `div` tags create a nested tag structure for the widget:

```
Container div
    Tab div
    Content div
```

This 2-level, 3-container structure is essential for the Collapsible Panel widget to work properly; the structure, however, is more important than the actual tags you decide to use. Spry reads the structure (not `div` tags necessarily) and builds the widget accordingly. As long as the 2-level, 3-container structure is in place, you can use any block-level element to create the widget:

```
Container div
    H3 tag
    P tag
```

The `div` tags in the example are flexible and can contain other block-level elements. A `p` tag (or any other inline tag), however, cannot contain other block-level elements, so you cannot use it as a container or panel tag for the collapsible panel.

CSS code for the Collapsible Panel widget

The SpryCollapsiblePanel.css file contains the rules that style the Collapsible Panel widget. You can edit these rules to style the collapsible panel's look and feel. The names of the rules in the CSS file correspond directly to the names of the classes specified in the Collapsible Panel widget's HTML code.

Note: *You can replace style-related class names in the SpryCollapsiblePanel.css file and HTML code with class names of your own. Doing so does not affect the functionality of the widget, as CSS code is not required for widget functionality.*

The default functionality for the behaviors classes at the end of the style sheet are defined in the SpryCollapsiblePanel.js JavaScript library file. You can change the default functionality by adding properties and values to the behavior rules in the style sheet.

The following is the CSS code for the SpryCollapsiblePanel.css file:

```
/*Collapsible Panel styling classes*/
.CollapsiblePanel {
    margin: 0px;
    padding: 0px;
    border-left: solid 1px #CCC;
    border-right: solid 1px #999;
}
.CollapsiblePanelTab {
    font: bold 0.7em sans-serif;
    background-color: #DDD;
    border-top: solid 1px #999;
    border-bottom: solid 1px #CCC;
    margin: 0px;
    padding: 2px;
    cursor: pointer;
    -moz-user-select: none;
    -khtml-user-select: none;
}
.CollapsiblePanelContent {
    margin: 0px;
    padding: 0px;
    border-bottom: solid 1px #CCC;
}
.CollapsiblePanelTab a {
    color: black;
    text-decoration: none;
}
.CollapsiblePanelOpen .CollapsiblePanelTab {
    background-color: #EEE;
}
.CollapsiblePanelTabHover,  .CollapsiblePanelOpen .CollapsiblePanelTabHover {
background-color: #CCC;
}
.CollapsiblePanelFocused .CollapsiblePanelTab {
    background-color: #3399FF;
}
```

The SpryCollapsiblePanel.css file contains extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Insert the Collapsible Panel widget

1 Locate the SpryCollapsiblePanel.js file and add it to your web site. You can find the SpryCollapsiblePanel.js file in the widgets/collapsiblepanel directory, located in the Spry directory that you downloaded from Adobe Labs. For more information, see “Prepare your files” on page 3.

For example, create a folder called *SpryAssets* in the root folder of your web site, and move the SpryCollapsiblePanel.js file to it. The SpryCollapsiblePanel.js file contains all of the information necessary for making the Collapsible Panel widget interactive.

2 Locate the SpryCollapsiblePanel.css file and add it to your web site. You might choose to add it to the main directory, or to a CSS directory if you have one.

3 Open the web page to add the Collapsible Panel widget to and link the SpryCollapsiblePanel.js file to the page by inserting the following `script` tag in the page’s head tag:

```
<script src="SpryAssets/SpryCollapsiblePanel.js" type="text/javascript"></script>
```

Make sure that the file path to the SpryCollapsiblePanel.js file is correct. This path varies depending on where you've placed the file in your web site.

4 Link the SpryCollapsiblePanel.css file to your web page by inserting the following `link` tag in the page's head tag:

```
<link href="SpryAssets/SpryCollapsiblePanel.css" rel="stylesheet" type="text/css" />
```

Make sure that the file path to the SpryCollapsiblePanel.css file is correct. This path varies depending on where you've placed the file in your web site.

5 To add the collapsible panel to your web page, insert the following `div` tag where you want the collapsible panel to appear on the page:

```
<div id="CollapsiblePanel1" class="CollapsiblePanel">
</div>
```

6 To add a tab to the collapsible panel, insert a `div class="CollapsiblePanelTab"` tag inside the `div class="CollapsiblePanel"` tag, as follows:

```
<div id="CollapsiblePanel1" class="CollapsiblePanel">
    <div class="CollapsiblePanelTab">Tab</div>
</div>
```

7 To add a content area to the panel, insert a `div class="CollapsiblePanelContent"` tag inside the `div class="CollapsiblePanel"` tag, as follows:

```
<div class="AccordionPanel">
    <div class="CollapsiblePanelTab">Tab</div>
    <div class="CollapsiblePanelContent">Content</div>
</div>
```

Insert the content between the opening and closing `CollapsiblePanelContent` tags (For example, `Content`, as in the preceding example). The content can be any valid HTML code, including HTML form elements.

8 To initialize the Spry collapsible panel object, insert the following script block after the HTML code for the Collapsible Panel widget:

```
<div id="CollapsiblePanel1" class="CollapsiblePanel">
. . .
</div>
<script type="text/javascript">
    var acc1 = new Spry.Widget.CollapsiblePanel("CollapsiblePanel1");
</script>
```

The `new` JavaScript operator initializes the Collapsible Panel widget object, and transforms the `div` content with the ID of CollapsiblePanel1 from static HTML code into an interactive collapsible panel object. The `Spry.Widget.CollapsiblePanel` method is a constructor in the Spry framework that creates collapsible panel objects. The information necessary to initialize the object is contained in the SpryCollapsiblePanel.js JavaScript library that you linked to at the beginning of this procedure.

Make sure that the ID of the collapsible panel's `div` tag matches the ID parameter you specified in the `Spry.Widgets.CollapsiblePanel` method. Make sure that the JavaScript call comes after the HTML code for the widget.

9 Save the page.

The complete code looks as follows:

```
<head>
...
<link href="SpryAssets/SpryCollapsiblePanel.css" rel="stylesheet" type="text/css" />
<script src="SpryAssets/SpryCollapsiblePanel.js" type="text/javascript"></script>
</head>
<body>
    <div id="CollapsiblePanel1" class="CollapsiblePanel">
        <div class="CollapsiblePanelTab">Tab</div>
        <div class="CollapsiblePanelContent">Content</div>
    </div>
<script type="text/javascript">
    var CollapsiblePanel1 = new Spry.Widget.CollapsiblePanel("CollapsiblePanel1");
</script>
</body>
```

Enable keyboard navigation

Making widgets accessible for keyboard navigation is an important part of every widget. Keyboard navigation lets the user control the widget with the Space bar or the Enter key.

The foundation of keyboard navigation is the `tabIndex` attribute. This attribute tells the browser how to use the tabs to navigate through the document.

❖ To enable keyboard navigation on the collapsible panel, add a `TabIndex` value to the collapsible panel tab tag, as follows:

```
<div id="CollapsiblePanel1" class="CollapsiblePanel">
    <div class="CollapsiblePanelTab" tabIndex="0">Tab</div>
    <div class="CollapsiblePanelContent">Content</div>
</div>
```

If the `tabIndex` attribute has a value of zero (0), the browser determines the order. If the `tabIndex` attribute has a positive integer value, that order value is used.

To enable keyboard navigation, wrap the tab content with an `a` tag.

Note: *Using tabIndex on a div tag is not XHTML 1.0 compliant.*

Set the default state of the panel

By default the Collapsible Panel widget is open when the web page loads in a browser. You can, however, change the status of the panel if you want the panel to be closed when the page loads.

❖ Set the contentIsOpen option in the constructor as follows:

```
<script type="text/javascript">
    var CollapsiblePanel1 = new Spry.Widget.CollapsiblePanel("CollapsiblePanel1", {
contentIsOpen: false });
</script>
```

You can also check the status of the panel with the isOpen function. The following code returns `true` if the panel is open and `false` if it is not:

```
<script type="text/javascript">
function  getStatus(){
var xx = CollapsiblePanel1.isOpen();
document.form1.textfield.value = xx
}
</script>
</head>
<body>
<div id="CollapsiblePanel1" class="CollapsiblePanel">
  <div class="CollapsiblePanelTab" tabIndex="0" onclick="getStatus();">Tab</div>
  <div class="CollapsiblePanelContent">Content</div>
</div>
<form id="form1" name="form1" method="post" action="">
  <input name="textfield" type="text" id="textfield" value="a" size="50" />
</form>
<script type="text/javascript">
<!--
var CollapsiblePanel1 = new Spry.Widget.CollapsiblePanel("CollapsiblePanel1");
//-->
</script>
```

Open the panel programatically

You can programatically open and close the Collapsible Panel widget by using JavaScript functions. For example, you might have a button on your page, that opens the collapsible panel when the user clicks the button.

❖ Use the following functions to open or close a collapsible panel:

```
<input type="button" onclick="CollapsiblePanel1.open();" >open panel</input>
<input type="button" onclick="CollapsiblePanel1.close();">close panel</input>
<script type="text/javascript">
    var CollapsiblePanel1 = new Spry.Widget.CollapsiblePanel("CollapsiblePanel1");
</script>
```

Customize the Collapsible Panel widget

The SpryCollapsiblePanel.css file provides the default styling for the Collapsible Panel widget. You can, however, easily customize the widget by changing the appropriate CSS rule. The CSS rules in the SpryCollapsiblePanel.css file use the same class names as the related elements in the collapsible panel's HTML code, so it's easy for you to know which CSS rules correspond to the different sections of the Collapsible Panel widget. Additionally, the SpryCollapsiblePanel.css file contains class names for behaviors that are related to the widget (for example, hovering and clicking behaviors).

The SpryCollapsiblePanel.css file should already be included in your website before you start customizing. For more information, see "Prepare your files" on page 3.

***Note:** Internet Explorer up to and including version 6 does not support Sibling and child contextual selectors (for example, `.CollapsiblePanel + .CollapsiblePanel` or `.CollapsiblePanel > .CollapsiblePanel`).*

Style a CollapsiblePanel widget (general instructions)

Set properties for the entire Collapsible Panel widget container, or set properties for the components of the widget individually.

1 Open the SpryCollapsiblePanel.css file.

2 Locate the CSS rule for the part of the collapsible panel to change. For example, to change the background color of the collapsible panel's tab, edit the `CollapsiblePanelTab` rule in the SpryCollapsiblePanel.css file.

3 Make your changes to the CSS rule and save the file.

You can replace style-related class names in the SpryCollapsiblePanel.css file and HTML code with class names of your own. Doing so does not affect the functionality of the widget.

The SpryCollapsiblePanel.css file contains extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Style Collapsible Panel widget text

Set properties for the entire Collapsible Panel widget container, or set properties for the components of the widget individually.

❖ To change the text format of a Collapsible Panel widget, use the following table to locate the appropriate CSS rule, and then add your own text styling properties and values.

Style to change	Relevant CSS rule	Example of properties and values to add or change
Text in the entire collapsible panel	.CollapsiblePanel	font: Arial; font-size:medium;
Text in panel tab only	.CollapsiblePanelTab	font: bold 0.7em sans-serif; (This is the default value.)
Text in content panel only	.CollapsiblePanelContent	font: Arial; font-size:medium;

Change Collapsible Panel widget background colors

❖ Use the following table to locate the appropriate CSS rule, and then add or change background color properties and values.

Color to change	Relevant CSS rule	Example of property and value to add or change
Background color of panel tab	.CollapsiblePanelTab	background-color: #DDD; (This is the default value.)
Background color of content panel	.CollapsiblePanelContent	background-color: #DDD;
Background color of tab when panel is open	.CollapsiblePanelOpen .CollapsiblePanelTab	background-color: #EEE; (This is the default value.)
Background color of open panel tab when the mouse pointer moves over it	.CollapsiblePanelTabHover, .CollapsiblePanelOpen .CollapsiblePanelTabHover	background-color: #CCC; (This is the default value.)

Constrain the width of a collapsible panel

By default, the Collapsible Panel widget expands to fill available space. To constrain the width of a Collapsible Panel widget, set a width property for the collapsible panel container.

1 Locate the `CollapsiblePanel` CSS rule in the SpryCollapsiblePanel.css file. This rule defines properties for the main container element of the Collapsible Panel widget.

2 Add a width property and value to the rule, for example `width: 300px;`.

Change collapsible panel height

To set the height of a collapsible panel, add a height property to the `CollapsiblePanelContent` rule or the `CollapsiblePanel` rule.

❖ In the SpryCollapsiblePanel.css file, add a height property and value to the `CollapsiblePanelContent` rule, for example, `height: 300px;`.

***Note:** Setting the height of the `CollapsiblePanel` rule sets the height of the entire widget, independent of the content panel size.*

Change collapsible panel behaviors

The collapsible panel widget includes a few predefined behaviors. These behaviors consist of changing CSS classes when a particular event occurs. For example, when a mouse pointer hovers over a collapsible panel tab, Spry applies the `CollapsiblePanelTabHover` class to the widget. (This behavior is similar to `a:hover` for links.) The behaviors are blank by default; to change them, add properties and values to the rules.

1 Open the SpryCollapsiblePanel.css file and locate the CSS rule for the collapsible panel behavior to change. The Collapsible Panel widget includes four built-in rules for behaviors.

Behavior	Description
.CollapsiblePanelTabHover	Spry adds this class to the widget's tab element whenever the mouse enters hovers over it. The class is automatically removed when the mouse leaves the tab.
.CollapsiblePanelOpen	Spry adds this class to the widget's top-level element when the panel’s content area is visible.
.CollapsiblePanelClosed	Spry adds this class to the widget's top-level element when the panel's content area is closed
.CollapsiblePanelFocused	Spry adds this class to the widget's top-level element when the widget has keyboard focus.

For examples, see the collapsible panel sample file located in the samples directory of the Spry directory that you downloaded from Adobe Labs. For more information, see “Prepare your files” on page 3.

2 Add CSS rules for any of the behaviors you want to enable.

***Note:** You cannot replace behavior-related class names in the SpryCollapsiblePanel.css file with class names of your own because the behaviors are hard coded as part of the Spry framework. To override the default class with your class name, send the new values as arguments to the collapsible panel constructor:*

```
<script type="text/javascript">
    var CP2 = new Spry.Widget.CollapsiblePanel("CollapsiblePanel2", { hoverClass: "hover",
openClass: "open", closedClass: "closed", focusedClass: "focused" });
</script>
```

Turn off panel animation

By default, a collapsible panel uses animation to smoothly open and close.

❖ To turn off animation so that the panel instantly opens and closes, send an argument in the collapsible panel constructor, as follows:

```
<script type="text/javascript">
    var CP5 = new Spry.Widget.CollapsiblePanel("CollapsiblePanel5", { enableAnimation: false
});
</script>
```

Set panel animation timing

You can change the time it takes for a panel to open. Set the time in milliseconds (1000 = 1 second). By default, Spry uses 500 milliseconds.

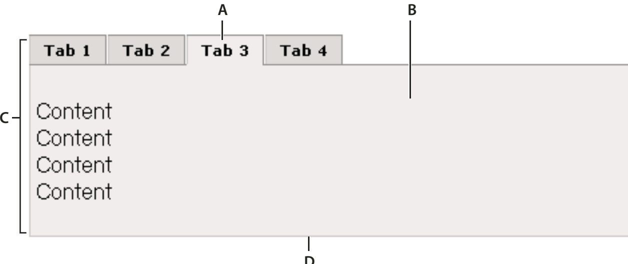
❖ Add the `duration` option to the constructor:

```
<script type="text/javascript">
    var CP9 = new Spry.Widget.CollapsiblePanel("CollapsiblePanel9", { duration: 100 });
</script>
```

Working with the Tabbed Panels widget

Tabbed Panels widget overview and structure

A Tabbed Panels widget is a set of panels that can store content in a compact space. Site viewers hide or reveal the content stored in the Tabbed Panels by clicking the tab of the panel they want to access. The panels of the widget open accordingly as the visitor clicks different tabs. In a Tabbed Panels widget, only one content panel is open at a given time. The following example shows a Tabbed Panels widget, with the third panel open.



***A.** Tab* ***B.** Content* ***C.** Tabbed Panels widget* ***D.** Tabbed panel*

The HTML code for the Tabbed Panels widget is made up of an outer `div` tag that contains all of the panels, a list for the tabs, a `div` tag to contain the content panels, and a `div` tag for each content panel. The HTML code for the Tabbed Panels widget also includes `script` tags in the head of the document and after the Tabbed Panel widget's HTML code.

The `script` tag in the head of the document defines all of the JavaScript functions related to the Tabbed Panel widget. The `script` tag after the Tabbed Panel widget code creates a JavaScript object that makes the Tabbed Panel interactive. Following is the HTML code for a Tabbed Panel widget:

```
<head>
. . .
    <!--Link the CSS style  sheet that styles the tabbed panel-->
    <link href="SpryAssets/SpryTabbedPanels.css" rel="stylesheet" type="text/css" />
    <!--Link the Spry TabbedPanels JavaScript library-->
    <script src="SpryAssets/SpryTabbedPanels.js" type="text/javascript"></script>
</head>
<body>
    <!--Create the Tabbed Panel widget and assign classes to each element-->
    <div class="TabbedPanels" id="TabbedPanels1">
        <ul class="TabbedPanelsTabGroup">
            <li class="TabbedPanelsTab">Tab 1</li>
            <li class="TabbedPanelsTab">Tab 2</li>
            <li class="TabbedPanelsTab">Tab 3</li>
            <li class="TabbedPanelsTab">Tab 4</li>
        </ul>
        <div class="TabbedPanelsContentGroup">
            <div class="TabbedPanelsContent">Tab 1 Content</div>
            <div class="TabbedPanelsContent">Tab 2 Content</div>
            <div class="TabbedPanelsContent">Tab 3 Content</div>
            <div class="TabbedPanelsContent">Tab 4 Content</div>
        </div>
    </div>
<!--Initialize the Tabbed Panel widget object-->
<script type="text/javascript">
    var TabbedPanels1 = new Spry.Widget.TabbedPanels("TabbedPanels1");
</script>
</body>
```

In the code, the `new` JavaScript operator initializes the Tabbed Panel widget object, and transforms the `div` content with the ID of `TabbedPanels1` from static HTML code into an interactive page element. The `Spry.Widget.TabbedPanels` method is a constructor in the Spry framework that creates Tabbed Panel objects, and the information necessary to initialize the object is contained in the SpryTabbedPanels.js JavaScript library that you linked to in the head of the document.

Each of the elements in the Tabbed Panel widget contains a CSS class. These classes control the style of the Tabbed Panel widget, and exist in the accompanying SpryTabbedPanels.css file.

You can change the appearance of any given part of the Tabbed Panel widget by editing the CSS rule that corresponds to the class names assigned to it in the HTML code. For example, to change the background color of the Tabbed Panel's tabs, edit the `TabbedPanelsTab` rule in the SpryTabbedPanels.css file. Keep in mind that changing the CSS code in the SpryTabbedPanels.css file will affect all tabbed panels that are linked to that file.

In addition to the classes shown in the HTML code, the Tabbed Panel widget includes certain default behaviors that are attached to the widget. These behaviors are a built-in part of the Spry framework, and are in the SpryTabbedPanels.js JavaScript library file. The Tabbed Panel library includes behaviors related to hovering, tab clicking (to open panels), panel focus, and keyboard navigation.

You can change the look of the Tabbed Panel as it relates to these behaviors by editing the appropriate classes in the SpryTabbedPanels.css file. If you want to remove a given behavior, you can delete the CSS rules that correspond to that behavior.

Note: *While you can change the look of the Tabbed Panel as it relates to a certain behavior, you cannot alter the built-in behaviors themselves. For example, Spry still places a TabbedPanelsContentVisible class on the currently open panel, even if no properties are set for the TabbedPanelsContentVisible class in the SpryTabbedPanels.css file.*

Variation on tags used for Tabbed Panels widget structure

In the preceding example, `div` tags and list items create a nested tag structure for the widget:

```
Container <div>
    Tabs <ul>
        Tab <li>
    Content container <div>
        Content <div>
```

This 3-level, 5-container structure is essential for the Tabbed Panels widget to work properly; the structure, however, is more important than the actual tags you decide to use. Spry reads the structure (not `div` tags necessarily) and builds the widget accordingly. As long as the 3-level, 4-container structure is in place, you can use any block-level element to create the widget:

```
Container <div>
    Tabs <div>
        Tab <h3>
    Content container <div>
        Content <p>
```

The `div` tags in the example are flexible and can contain other block-level elements. A `p` tag (or any other inline tag), however, cannot contain other block-level elements, so you cannot use it as a container or panel tag for the tabbed panel.

CSS code for the Tabbed Panels widget

The SpryTabbedPanels.css file contains the rules that style the Tabbed Panels widget. You can edit these rules to style the tabbed panels' look and feel. The names of the rules in the CSS file correspond directly to the names of the classes specified in the Tabbed Panels widget's HTML code.

***Note:** You can replace style-related class names in the SpryTabbedPanels.css file and HTML code with class names of your own. Doing so does not affect the functionality of the widget, as CSS code is not required for widget functionality.*

The default functionality for the behaviors classes at the end of the style sheet are defined in the SpryTabbedPanels.js JavaScript library file. You can change the default functionality by adding properties and values to the behavior rules in the style sheet.

The following is the CSS code for the SpryTabbedPanels.css file. The first half of the file contains styling rules for horizontal tabbed panels. The second half of the file contains styling rules for vertical tabbed panels.

```
/* Horizontal Tabbed Panels */
.TabbedPanels {
    margin: 0px;
    padding: 0px;
    clear: both;
    width: 100%; /* IE Hack to force proper layout when preceded by a paragraph. (hasLayout Bug)*/
}
.TabbedPanelsTabGroup {
    margin: 0px;
    padding: 0px;
}
.TabbedPanelsTab {
    position: relative;
    top: 1px;
    float: left;
    padding: 4px 10px;
    margin: 0px 1px 0px 0px;
    font: bold 0.7em sans-serif;
    background-color: #DDD;
    list-style: none;
    border-left: solid 1px #CCC;
    border-bottom: solid 1px #999;
    border-top: solid 1px #999;
    border-right: solid 1px #999;
    -moz-user-select: none;
    -khtml-user-select: none;
    cursor: pointer;
}
.TabbedPanelsTabHover {
    background-color: #CCC;
}
.TabbedPanelsTabSelected {
    background-color: #EEE;
    border-bottom: 1px solid #EEE;
}
.TabbedPanelsTab a {
    color: black;
    text-decoration: none;
}
.TabbedPanelsContentGroup {
    clear: both;
    border-left: solid 1px #CCC;
    border-bottom: solid 1px #CCC;
    border-top: solid 1px #999;
    border-right: solid 1px #999;
    background-color: #EEE;
}
.TabbedPanelsContent {
    padding: 4px;
}
.TabbedPanelsContentVisible {
}
/* Vertical Tabbed Panels */
.VTabbedPanels .TabbedPanelsTabGroup {
    float: left;
```

```
    width: 10em;
    height: 20em;
    background-color: #EEE;
    position: relative;
    border-top: solid 1px #999;
    border-right: solid 1px #999;
    border-left: solid 1px #CCC;
    border-bottom: solid 1px #CCC;
}
.VTabbedPanels .TabbedPanelsTab {
    float: none;
    margin: 0px;
    border-top: none;
    border-left: none;
    border-right: none;
}
.VTabbedPanels .TabbedPanelsTabSelected {
    background-color: #EEE;
    border-bottom: solid 1px #999;
}
.VTabbedPanels .TabbedPanelsContentGroup {
    clear: none;
    float: left;
    padding: 0px;
    width: 30em;
    height: 20em;
}
```

The SpryTabbedPanels.css file contains extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Insert the Tabbed Panels widget

1 Locate the SpryTabbedPanels.js file and add it to your web site. You can find the SpryTabbedPanels.js file in the widgets/tabbedpanels directory, located in the Spry directory that you downloaded from Adobe Labs. For more information, see “Prepare your files” on page 3.

For example, create a folder called *SpryAssets* in the root folder of your web site, and move the SpryTabbedPanels.js file to it. The SpryTabbedPanels.js file contains all of the information necessary for making the Tabbed Panels widget interactive.

2 Locate the SpryTabbedPanels.css file and add it to your web site. You might choose to add it to the main directory, a SpryAssets directory, or to a CSS directory if you have one.

3 Open the web page to add the Tabbed Panels widget to and link the SpryTabbedPanels.js file to the page by inserting the following `script` tag in the page’s head tag:

```
<script src="SpryAssets/SpryTabbedPanels.js" type="text/javascript"></script>
```

Make sure that the file path to the SpryTabbedPanels.js file is correct. This path varies depending on where you’ve placed the file in your web site.

4 Link the SpryTabbedPanels.css file to your web page by inserting the following `link` tag in the page’s head tag:

```
<link href="SpryAssets/SpryTabbedPanels.css" rel="stylesheet" type="text/css" />
```

Make sure that the file path to the SpryTabbedPanels.css file is correct. This path varies depending on where you’ve placed the file in your web site.

5 To add the tabbed panels widget to your web page, insert the following `div` tag where you want the accordion to appear on the page:

```
<div id="TabbedPanels1" class="TabbedPanels">
</div>
```

6 To add tabs to the tabbed panel, insert a `ul class="TabbedPanelsTabGroup"` tag inside the `div id="TabbedPanels1"...` tag, and an `li class="TabbedPanelsTab"` tag for each tab to add, as follows:

```
<div class="TabbedPanels" id="TabbedPanels1">
    <ul class="TabbedPanelsTabGroup">
        <li class="TabbedPanelsTab">Tab 1</li>
        <li class="TabbedPanelsTab">Tab 2</li>
        <li class="TabbedPanelsTab">Tab 3</li>
        <li class="TabbedPanelsTab">Tab 4</li>
    </ul>
</div>
```

The preceding code adds four tabs to the widget. You can add unlimited tabs.

7 To add a content area (or panel) for each of the tabs, insert a `div class="TabbedPanelsContentGroup"` container tag after the `ul` tag, and a `div class="TabbedPanelsContent"` tag for each content panel, as follows:

```
<div class="TabbedPanels" id="TabbedPanels1">
    <ul class="TabbedPanelsTabGroup">
        <li class="TabbedPanelsTab">Tab 1</li>
        <li class="TabbedPanelsTab">Tab 2</li>
        <li class="TabbedPanelsTab">Tab 3</li>
        <li class="TabbedPanelsTab">Tab 4</li>
    </ul>
    <div class="TabbedPanelsContentGroup">
        <div class="TabbedPanelsContent">Tab 1 Content</div>
        <div class="TabbedPanelsContent">Tab 2 Content</div>
        <div class="TabbedPanelsContent">Tab 3 Content</div>
        <div class="TabbedPanelsContent">Tab 4 Content</div>
    </div>
</div>
```

Insert the content between the opening and closing `TabbedPanelsContent` tags (for example, `Tab 1 Content`, as in the preceding example). The content can be any valid HTML code, including HTML form elements

8 To initialize the Spry tabbed panels object, insert the following `script` block after the HTML code for the Tabbed Panels widget:

```
<div id="TabbedPanels1" class="TabbedPanels">
. . .
</div>
<script type="text/javascript">
    var TabbedPanels1 = new Spry.Widget.TabbedPanels("TabbedPanels1");
</script>
```

The `new` JavaScript operator initializes the Tabbed Panels widget object, and transforms the `div` content with the ID of `TabbedPanels1` from static HTML code into an interactive tabbed panels object. The `Spry.Widget.TabbedPanels` method is a constructor in the Spry framework that creates tabbed panels objects. The information necessary to initialize the object is contained in the SpryTabbedPanels.js JavaScript library that you linked to at the beginning of this procedure.

Make sure that the ID of the tabbed panels' `div` tag matches the ID parameter you specified in the `Spry.Widgets.TabbedPanels` method. Make sure that the JavaScript call comes after the HTML code for the widget.

9 Save the page.

The complete code looks as follows:

```
<head>
. . .
    <link href="SpryAssets/SpryTabbedPanels.css" rel="stylesheet" type="text/css" />
    <script src="SpryAssets/SpryTabbedPanels.js" type="text/javascript"></script>
</head>
<body>
    <div class="TabbedPanels" id="TabbedPanels1">
        <ul class="TabbedPanelsTabGroup">
            <li class="TabbedPanelsTab">Tab 1</li>
            <li class="TabbedPanelsTab">Tab 2</li>
            <li class="TabbedPanelsTab">Tab 3</li>
            <li class="TabbedPanelsTab">Tab 4</li>
        </ul>
        <div class="TabbedPanelsContentGroup">
            <div class="TabbedPanelsContent">Tab 1 Content</div>
            <div class="TabbedPanelsContent">Tab 2 Content</div>
            <div class="TabbedPanelsContent">Tab 3 Content</div>
            <div class="TabbedPanelsContent">Tab 4 Content</div>
        </div>
    </div>
<script type="text/javascript">
    var TabbedPanels1 = new Spry.Widget.TabbedPanels("TabbedPanels1");
</script>
</body></body>
```

Add a panel to a Tabbed Panels widget

❖ Add an `li class="TabbedPanelsTab"` tag to the `ul` list of tabs, and a `div class="TabbedPanelsContent"` tag to the panel content container `div` tag. Do not forget to add the closing `/li` and `/div` tags when you add the code. For example:

```
<div class="TabbedPanels" id="TabbedPanels1">
    <ul class="TabbedPanelsTabGroup">
        <li class="TabbedPanelsTab">Tab 1</li>
        <li class="TabbedPanelsTab">Tab 2</li>
    </ul>
    <div class="TabbedPanelsContentGroup">
        <div class="TabbedPanelsContent">Tab 1 Content</div>
        <div class="TabbedPanelsContent">Tab 2 Content</div>
    </div>
</div>
```

You can add unlimited panels. The ratio between the number of `TabbedPanelsTabli` items and the number of `TabbedPanelsContentdiv` tags must always be 1:1.

Delete a panel from a Tabbed Panels widget

❖ Delete the desired `li class="TabbedPanelsTab"` tag and corresponding `<div class="TabbedPanels-Content">` from the code. Do not forget to delete the closing `</li>` and `</div>` tags when you delete the code.

Enable keyboard navigation

Making widgets accessible for keyboard navigation is an important part of every widget. Keyboard navigation lets the user control the widget with arrow keys or custom keys.

The foundation of keyboard navigation is the `tabIndex` attribute. This attribute tells the browser how to use the tabs to navigate through the document.

❖ To enable keyboard navigation on the tabbed panels, add a `TabIndex` value to each `li` tag, as follows:

```
<ul class="TabbedPanelsTabGroup">
    <li class="TabbedPanelTab" tabIndex="0">Tab</li>
    <li class="TabbedPanelTab" tabIndex="0">Tab</li>
</ul>
```

If the `tabIndex` attribute has a value of zero (0), the browser determines the order. If it has a positive integer value, that order value is used.

***Note:** Using tabIndex on a div tag is not XHTML 1.0 compliant.*

Change the orientation of tabbed panels

By default, tabbed panels appear horizontally, but you can easily create vertical tabbed panels as well.

❖ To change from a horizontal to a vertical Tabbed Panel widget, change the class on the main container `div` tag from TabbedPanels to VTabbedPanels, as follows:

```
<div class="VTabbedPanels" id="TabbedPanels1">
    <ul class="TabbedPanelsTabGroup">
        <li class="TabbedPanelsTab">Tab 1</li>
        <li class="TabbedPanelsTab">Tab 2</li>
. . .
</div>
```

Set default open panel

You can set a panel to be open when the page containing the Tabbed Panels widget loads in a browser.

❖ Set the defaultTab option in the constructor as follows:

```
<script type="text/javascript">
    var TabbedPanels1 = new Spry.Widget.TabbedPanels("TabbedPanels1", { defaultTab: 2 });
</script>
```

***Note:** The Tabbed Panels widget uses a zero-based counting system, so setting the value to 2 opens the third tabbed panel.*

Open panels programatically

Use JavaScript functions to programatically open specific panels. For example, you might have a button on your page that opens a particular tabbed panel when the user clicks the button.

Remember, Spry uses a zero-based counting system, so 0 indicates the first, leftmost tabbed panel. If the tabbed panel has an ID, you can also use the ID to refer to panels.

❖ Use the following functions to open specific tabbed panels:

```
<button onclick="TabbedPanels1.showPanel(0)" >open first panel</button>
<button onclick="TabbedPanels1.showPanel('tabID')">open panel</button>
<script type="text/javascript">
    var TabbedPanels1 = new Spry.Widget.TabbedPanels("TabbedPanels1");
</script>
```

Customize the Tabbed Panels widget

The SpryTabbedPanels.css file provides the default styling for the Tabbed Panels widget. You can, however, easily customize the widget by changing the appropriate CSS rule. The CSS rules in the SpryTabbedPanels.css file use the same class names as the related elements in the accordion's HTML code, so it's easy for you to know which CSS rules correspond to the different sections of the Tabbed Panels widget. Additionally, the SpryTabbedPanels.css file contains class names for behaviors that are related to the widget (for example, hovering and clicking behaviors).

The SpryTabbedPanels.css file should already be a included in your website before you start customizing. For more information, see "Prepare your files" on page 3.

Note: *Internet Explorer up to and including version 6 does not support sibling and child contextual selectors (for example,* `.TabbedPanels + .TabbedPanels` *or* `.TabbedPanels > .TabbedPanels`*).*

Style an Tabbed Panels widget (general instructions)

Set properties for the entire Tabbed Panels widget container, or set properties for the components of the widget individually.

1 Open the SpryTabbedPanels.css file.

2 Locate the CSS rule for the part of the tabbed panel to change. For example, to change the background color of the tabbed panels' tabs, edit the `TabbedPanelsTab` rule in the SpryTabbedPanels.css file.

3 Make your changes to the CSS rule and save the file.

You can replace style-related class names in the SpryTabbed Panels.css file and HTML code with class names of your own. Doing so does not affect the functionality of the widget.

The SpryAccordion.css file contains extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Style Tabbed Panels widget text

Set properties for the entire Tabbed Panels widget container, or set properties for the components of the widget individually.

❖ Use the following table to locate the appropriate CSS rule, and then add your own text-styling properties and values.

Text to change	Relevant CSS rule	Example of properties and values to add
Text in the entire widget	.TabbedPanels	font: Arial; font-size:medium;
Text in panel tabs only	.TabbedPanelsTabGroup or .Tabbed-PanelsTab	font: Arial; font-size:medium;
Text in content panels only	.TabbedPanelsContentGroup or .TabbedPanelsContent	font: Arial; font-size:medium;

Change Tabbed Panels widget background colors

❖ Use the following table to locate the appropriate CSS rule, and then add or change background-color properties and values.

Color to change	Relevant CSS rule	Example of property and value to add or change
Background color of panel tabs	.TabbedPanelsTabGroup or .Tabbed-PanelsTab	background-color: #DDD; (This is the default value.)
Background color of content panels	.Tabbed PanelsContentGroup or .TabbedPanelsContent	background-color: #EEE; (This is the default value.)
Background color of selected tab	.TabbedPanelsTabSelected	background-color: #EEE; (This is the default value.)
Background color of panel tabs when the mouse pointer moves over them	.TabbedPanelsTabHover	background-color: #CCC; (This is the default value.)

Constrain the width of tabbed panels

By default, the Tabbed Panels widget expands to fill available space. To constrain the width of a Tabbed Panels widget, set a width property for the accordion container.

1 Locate the `TabbedPanels` CSS rule in the SpryTabbedPanels.css file. This rule defines properties for the main container element of the Tabbed Panels widget.

2 Add a width property and value to the rule, for example `width: 300px;`.

Change tabbed panels height

By default, the height of tabbed panels expands according to content. To set a specific height for panels, add a height property to the `TabbedPanelsContent` rule.

1 Locate the `TabbedPanelsContent` CSS rule in the SpryTabbedPanels.css file.

2 Add a height property and value to the rule, for example `width: 300px;`.

Change tabbed panels behaviors

The Tabbed Panels widget includes a few predefined behaviors. These behaviors consist of changing CSS classes when a particular event occurs. For example, when a mouse pointer hovers over a panel tab, Spry applies the Tabbed-PanelsTabHover class to the widget. (This behavior is similar to `a:hover` for links.) The behaviors are blank by default; to change them, add properties and values to the rules.

1 Open the SpryTabbedPanels.css file and locate the CSS rule for the tabbed panels behavior to change. The Tabbed Panels widget includes four built-in rules for behaviors.

Behavior	Description
.Tabbed PanelsTabHover	Activates when hovering over the panel tab
.Tabbed PanelsTabFocused	Activates when a tab has keyboard focus
.Tabbed PanelsTabSelected	Activates on currently selected tab
.TabbedPanelsContentVisible	Activates on content area of currently selected tab

For examples, see the Tabbed Panels sample file located in the samples directory of the Spry directory that you downloaded from Adobe Labs. For more information, see “Prepare your files” on page 3.

2 Add CSS rules for any of the behaviors you want to enable.

Note: *You cannot replace behavior-related class names in the SpryTabbedPanels.css file with class names of your own because the behaviors are hard coded as part of the Spry framework. To override the default class with your class names, send the new values as arguments to the tabbed panels constructor:*

```
<script type="text/javascript">
    var TabbedPanels1 = new Spry.Widget.TabbedPanels("TabbedPanels1", { tabHoverClass:
"hover", panelVisibleClass: "open", tabSelectedClass: "selected", tabFocusedClass:
"focused" });
</script>
```

Working with the Menu Bar widget

Menu Bar widget overview and structure

A Menu Bar widget is a set of navigational menu buttons that display submenus when a mouse pointer hovers over one of the buttons. Menu Bars let you display a large amount of navigational information in a compact space, and also give visitors to the site a sense of what is available on the site without having to browse it extensively.

The following example shows a horizontal Menu Bar widget, with the third menu item expanded.

Menu Bar widget (consists of <ul>, <li>, and <a> tags)
A. *Menu item has submenu* ***B.*** *Submenu item has submenu*

The HTML code for the Menu Bar widget consists of an outer `ul` tag that contains an `li` tag for each of the top-level menu items. The top-level menu items (`li` tags) in turn contain `ul` and `li` tags that define submenus for each of the items, and submenus can likewise contain submenus. Top-level menus and submenus can contain an unlimited number of submenu items.

***Note:** As a best practice, try not to list every page on your site in the various levels of a menu bar. If a user has JavaScript turned off in their browser (and many do), the user can see only the top-level menu items of the menu bar.*

The HTML code for the Menu Bar widget also includes `script` tags in the head of the document and after the Menu Bar's HTML code. These tags create a JavaScript object, which makes the Menu Bar interactive. You define whether the Menu Bar widget will be vertical or horizontal in the main container `ul` tag for the Menu Bar. Following is the HTML code for a horizontal Menu Bar widget:

```
<head>
...
<!--Link the Spry Manu Bar JavaScript library-->
<script src="SpryAssets/SpryMenuBar.js" type="text/javascript"></script>
<!--Link the CSS style sheet that styles the menu bar. You can select between horizontal and
vertical-->
<link href="SpryAssets/SpryMenuBarHorizontal.css" rel="stylesheet" type="text/css" />
</head>
<body>
<!--Create a Menu bar widget and assign classes to each element-->
<ul id="menubar1" class="MenuBarHorizontal">
    <li><a class="MenuBarItemSubmenu" href="#">Item 1</a>
        <ul>
            <li><a href="#">Item 1.1</a></li>
            <li><a href="#">Item 1.2</a></li>
            <li><a href="#">Item 1.3</a></li>
        </ul>
    </li>
    <li><a href="#">Item 2</a></li>
    <li><a class="MenuBarItemSubmenu" href="#">Item 3</a>
        <ul>
            <li><a class="MenuBarItemSubmenu" href="#">Item 3.1</a>
                <ul>
                    <li><a href="#">Item 3.1.1</a></li>
                    <li><a href="#">Item 3.1.2</a></li>
                </ul>
            </li>
            <li><a href="#">Item 3.2</a></li>
            <li><a href="#">Item 3.3</a></li>
        </ul>
    </li>
    <li><a href="#">Item 4</a></li>
</ul>
<!--Initialize the Menu Bar widget object-->
<script type="text/javascript">
    var menubar1 = new Spry.Widget.MenuBar("menubar1",
{imgDown:"SpryAssets/SpryMenuBarDownHover.gif",
imgRight:"SpryAssets/SpryMenuBarRightHover.gif"});
</script>
</body>
```

In the code, the `new` JavaScript operator initializes the Menu Bar object, and transforms the `ul` content with the ID of `menubar1` from static HTML code into an interactive page element. The `Spry.Widget.MenuBar` method is a constructor in the Spry framework that creates Menu Bar objects, and the information necessary to initialize the object is contained in the MenuBar.js JavaScript library that you linked to in the head of the document.

Many of the a tags that create the widget contain a CSS class. These classes control the style of the Menu Bar widget, and exist in the accompanying CSS file, SpryMenuBarHorizontal.css or SpryMenuBarVertical.css, depending on your selection.

You can change the appearance of any given part of the Menu Bar widget by editing the CSS rule that corresponds to the class names assigned to it in the HTML code. For example, to change the background color of the Menu Bar's top-level menu items, edit the corresponding CSS rule in the SpryMenuBarHorizontal.css file. Keep in mind that changing the CSS code in the SpryManuBarHorizontal.css file will affect all menu bars that are linked to that file.

In addition to the classes shown in the HTML code, the Menu Bar widget includes certain default behaviors that are attached to the widget. These behaviors are a built-in part of the Spry framework, and are in the SpryMenuBar.js JavaScript library file. The library file includes behaviors related to hovering.

You can change the look of the Menu Bar as it relates to these behaviors by editing the appropriate classes in one of the CSS files. If for some reason you want to remove a given behavior, you can delete the CSS rules that correspond to that behavior.

CSS code for the Menu Bar widget

The SpryMenuBarHorizontal.css and SpryMenuBarVertical.css files contain the rules that style the Menu Bar widget. You can edit these rules to style the menu bar's look and feel. The names of the rules in the CSS file correspond directly to the names of the classes specified in the Menu Bar widget's HTML code.

Note: *You can replace style-related class names in the SpryMenuBarHorizontal.css and SpryMenuBarVertical.css files and HTML code with class names of your own. Doing so does not affect the functionality of the widget, as CSS code is not required for widget functionality.*

The default functionality for the behaviors classes at the end of the style sheet are defined in the SpryMenuBar.js JavaScript library file. You can change the default functionality by adding properties and values to the behavior rules in the style sheet.

The following is the CSS code for the SpryMenuBarHorizontal.css file:

```
/*Menu Bar styling classes*/
ul.MenuBarHorizontal{
    margin: 0;
    padding: 0;
    list-style-type: none;
    font-size: 100%;
    cursor: default;
    width: auto;
}
ul.MenuBarActive{
    z-index: 1000;
}
ul.MenuBarHorizontal li{
    margin: 0;
    padding: 0;
    list-style-type: none;
    font-size: 100%;
    position: relative;
    text-align: left;
    cursor: pointer;
    width: 8em;
    float: left;
```

```
}
ul.MenuBarHorizontal ul{
    margin: 0;
    padding: 0;
    list-style-type: none;
    font-size: 100%;
    z-index: 1020;
    cursor: default;
    width: 8.2em;
    position: absolute;
    left: -1000em;
}
ul.MenuBarHorizontal ul.MenuBarSubmenuVisible{
    left: auto;
}
ul.MenuBarHorizontal ul li{
    width: 8.2em;
}
ul.MenuBarHorizontal ul ul{
    position: absolute;
    margin: -5% 0 0 95%;
}
ul.MenuBarHorizontal ul.MenuBarSubmenuVisible ul.MenuBarSubmenuVisible{
    left: auto;
    top: 0;
}
ul.MenuBarHorizontal ul{
    border: 1px solid #CCC;
}
ul.MenuBarHorizontal a{
    display: block;
    cursor: pointer;
    background-color: #EEE;
    padding: 0.5em 0.75em;
    color: #333;
    text-decoration: none;
}
ul.MenuBarHorizontal a:hover, ul.MenuBarHorizontal a:focus{
    background-color: #33C;
    color: #FFF;
}
ul.MenuBarHorizontal a.MenuBarItemHover, ul.MenuBarHorizontal a.MenuBarItemSubmenuHover,
ul.MenuBarHorizontal a.MenuBarSubmenuVisible{
    background-color: #33C;
    color: #FFF;
}
ul.MenuBarHorizontal a.MenuBarItemSubmenu{
    background-image: url(SpryMenuBarDown.gif);
    background-repeat: no-repeat;
    background-position: 95% 50%;
}
ul.MenuBarHorizontal ul a.MenuBarItemSubmenu{
    background-image: url(SpryMenuBarRight.gif);
    background-repeat: no-repeat;
    background-position: 95% 50%;
```

```
}
ul.MenuBarHorizontal a.MenuBarItemSubmenuHover{
    background-image: url(SpryMenuBarDownHover.gif);
    background-repeat: no-repeat;
    background-position: 95% 50%;
}
ul.MenuBarHorizontal ul a.MenuBarItemSubmenuHover{
    background-image: url(SpryMenuBarRightHover.gif);
    background-repeat: no-repeat;
    background-position: 95% 50%;
}
ul.MenuBarHorizontal iframe{
    position: absolute;
    z-index: 1010;
}
@media screen, projection{
    ul.MenuBarHorizontal li.MenuBarItemIE{
        display: inline;
        f\loat: left;
        background: #FFF;
    }
}
```

The SpryMenuBar.css file contains extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Insert the Menu Bar widget

1 Locate the SpryMenuBar.js file and add it to your web site. You can find the SpryMenuBar.js file in the widgets/menubar directory, located in the Spry directory that you downloaded from Adobe Labs. For more information, see “Prepare your files” on page 3.

For example, create a folder called *SpryAssets* in the root folder of your web site, and move the SpryMenuBar.js file to it. The SpryMenuBar.js file contains all of the information necessary for making the Menu Bar widget interactive.

2 Locate the SpryMenuBarHorizontal.css or SpryMenuBarVertical.css file (depending on which kind of menu bar you want to create), and add it to your web site. You might choose to add it to the main directory, a SpryAssets directory, or to a CSS directory if you have one.

3 Open the web page to add the Menu Bar widget to and link the SpryMenuBar.js file to the page by inserting the following `script` tag in the page’s head tag:

```
<script src="SpryAssets/SpryMenuBar.js" type="text/javascript"></script>
```

Make sure that the file path to the SpryMenuBar.js file is correct. This path varies depending on where you’ve placed the file in your web site.

4 Link the SpryMenuBarHorizontal.css or SpryMenuBarVertical.css file to your web page by inserting the following `link` tag in the page’s head tag:

```
<link href="SpryAssets/SpryMenuBarHorizontal.css" rel="stylesheet" type="text/css" />
```

Make sure that the file path to the SpryMenuBarHorizontal.css or SpryMenuBarVertical.css file is correct. This path varies depending on where you’ve placed the file in your web site.

5 Add the menu bar HTML code to your web page by inserting a `ul` tag as a container tag, and then `li` tags with some sample text for each top-level menu item in the menu bar, as follows:

```
<body>
    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
        <li>Item 4</li>
    </ul>
</body>
```

6 Wrap the text of the `li` tags with `a` tags:

```
<body>
    <ul>
        <li><a href="#">Item 1</a></li>
        <li><a href="#">Item 2</a></li>
        <li><a href="#">Item 3</a></li>
        <li><a href="#">Item 4</a></li>
    </ul>
</body>
```

7 Nest another unordered list (including `a` tags) in the third menu item (or any other menu item), as follows:

```
<body>
    <ul>
        <li><a href="#">Item 1</a></li>
        <li><a href="#">Item 2</a></li>
        <li><a href="#">Item 3</a>
            <ul>
                <li><a href="#">Submenu Item 1</a></li>
                <li><a href="#">Submenu Item 2</a></li>
            </ul>
        </li>
        <li><a href="#">Item 4</a></li>
    </ul>
</body>
```

This nested unordered list is the submenu for the third menu item. Make sure that the nested list is not within the `a` tag of the top-level menu item.

8 Add a unique ID that identifies the menu bar container (`ul` tag), as follows:

```
<body>
    <ul id="menubar1">
        <li><a href="#">Item 1</a></li>
        <li><a href="#">Item 2</a></li>
        <li><a href="#">Item 3</a>
            <ul>
                <li><a href="#">Submenu Item 1</a></li>
                <li><a href="#">Submenu Item 2</a></li>
            </ul>
        </li>
        <li><a href="#">Item 4</a></li>
    </ul>
</body>
```

Later, you'll use this ID to identify the container in the widget constructor.

9 To add styling to the menu bar, add the appropriate classes, as follows:

```
<body>
    <ul id="menubar1" class="MenuBarHorizontal">
        <li><a href="#">Item 1</a></li>
        <li><a href="#">Item 2</a></li>
        <li><a href="#" class="MenuBarItemSubmenu">Item 3</a>
            <ul>
                <li><a href="#">Submenu Item 1</a></li>
                <li><a href="#">Submenu Item 2</a></li>
            </ul>
        </li>
        <li><a href="#">Item 4</a></li>
    </ul>
</body>
```

Identify whether you're creating a horizontal menu bar or a vertical menu bar. Assign the MenuBarItemSubmenu class to `a` tags when top-level menu bar items have submenus. This class displays a down-arrow image that lets the user know submenu is present.

10 To initialize the Spry menu bar object, insert the following `script` block after the HTML code for the Menu Bar widget:

```
<ul id="menubar1" class="MenuBarHorizontal">
. . .
</ul>
<script type="text/javascript">
    var menubar1 = new Spry.Widget.MenuBar("menubar1");
</script>
```

The `new` JavaScript operator initializes the Menu Bar widget object, and transforms the `ul` content with the ID of `menubar1` from static HTML code into an interactive menu bar object. The `Spry.Widget.MenuBar` method is a constructor in the Spry framework that creates menu bar objects. The information necessary to initialize the object is contained in the SpryMenuBar.js JavaScript library that you linked to at the beginning of this procedure.

Make sure that the ID of the menu bar's `ul` container tag matches the ID parameter you specified in the `Spry.Widgets.MenuBar` method. Make sure that the JavaScript call comes after the HTML code for the widget.

11 Save the page.

The complete code looks as follows:

```
<head>
...
<script src="SpryAssets/SpryMenuBar.js" type="text/javascript"></script>
<link href="SpryAssets/SpryMenuBarHorizontal.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <ul id="menubar1" class="MenuBarHorizontal">
        <li><a href="#">Item 1</a></li>
        <li><a href="#">Item 2</a></li>
        <li><a href="#" class="MenuBarItemSubmenu">Item 3</a>
            <ul>
                <li><a href="#">Submenu Item 1</a></li>
                <li><a href="#">Submenu Item 2</a></li>
            </ul>
        </li>
        <li><a href="#">Item 4</a></li>
    </ul>
<script type="text/javascript">
    var menubar1 = new Spry.Widget.MenuBar("menubar1");
</script>
</body>
```

***Note:** The Spry Menu Bar widget uses DHTML layers to display sections of HTML code on top of other sections. If your page contains Flash content, this might cause a problem because Flash movies are always displayed on top of all other DHTML layers, so the Flash content might be displayed on top of your submenus. The workaround for this situation is to change the parameters for the Flash content to use `wmode="transparent"`. For more information, see www.adobe.com/go/15523.*

Add or delete menus and submenus

Add a main menu item

❖ To add a main menu item, add a new list item (`li` tag) to the container `ul` tag. For example:

```
<ul id="menubar1" class="MenuBarHorizontal">
    <li><a href="#">Item 1</a></li>
    <li><a href="#">Item 2</a></li>
    <li><a href="#" class="MenuBarItemSubmenu">Item 3</a>
        <ul>
            <li><a href="#">Submenu Item 1</a></li>
            <li><a href="#">Submenu Item 2</a></li>
        </ul>
    </li>
    <li><a href="#">Item 4</a></li>
    <li><a href="#">Item 5--NEW MENU ITEM</a></li>
</ul>
```

Add a submenu item

❖ To add a submenu item, add a new list item (`li` tag) to a submenu `ul` tag. For example:

```
<ul id="menubar1" class="MenuBarHorizontal">
    <li><a href="#">Item 1</a></li>
    <li><a href="#">Item 2</a></li>
    <li><a href="#" class="MenuBarItemSubmenu">Item 3</a>
        <ul>
            <li><a href="#">Submenu Item 1</a></li>
            <li><a href="#">Submenu Item 2</a></li>
            <li><a href="#">Submenu Item 3--NEW SUBMENU ITEM</a></li>
        </ul>
    </li>
    <li><a href="#">Item 4</a></li>
</ul>
```

Delete a main menu or submenu item

❖ Delete the `li` tag for the menu item or submenu item to delete.

Create a tool tip for a menu item

❖ To create a tooltip for a menu item, add a `title` attribute to the `a` tag of the relevant menu item. For example:

```
<ul id="menubar1" class="MenuBarHorizontal">
    <li><a href="#">Item 1</a></li>
    <li><a href="#">Item 2</a></li>
    <li><a href="#">Item 3</a></li>
    <li><a href="contacts.html" title="Contacts">Item 4</a></li>
</ul>
```

Preload images

❖ To preload the images used for the down and right submenu arrows, add either the `imgDown` option, the `imgRight` option, or both options to the widget constructor, as follows:

```
<script type="text/javascript">
    var menubar1 = new Spry.Widget.MenuBar("menubar1",
{imgDown:"SpryAssets/SpryMenuBarDownHover.gif",
imgRight:"SpryAssets/SpryMenuBarRightHover.gif"});
</script>
```

Add the correct path to the image as the value for the option. This path varies depending on where you store the images.

Change the orientation of a Menu Bar widget

You can change the orientation of a Menu Bar widget from horizontal to vertical, and vice versa. To do so, alter the HTML code for the menu bar and make sure you have the correct CSS file in your website.

The following procedure changes a horizontal Menu Bar widget to a vertical Menu Bar widget.

1 Make sure you have the SpryMenuBarVertical.css file in your website. (For example, you might store this file in a SpryAssets folder somewhere in the site.)

2 Replace the link to the SpryMenuBarHorizontal.css file with a link to the SpryMenuBarVertical.css file in the head of your document, as follows:

```
<link href="SpryAssets/SpryMenuBarVertical.css" rel="stylesheet" type="text/css" />
```

3 Locate the MenuBarHorizontal class in the HTML code for the horizontal menu bar, and change it to MenuBarVertical. The MenuBarHorizontal class is defined in the container `ul` tag for the menu bar (`<ul id="menubar1" class="MenuBarHorizontal">`).

4 After the code for the menu bar, locate the menu bar constructor:

```
var menubar1 = new Spry.Widget.MenuBar("menubar1",
{imgDown:"SpryAssets/SpryMenuBarDownHover.gif",
imgRight:"SpryAssets/SpryMenuBarRightHover.gif"});
```

5 Remove the `imgDown` preload option (and comma) from the constructor:

```
var menubar1 = new Spry.Widget.MenuBar("menubar1",
{imgRight:"SpryAssets/SpryMenuBarRightHover.gif"});
```

Note: *If you are converting from a vertical menu bar to a horizontal menu bar, add the* `imgDown` *preload option and comma instead.*

6 (Optional) If your page no longer contains any other horizontal Menu Bar widgets, delete the link to the former MenuBarHorizontal.css file in the head of the document.

7 Save the page.

Customize the Menu Bar widget

The SpryMenuBarHorizontal.css and SpryMenuBarVertical.css files provide the default styling for the Menu Bar widget. You can, however, customize the widget by changing the appropriate CSS rule. The CSS rules in the SpryMenuBarHorizontal.css and SpryMenuBarVertical.css files use the same class names as the related elements in the menu bar's HTML code, so it's easy for you to know which CSS rules correspond to the different sections of the Menu Bar widget. Additionally, the SpryMenuBarHorizontal.css and SpryMenuBarVertical.css files contain class names for behaviors that are related to the widget (for example, hovering and clicking behaviors).

The horizontal or vertical styling sheet for the widget should already be a included in your website before you start customizing. For more information, see "Prepare your files" on page 3.

Style a Menu Bar widget (general instructions)

You can style an Menu Bar widget by setting properties for the entire Menu Bar widget container, or by setting properties for the components of the widget individually.

1 Open the SpryMenuBarHorizontal.css or SpryMenuBarVertical.css file.

2 Locate the CSS rule for the part of the menu bar to change. For example, to change the background color of the top-level menu items, edit the `ulMenuBarHorizontal a` or `ulMenuBarVertical a` rule in the SpryAccordion.css file.

3 Make your changes to the CSS and save the file.

You can replace style-related class names in the CSS files and HTML code with class names of your own. Doing so does not affect the functionality of the widget.

The SpryMenuBarHorizontal.css and SpryMenuBarVertical.css files have extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Change text styling of a menu item

The CSS code attached to the `a` tag contains the information for text styling. Several relevant text-styling class values that pertain to different menu states are attached to the `a` tag.

❖ To change the text styling of a menu item, use the following table to locate the appropriate CSS rule, and then change the default value.

Style to change	CSS rule for vertical or horizontal menu bar	Relevant properties and default values
Default text	ul.MenuBarVertical a, ul.MenuBarHorizontal a	color: #333; text-decoration: none;
Text color when mouse pointer moves over it	ul.MenuBarVertical a:hover, ul.MenuBarHorizontal a:hover	color: #FFF;
Text color when in focus	ul.MenuBarVertical a:focus, ul.MenuBarHorizontal a:focus	color: #FFF;
Menu Bar item color when mouse pointer moves over it	ul.MenuBarVertical a.MenuBarItemHover, ul.MenuBarHorizontal a.MenuBarItemHover	color: #FFF;
Submenu item color when mouse pointer moves over it	ul.MenuBarVertical a.MenuBarItemSubmenuHover, ul.MenuBarHorizontal a.MenuBarItemSubmenuHover	color: #FFF;

Change the background color of a menu item

The CSS rule attached to the `a` tag contains the information for a menu item's background color. Several relevant background color class values are attached to the `a` tag that pertain to different menu states.

❖ To change the background color of a menu item, use the following table to locate the appropriate CSS rule, and then change the default value.

Color to change	CSS rule for vertical or horizontal menu bar	Relevant properties and default values
Default background	ul.MenuBarVertical a, ul.MenuBarHorizontal a	background-color: #EEE;
Background color when mouse pointer moves over it	ul.MenuBarVertical a:hover, ul.MenuBarHorizontal a:hover	background-color: #33C;
Background color when in focus	ul.MenuBarVertical a:focus, ul.MenuBarHorizontal a:focus	background-color: #33C;
Menu Bar item color when mouse pointer moves over it	ul.MenuBarVertical a.MenuBarItemHover, ul.MenuBarHorizontal a.MenuBarItemHover	background-color: #33C;
Submenu item color when mouse pointer moves over it	ul.MenuBarVertical a.MenuBarItemSubmenuHover, ul.MenuBarHorizontal a.MenuBarItemSubmenuHover	background-color: #33C;

Change the dimension of menu items

To change the dimension of menu items by changing the width properties of the menu item's `li` and `ul` tags.

1 Locate the `ul.MenuBarVertical li` or `ul.MenuBarHorizontal li` rule.

2 Change the width property to a desired width, or change the property to `auto` to remove a fixed width, and add the property and value `white-space: nowrap;` to the rule.

3 Locate the `ul.MenuBarVertical ul` or `ul.MenuBarHorizontal ul` rule.

4 Change the width property to a desired width (or change the property to `auto` to remove a fixed width).

5 Locate the `ul.MenuBarVertical ul li` or `ul.MenuBarHorizontal ul li` rule.

6 Add the following properties to the rule: `float: none;` and `background-color: transparent;`.

7 Delete the `width: 8.2em;` property and value.

Position submenus

The position of Spry Menu Bar submenus is controlled by the margin property on submenu `ul` tags.

1 Locate the `ul.MenuBarVertical ul` or `ul.MenuBarHorizontal ul` rule.

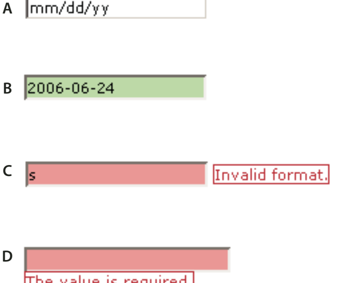
2 Change the `margin: -5% 0 0 95%;` default values to the desired values.

Working with the Validation Text Field widget

Validation Text Field widget overview and structure

A Spry Validation Text Field widget is a text field that displays valid or invalid states when the site visitor enters text. For example, you can add a Validation Text Field widget to a form in which visitors type their e-mail addresses. If they fail to type the @ sign and a period (.) in the e-mail address, the widget returns a message stating that the information the user entered is invalid.

The following example shows a Validation Text Field widget in various states:



A. Text Field widget, hint activated B. Text Field widget, valid state C. Text Field widget, invalid state D. Text Field widget, required state

The Validation Text Field widget includes a number of states (for example, valid, invalid, required value, and so on). You can alter the properties of these states by editing the corresponding CSS file (SpryValidationTextField.css), depending on the desired validation results. A Validation Text Field widget can validate at various points—for example, when the site visitor clicks outside the widget, when they type, or when they try to submit the form.

Initial state When the page loads in the browser, or when the user resets the form.

Focus state When the user places the insertion point in the widget.

Valid state When the user enters information correctly, and the form can be submitted.

Invalid state When the user enters text in an invalid format. (For example, 06 for a year instead of 2006).

Required state When the user fails to enter required text in the text field.

Minimum Number Of Characters state When the user enters fewer than the minimum number of characters required in the text field.

Maximum Number Of Characters state When the user enters greater than the maximum number of characters allowed in the text field.

Minimum Value state When the user enters a value that is less than the value that the text field requires. (Applies to integer, real, and data type validations.)

Maximum Value state When the user enters a value that is greater than the maximum value that the text field allows. (Applies to integer, real, and data type validations.)

Whenever a Validation Text Field widget enters one of these states through user interaction, the Spry framework logic applies a specific CSS class to the HTML container for the widget at run time. For example, if a user tries to submit a form, but enters no text in a required text field, Spry applies a class to the widget that causes it to display the error message, "A value is required," and blocks the form submission. The rules that control the style and display states of error messages exist in the SpryValidationTextField.css file that accompanies the widget.

The default HTML code for the Validation Text Field widget, usually inside a form, is made up of a container `span` tag that surrounds the `input` tag of the text field. The HTML code for the Validation Text Field widget also includes `script` tags in the head of the document and after the widget's HTML code. The `script` tag in the head of the document defines all of the JavaScript functions related to the Text Field widget. The `script` tag after the widget code creates a JavaScript object that makes the text field interactive.

Following is the HTML code for a Validation Text Field widget:

```
<head>
...
<!-- Link the Spry Validation Text Field JavaScript library -->
<script src="SpryAssets/SpryValidationTextField.js" type="text/javascript"></script>
<!-- Link the CSS style sheet that styles the widget -->
<link href="SpryAssets/SpryValidationTextField.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" name="form1" method="post" action="">
        <!-- Create the text field widget and assign a unique id-->
        <span id="sprytextfield1">
            <input type="text" name="mytextfield" id="mytextfield" />
            <!--Display an error message>
            <span class="textfieldRequiredMsg">A value is required.</span>
        </span>
    </form>
<!-- Initialize the Validation Text Field widget object-->
<script type="text/javascript">
var sprytextfield1 = new Spry.Widget.ValidationTextField("sprytextfield1");
</script>
</body>
```

In the code, the `new` JavaScript operator initializes the Text Field widget object, and transforms the `span` content with the ID of `sprytextfield1` from static HTML code into an interactive page element.

The `span` tag for the error message in the widget has a CSS class applied to it. This class (which is set to `display:none;` by default), controls the style and visibility of the error message, and exists in the accompanying CSS file SpryValidationTextField.css. When the widget enters different states as a result of user interaction, Spry places different classes on the container for the widget, which in turn affects the error-message class.

To add other error messages to a Validation Text Field widget, create a `span` tag (or any other type of tag) to hold the text of the error message. Then, by applying a CSS class to it, you can hide or show the message, depending on the widget state.

You can add other error messages to a Validation Text Field widget by creating the corresponding CSS rule in the SpryValidationTextField.css file. For example, to change the background color for a state, edit the corresponding rule or add a new rule (if it's not already present) in the style sheet.

Variation on tags used for Text Field widget structure

In the preceding example, we used `span` tags to create the structure for the widget:

```
Container SPAN
    INPUT type="text"
    Error message SPAN
```

You can, however, use almost any container tag to create the widget:

```
Container DIV
    INPUT type="text"
    Error Message P
```

Spry uses the tag ID (not the tag itself) to create the widget. Spry also displays error messages using CSS code that is indifferent to the actual tag used to contain the error message.

The ID passed into the widget constructor identifies a specific HTML element. The constructor finds this element and searches the identified container for a corresponding `input` tag. If the ID passed to the constructor is the ID of the `input` tag (rather than a container tag), the constructor attaches validation triggers directly to the `input` tag. If no container tag is present, however, the widget cannot display error messages, and different validation states alter only the appearance of the `input` tag element (for example, its background color).

***Note:** Multiple INPUT tags do not work inside the same HTML widget container. Each text field should be its own widget.*

CSS code for the Validation Text Field widget

The SpryValidationTextField.css file contains the rules that style the Validation Text Field widget and its error messages. You can edit these rules to style the look and feel of the widget and error messages. The names of the rules in the CSS file correspond to the names of the classes specified in the widget's HTML code.

The following is the CSS code for the SpryValidationTextField.css file:

```
/*Text Field styling classes*/
.textfieldRequiredMsg,
.textfieldInvalidFormatMsg,
.textfieldMinValueMsg,
.textfieldMaxValueMsg,
.textfieldMinCharsMsg,
.textfieldMaxCharsMsg,
.textfieldValidMsg {
    display: none;
}
.textfieldRequiredState .textfieldRequiredMsg,
.textfieldInvalidFormatState .textfieldInvalidFormatMsg,
.textfieldMinValueState .textfieldMinValueMsg,
.textfieldMaxValueState .textfieldMaxValueMsg,
.textfieldMinCharsState .textfieldMinCharsMsg,
.textfieldMaxCharsState .textfieldMaxCharsMsg {
    display: inline;
    color: #CC3333;
    border: 1px solid #CC3333;
}
.textfieldValidState input, input.textfieldValidState {
    background-color: #B8F5B1;
}
input.textfieldRequiredState, .textfieldRequiredState input,
input.textfieldInvalidFormatState, .textfieldInvalidFormatState input,
input.textfieldMinValueState, .textfieldMinValueState input,
input.textfieldMaxValueState, .textfieldMaxValueState input,
input.textfieldMinCharsState, .textfieldMinCharsState input,
input.textfieldMaxCharsState, .textfieldMaxCharsState input {
    background-color: #FF9F9F;
}
.textfieldFocusState input, input.textfieldFocusState {
    background-color: #FFFFCC;
}
.textfieldFlashText input, input.textfieldFlashText {
    color: red !important;
}
```

The SpryValidationTextField.css file also contains extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Insert the Validation Text Field widget

1 Locate the SpryValidationTextField.js file and add it to your web site. You can find the SpryValidationTextField.js file in the widgets/textfieldvalidation directory, located in the Spry directory that you downloaded from Adobe Labs. For more information, see “Prepare your files” on page 3.

For example, create a folder called *SpryAssets* in the root folder of your web site, and move the SpryValidationText-Field.js file to it. The SpryValidationTextField.js file contains all of the information necessary for making the Text Field widget interactive.

2 Locate the SpryValidationTextField.css file and add it to your web site. You might choose to add it to the main directory, a SpryAssets directory, or to a CSS directory if you have one.

3 Open the web page to add the Text Field widget to and link the SpryValidationTextField.js file to the page by inserting the following `script` tag in the page’s head tag:

```
<script src="SpryAssets/SpryValidationTextField.js" type="text/javascript"></script>
```

Make sure that the file path to the SpryValidationTextField.js file is correct. This path varies depending on where you've placed the file in your web site.

4 Link the SpryValidationTextField.css file to your web page by inserting the following `link` tag in the page's head tag:

```
<link href="SpryAssets/SpryValidationTextField.css" rel="stylesheet" type="text/css" />
```

Make sure that the file path to the SpryValidationTextField.css file is correct. This path varies depending on where you've placed the file in your web site.

5 Add a text field the page and give it a name and a unique ID:

```
<input type="text" name="mytextfield" id="mytextfield"/>
```

6 To add a container around the text field, insert `span` tags around the `input` tags, and assign a unique ID to the widget, as follows:

```
<span id="sprytextfield1">
    <input type="text" name="mytextfield" id="mytextfield"/>
</span>
```

7 Initialize the text field object by inserting the following `script` block after the HTML code for the widget:

```
<script type="text/javascript">
    var sprytextfield1 = new Spry.Widget.ValidationTextField("sprytextfield1");
</script>
```

The `new` JavaScript operator initializes the Text Field widget object, and transforms the `span` tag content with the ID of `sprytextfield1` from static HTML code into an interactive text field object. The `Spry.Widget.Validation-TextField` method is a constructor in the Spry framework that creates text field objects. The information necessary to initialize the object is contained in the SpryValidationTextField.js JavaScript library that you linked to at the beginning of this procedure.

Make sure that the ID of the text field's container `span` tag matches the ID parameter you specified in the `Spry.Widgets.ValidationTextField` method. Make sure that the JavaScript call comes after the HTML code for the widget.

8 Save the page.

The complete code looks as follows:

```
<head>
...
<script src="SpryAssets/SpryValidationTextField.js" type="text/javascript"></script>
<link href="SpryAssets/SpryValidationTextField.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <span id="sprytextfield1">
        <input type="text" name="mytextfield" id="mytextfield" />
    </span>
<script type="text/javascript">
    var sprytextfield1 = new Spry.Widget.ValidationTextField("sprytextfield1");
</script>
</body>
```

Display error messages

❖ Create a `span` tag (or any other type of tag) to display the error message, and assign the appropriate class to it, as follows:

```
<span id="sprytextfield1">
    <input type="text" name="mytextfield" id="mytextfield" />
    <span class="textfieldRequiredMsg">Please enter a description</span>
</span>
```

The `textfieldRequiredMsg` rule is located in the SpryValidationTextField.css file, and is set to `display:none` by default. When the widget enters a different state through user interaction, Spry applies the appropriate class—the state class—to the container of the widget. This action affects the error message class, and by extension, the appearance of the error message.

For example, the following shows a portion of the CSS rule from the SpryValidationTextField.css file:

```
.textfieldRequiredMsg,
.textfieldInvalidFormatMsg,
.textfieldMinValueMsg,
.textfieldMaxValueMsg,
.textfieldMinCharsMsg,
.textfieldMaxCharsMsg,
.textfieldValidMsg {
    display: none;
}
.textfieldRequiredState .textfieldRequiredMsg,
.textfieldInvalidFormatState .textfieldInvalidFormatMsg,
.textfieldMinValueState .textfieldMinValueMsg,
.textfieldMaxValueState .textfieldMaxValueMsg,
.textfieldMinCharsState .textfieldMinCharsMsg,
.textfieldMaxCharsState .textfieldMaxCharsMsg{
    display: inline;
    color: #CC3333;
    border: 1px solid #CC3333;
}
```

By default, no state class is applied to the widget container, so that when the page loads in a browser, the error message text in the preceding HTML code example only has the textfieldRequiredMsg class applied to it. (The property and value pair for this rule is `display:none`, so the message remains hidden.) If the user fails to enter text in a required text field, however, Spry applies the appropriate class to the widget container, as follows:

```
<span id="sprytextfield1" class="textfieldRequiredState">
    <input type="text" name="mytextfield" id="mytextfield" />
    <span class="textfieldRequiredMsg">Please enter a description</span>
</span>
```

In the preceding CSS code, the state rule with the contextual selector `.textfieldRequiredState .textfield-RequiredMsg` overrides the default error-message rule responsible for hiding the error-message text. Thus, when Spry applies the state class to the widget container, the state rule determines the appearance of the widget, and displays the error message inline in red with a 1-pixel solid border.

Following is a list of default error-message classes and their descriptions. You can change these classes and rename them. If you do so, don't forget to change them in the contextual selector also.

Error message class	Description
.textfieldRequiredMsg	Causes error message to display when the widget enters the required state
.textfieldInvalidFormatMsg	Causes error message to display when the widget enters the invalid state
.textfieldMinValueMsg	Causes error message to display when the widget enters the minimum value state
.textfieldMaxValueMsg	Causes error message to display when the widget enters the maximum value state
.textfieldMinCharsMsg	Causes error message to display when the widget enters the minimum number of characters state
.textfieldMaxCharsMsg	Causes error message to display when the widget enters the maximum number of characters state
.textfieldValidMsg	Causes error message to display when the widget enters the valid state

***Note:** You cannot rename state-related class names because they are hard-coded as part of the Spry framework.*

Specify a validation type and format

You can specify many different validation types, formats, and other options for the Validation Text Field widget. For example, you can specify a credit card validation type if the text field will receive credit card numbers.

You specify validation types, formats, and other options as parameters in the widget constructor. The first parameter in the constructor is the widget container ID, followed by a second parameter (the validation-type parameter), and optionally, a third parameter. The third parameter is a JavaScript array of options that depends on the validation type set in the second parameter. Thus, you cannot set the third parameter without setting the second.

***Note:** In cases where you want to set the third parameter, but no validation type is required, you can set the validation type to* `"none"`.

The following code shows the generic syntax for a Text Field widget constructor with various parameters:

```
<script type="text/javascript">
    var sprytextfield1= new Spry.Widget.ValidationTextField("WidgetContainerID",
"ValidationType", {option1:"value1", option2:"value2", ..});
</script>
```

Validation type options

Most validation types cause the text field to expect a standard format. For example, if you apply the integer validation type to a text field, the widget won't validate unless the user enters numbers in the text field. Some validation types, however, let you choose the kind of format your text field will accept.

The following table lists the available validation types and their formats.

Validation type	Format
none	No particular format required.
integer	Text field accepts numbers only.
email	Text field accepts e-mail addresses that contain the @ sign and a period (.) that is both preceded and followed by at least one letter.
date	Formats vary.
time	Formats vary.
credit_card	Formats vary.
zip_code	Formats vary.
phone_number	Text field accepts phone numbers formatted for U.S. and Canada (000) 000-0000 as well as custom formats.
social_security_number	Text field accepts social security numbers formatted as 000-00-0000 by default.
currency	Text field accepts currency formatted as 1,000,000.00 or 1.000.000,00.
real	Validates various kinds of numbers and scientific notation: integers (for example, 1); float values (for example, 12.123); and float values in scientific notation (for example, 1.212e+12, 1.221e-12 where e is used as a power of 10).
ip	Validates IPv4, IPv6, or both kinds of IP addresses.
url	Text field accepts URLs formatted as http://xxx.xxx.xxx, https://xxx.xxx.xxx, or ftp://xxx.xxx.xxx.
custom	Lets you specify a custom validation type and format.

Validate an integer

❖ To set the text field to accept only integer values, add `"integer"` as the value for the second parameter in the constructor, as follows:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"integer");
</script>
```

The integer validation type accepts both negative and positive values. To accept only positive values, add the `allowNegative` option to the third parameter, and set the value to `false`.

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"integer", {allowNegative:false});
</script>
```

The following table shows other common options and values for the third parameter.

Option	Value
format	Not applicable
validateOn	["blur"]; ["change"]; or both together (["blur", "change"])
isRequired	true (the default); false
useCharacterMasking	false (the default); true
minChars/maxChars	Not applicable
minValue/maxValue	Integer value
pattern	Not applicable

Validate an email

❖ To set the text field to accept only standard e-mail formats, add `"email"` as the value for the second parameter in the constructor, as follows:

```
<script type="text/javascript">
    var sprytextfield1= new Spry.Widget.ValidationTextField("sprytextfield1", "email");
</script>
```

You can also set options for the third parameter by using the following syntax:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1", "email",
{option:value});
</script>
```

The following table shows some common options and values for the third parameter.

Option	Value
format	Not applicable
validateOn	`["blur"]`;`["change"]`; or both together (`["blur", "change"]`)
isRequired	`true` (the default); `false`
useCharacterMasking	Not applicable
minChars/maxChars	Positive integer value
minValue/maxValue	Not applicable
pattern	Not applicable

Validate a date

❖ To set the text field to accept a date format, add `"date"` as the value for the second parameter in the constructor, as follows:

```
<script type="text/javascript">
    var sprytextfield1= new Spry.Widget.ValidationTextField("sprytextfield1", "date");
</script>
```

The default date format is "mm/dd/yy" (U.S. date format). You can, however, set a number of other date formats in the third parameter by using the `format` option, as in the following example:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1", "date",
{format:"yyyy-mm-dd"});
</script>
```

The following table shows a full list of formatting options, as well as some other common options and values, for the third parameter.

Option	Value
format	"mm/dd/yy" (default); "mm/dd/yyyy"; "dd/mm/yyyy"; "dd/mm/yy"; "yy/mm/dd"; "yyyy/mm/dd"; "mm-dd-yy"; "dd-mm-yy"; "yyyy-mm-dd"; "mm.dd.yyyy"; "dd.mm.yyyy";
validateOn	["blur"]; ["change"]; or both together (["blur", "change"])
isRequired	true (the default); false
useCharacterMasking	false (the default); true
minChars/maxChars	Not applicable
minValue/maxValue	Date value in the specified format
pattern	Not applicable

Validate a time

❖ To set the text field to accept a time format, add `"time"` as the value for the second parameter in the constructor, as follows:

```
<script type="text/javascript">
    var sprytextfield1= new Spry.Widget.ValidationTextField("sprytextfield1", "time");
</script>
```

The default date format is "HH:mm" (hours and minutes). You can, however, set a number of other time formats in the third parameter by using the `format` option, as in the following example:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1", "time",
{format:"HH:mm:ss"});
</script>
```

The following table shows a full list of formatting options, as well as some other common options and values, for the third parameter.

Option	Value
format	"HH:mm" (the default); "HH:mm:ss"; "hh:mm tt"; "hh:mm:ss tt"; "hh:mm t"; "hh:mm;ss t"; (where "tt" stands for am/pm format, and "t" stands for a/p format.)
validateOn	["blur"]; ["change"]; or both together (["blur", "change"])
isRequired	true (the default); false
useCharacterMasking	false (the default); true

Option	Value
minChars/maxChars	Not applicable
minValue/maxValue	Time value in the specified format
pattern	Not applicable

Validate a credit card

❖ To set the text field to accept a credit card format, add `"credit_card"` as the value for the second parameter in the constructor, as follows:

```
<script type="text/javascript">
    var sprytextfield1= new Spry.Widget.ValidationTextField("sprytextfield1",
"credit_card");
</script>
```

The default is to validate all types of credit cards, but you can specify particular credit card formats to accept in the third parameter by using the `format` option, as in the following example:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"credit_card", {format:"visa"});
</script>
```

The following table shows a full list of formatting options, as well as some other common options and values, for the third parameter.

Option	Value
format	No format (the default); "visa"; "mastercard"; "amex"; "discover"; "dinersclub"
validateOn	["blur"]; ["change"]; or both together (["blur", "change"])
isRequired	true (the default); false
useCharacterMasking	false (the default); true
minChars/maxChars	Not applicable
minValue/maxValue	Not applicable
pattern	Not applicable

Validate a zip code

❖ To set the text field to accept a zip code format, add `"zip_code"` as the value for the second parameter in the constructor, as follows:

```
<script type="text/javascript">
    var sprytextfield1= new Spry.Widget.ValidationTextField("sprytextfield1", "zip_code");
</script>
```

The default format is the U.S. 5-digit zip code format, but you can specify a number of other formats in the third parameter by using the `format` option, as in the following example:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"zip_code", {format:"zip_uk"});
</script>
```

The following table shows a full list of formatting options, as well as some other common options and values, for the third parameter.

Option	Value
format	"zip_us5" (the default); "zip_us9"; "zip_uk"; "zip_canada"; "zip_custom"
validateOn	["blur"]; ["change"]; or both together (["blur", "change"])
isRequired	true (the default); false
useCharacterMasking	false (the default); true
minChars/maxChars	Not applicable
minValue/maxValue	Not applicable
pattern	Custom zip code pattern. Used when format="zip_custom"

The `"zip_custom"` format lets you specify your own customized pattern for zip-code format. After specifying `"zip_custom"` as your format, add the pattern option to the third parameter to define the format. For example, you might want to validate a zip code that has three numbers, followed by another set of numbers and case-sensitive alphabetic characters. After the `format` option, add the `pattern` option in the third parameter of the constructor to specify the custom pattern, as follows:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"zip_code", {format:"zip_custom", pattern:"000 00AA"});
</script>
```

The following table shows a full list of values used for custom patterns.

Value	Description
"0"	Whole numbers between 0 and 9
"A"	Uppercase alphabetic characters
"a"	Lowercase alphabetic characters
"B"; "b"	Case-insensitive alphabetic characters
"X"	Uppercase alphanumeric characters
"x"	Lowercase alphanumeric characters
"Y"; "y"	Case-insensitive alphanumeric characters
"?"	Any character

The "A", "a", "X", and "x" custom pattern characters are case sensitive. In situations that use these characters, Spry converts the characters to the correct case, even if the user enters the wrong case.

Validate a phone number

❖ To set the text field to accept a phone number format, add `"phone_number"` as the value for the second parameter in the constructor, as follows:

```
<script type="text/javascript">
    var sprytextfield1= new Spry.Widget.ValidationTextField("sprytextfield1",
"phone_number");
</script>
```

The default format is U.S. area code and phone number format: (000) 000-0000, but you can specify a custom format in the third parameter by using the `"phone_custom"` and `"pattern"` options, as in the following example:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"zip_code", {format:"phone_custom" , pattern:"00 0000 A"});
</script>
```

The following table shows a full list of values used for custom patterns.

Value	Description
"0"	Whole numbers between 0 and 9
"A"	Uppercase alphabetic characters
"a"	Lowercase alphabetic characters
"B"; "b"	Case-insensitive alphabetic characters
"X"	Uppercase alphanumeric characters
"x"	Lowercase alphanumeric characters
"Y"; "y"	Case-insensitive alphanumeric characters
"?"	Any character

The "A", "a", "X", and "x" custom pattern characters are case sensitive. In situations that use these characters, Spry converts the characters to the correct case, even if the user enters the wrong case.

The following table shows some other common options and values for the third parameter.

Option	Value
format	"phone_number" (the default); "phone_custom"
validateOn	["blur"]; ["change"]; or both together (["blur", "change"])
isRequired	true (the default); false
useCharacterMasking	false (the default); true
minChars/maxChars	Not applicable
minValue/maxValue	Not applicable
pattern	Custom phone pattern. Used when format="phone_custom"

Validate a social security number

❖ To set the text field to accept a social security number format, add `"social_security_number"` as the value for the second parameter in the constructor, as follows:

```
<script type="text/javascript">
    var sprytextfield1= new Spry.Widget.ValidationTextField("sprytextfield1",
"social_security number");
</script>
```

The default format is U.S. social security number format with dashes: 000-00-0000, but you can specify a custom format in the third parameter by using the pattern option, as in the following example:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"social_security_number", {format:"custom" pattern:"00 0000 A"});
</script>
```

The following table shows a full list of values used for custom patterns.

Value	Description
"0"	Whole numbers between 0 and 9
"A"	Uppercase alphabetic characters
"a"	Lowercase alphabetic characters
"B"; "b"	Case-insensitive alphabetic characters
"X"	Uppercase alphanumeric characters
"x"	Lowercase alphanumeric characters
"Y"; "y"	Case-insensitive alphanumeric characters
"?"	Any character

The "A", "a", "X", and "x" custom pattern characters are case sensitive. In situations that use these characters, Spry converts the characters to the correct case, even if the user enters the wrong case.

The following table shows some other common options for the third parameter.

Option	Value
format	Not applicable
validateOn	["blur"]; ["change"]; or both together (["blur", "change"])
isRequired	true (the default); false
useCharacterMasking	false (the default); true
minChars/maxChars	Not applicable
minValue/maxValue	Not applicable
pattern	Custom social security pattern

Validate currency

❖ To set the text field to accept a currency number format, add `"currency"` as the value for the second parameter in the constructor, as follows:

```
<script type="text/javascript">
    var sprytextfield1= new Spry.Widget.ValidationTextField("sprytextfield1", "currency");
</script>
```

The default format is U.S. currency format: 1,000.00, but you can also specify the `"dot_comma"` format (1.000,00) in the third parameter, as follows:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"currency", {format:"dot_comma"});
</script>
```

In both cases, the separators for the 3-digit groups can be optional and the decimal part (i.e. the .00 in 1,000.00) is not required.

The following table shows some other common options for the third parameter.

Option	Value
format	"comma_dot" (the default); "dot_comma"
validateOn	["blur"]; ["change"]; or both together (["blur", "change"])
isRequired	true (the default); false
useCharacterMasking	false (the default); true
minChars/maxChars	Not applicable
minValue/maxValue	Numeric value with two decimal numbers allowed
pattern	Not applicable

Validate real numbers and scientific notation

❖ To set the text field to accept real numbers and scientific notation, add `"real"` as the value for the second parameter in the constructor, as follows:

```
<script type="text/javascript">
    var sprytextfield1= new Spry.Widget.ValidationTextField("sprytextfield1", "real");
</script>
```

The text field validates a real number in scientific notation, for example 1.231e10.

The following table shows some other common options for the third parameter.

Option	Value
format	Not applicable
validateOn	["blur"]; ["change"]; or both together (["blur", "change"])
isRequired	true (the default); false
useCharacterMasking	false (the default); true
minChars/maxChars	Not applicable
minValue/maxValue	Numeric value with decimal numbers
pattern	Not applicable

Validate an IP address

❖ To set the text field to validate an IP address, add `"ip"` as the value for the second parameter in the constructor, as follows:

```
<script type="text/javascript">
    var sprytextfield1= new Spry.Widget.ValidationTextField("sprytextfield1", "ip");
</script>
```

The default accepted IP address format is IPv4, but you can set other IP address formats in the third parameter by using the format option, as in the following example:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1", "ip",
{format:"ipv6"});
</script>
```

The following table shows the formatting options, as well as some other common options, for the third parameter.

Option	Value
format	"ipv4" (the default); "ipv6"; "ipv4_ipv6"
validateOn	["blur"]; ["change"]; or both together (["blur", "change"])
isRequired	true (the default); false
useCharacterMasking	false (the default); true
minChars/maxChars	Not applicable
minValue/maxValue	Not applicable
pattern	Not applicable

Validate a URL

❖ To set the text field to accept only URL values, add `"url"` as the value for the second parameter in the constructor, as follows:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1", "url");
</script>
```

The URL validation type accepts HTTP, HTTPS, and FTP URL values.

The following table shows other common options for the third parameter.

Option	Value
format	Not applicable
validateOn	["blur"]; ["change"]; or both together (["blur", "change"])
isRequired	true (the default); false
useCharacterMasking	Not applicable
minChars/maxChars	Positive integer value
minValue/maxValue	Not applicable
pattern	Not applicable

Validate a custom format

❖ To set the text field to accept any kind of customized format, specify `"custom"` as the value for the second parameter, and adding the pattern option in the third parameter, as follows:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"custom", {pattern:"00 0000 AX"});
</script>
```

The following table shows a full list of values used for custom patterns.

Value	Description
"0"	Whole numbers between 0 and 9
"A"	Uppercase alphabetic characters
"a"	Lowercase alphabetic characters
"B"; "b"	Case-insensitive alphabetic characters
"X"	Uppercase alpha-numeric characters
"x"	Lowercase alpha-numeric characters
"Y"; "y"	Case-insensitive alpha-numeric characters
"?"	Any character

The "A", "a", "X", and "x" custom pattern characters are case sensitive. In situations that use these characters, Spry converts the characters to the correct case, even if the user enters the wrong case.

Specify when validation occurs

By default, the Validation Text Field widget validates when the user clicks the submit button. You can, however, set two other options: `blur` or `change`. The `validateOn:["blur"]` parameter causes the widget to validate whenever the user clicks outside the text field. The `validateOn:["change"]` parameter causes the widget to validate as the user changes text inside the text field.

❖ To specify when validation occurs, add a `validateOn` parameter to the constructor as follows:

```
<script type="text/javascript">
    var sprytextfield1 = new Spry.Widget.ValidationTextField("sprytextfield1", "none",
{validateOn:["blur"]});
</script>
```

As a convenience, you can discard the brackets if your `validateOn` parameter contains a single value (for example, `validateOn: "blur"`). If the parameter contains both values, however (`validateOn:["blur", "change"]`), include brackets in the syntax.

***Note:** In the preceding example, the second parameter is set to `"none"`, but it could easily be set to any of the available validation types (for example, `"phone"` or `"credit_card"`). See “Specify a validation type and format” on page 50.*

Specify a minimum and maximum number of characters

This option is only available for the `"none"`, `"integer"`, `"email"`, and `"url"` validation types.

The `minChars` option does not enforce a minimum number of characters if the text field is not required.

❖ To specify a minimum or maximum number of characters, add the `minChars` property or `maxChars` property (or both) and a value to the constructor, as follows:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"validation_type", {minChars:value, maxChars:value});
</script>
```

***Note:** If a validation type is not required, you can set the validation type to `"none"`. See “Specify a validation type and format” on page 50.*

Specify minimum and maximum values

This option is only available for the `"integer"`, `"date"`, `"time"`, `"currency"`, and `"real"` validation types.

❖ To specify minimum or maximum values in the text field, add the `minValue` property or `maxValue` property (or both) and a value to the constructor, as follows:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"validation_type", {minValue:value, maxValue:value});
</script>
```

Change required status of a text field

By default, Validation Text Field widgets require user input when published on a web page. You can, however, make the completion of text fields optional for the user.

❖ To change the required status of a text field, add the `isRequired` property to the constructor and set its value to false, as follows:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1", "none",
{isRequired:false});
</script>
```

***Note:** In the preceding example, the second parameter is set to `"none"`, but it could easily be set to any of the available validation types (for example, `"phone"` or `"credit_card"`). See “Specify a validation type and format” on page 50.*

Create a hint for a text field

Because there are so many different kinds of formats for text fields, it is helpful to give your users a hint as to what format they need to enter. For example, a text field set with the Phone Number validation type accepts phone numbers in the form (000) 000-0000 by default. You can enter these sample numbers as a hint so that the text field displays the correct format when the user loads the page in a browser.

This option doesn't depend on the validation type, so you can always specify it. (Specify `"none"` as your validation type, if no other validation type is required.) When the user clicks inside the text field, the hint disappears; when the user clicks outside the text field, Spry restores the hint to the textfield if the field contains no value.

❖ To create a hint for a text field, add the `hint` property to the third parameter of the constructor and the text of your hint as the value, as follows:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"validation_type", {hint:"some hint text here"});
</script>
```

Block invalid characters

You can prevent users from entering invalid characters in a Validation Text Field widget. For example, if you set this option for a widget set with the Integer validation type, nothing appears in the text field if the user tries to type a letter.

You must specify a validation type for this option because the specification of the third parameter depends on the second parameter. If no validation type is required, specify `"none"` as your validation type.

This option does not work in older browsers.

❖ To block invalid characters, add the `useCharacterMasking` property to the constructor and set its value to true, as follows:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"validation_type", {useCharacterMasking:true});
</script>
```

Customize the Validation Text Field widget

The SpryValidationTextField.css file provides the default styling for the Validation Text Field widget. You can, however, customize the widget by changing the appropriate CSS rule. The CSS rules in the SpryValidationText-Field.css file use the same class names as the related elements in the widget's HTML code, so it's easy for you to know which CSS rules correspond to the widget and its error states.

The SpryValidationTextField.css file should already be included in your website before you start customizing. For more information, see "Prepare your files" on page 3.

Style a Validation Text Field widget (general instructions)

1 Open the SpryValidationTextField.css file.

2 Locate the CSS rule for the part of the widget to change. For example, to change the background color of the Text Field widget's required state, edit the `.textfieldRequiredState` rule in the SpryValidationTextField.css file.

3 Make your changes to the CSS and save the file.

The SpryValidationTextField.css file contains extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Style Validation Text Field widget error message text

By default, error messages for the Validation Text Field widget appear in red with a 1-pixel solid border surrounding the text.

❖ To change the text styling of Validation Text Field widget error messages, use the following table to locate the appropriate CSS rule, and then change the default properties, or add your own text-styling properties and values.

Text to change	Relevant CSS rule	Relevant properties to change
Error message text	.textfieldRequiredState .textfieldRequiredMsg, .textfieldInvalidFormatState .textfieldInvalidFormatMsg, .textfieldMinValueState .textfieldMinValueMsg, .textfieldMaxValueState .textfieldMaxValueMsg, .textfieldMinCharsState .textfieldMinCharsMsg, .textfieldMaxCharsState .textfieldMaxCharsMsg	color: #CC3333; border: 1px solid #CC3333;

Change Validation Text Field widget background colors

❖ To change the background colors of the Validation Text Field widget in various states, use the following table to locate the appropriate CSS rule, and then change the default background color values.

Color to change	Relevant CSS rule	Relevant property to change
Background color of widget in valid state	.textfieldValidState input, input.textfieldValidState	background-color: #B8F5B1;
Background color of widget in invalid state	input.textfieldRequiredState, .textfieldRequiredState input, input.textfieldInvalidFormatState, .textfieldInvalidFormatState input, input.textfieldMinValueState, .textfieldMinValueState input, input.textfieldMaxValueState, .textfieldMaxValueState input, input.textfieldMinCharsState, .textfieldMinCharsState input, input.textfieldMaxCharsState, .textfieldMaxCharsState input	background-color: #FF9F9F;
Background color of widget in focus	.textfieldFocusState input, input.textfieldFocusState	background-color: #FFFFCC;

Specify custom patterns

The following table shows a full list of values used for custom patterns.

Value	Description
"0"	Whole numbers between 0 and 9
"A"	Uppercase alphabetic characters
"a"	Lowercase alphabetic characters
"B"; "b"	Case-insensitive alphabetic characters
"X"	Uppercase alpha-numeric characters
"x"	Lowercase alpha-numeric characters
"Y"; "y"	Case-insensitive alpha-numeric characters
"?"	Any character

Use these values to create a custom pattern for any of the format types. For example, to specify a custom social security number that's a combination of numbers and capital letters, specify the following custom pattern in the constructor's third parameter:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"social_security_number", {format:"custom", pattern:"AA00AA"});
</script>
```

The preceding code results in a text field that accepts values such as UT99CW, AB87PP, GV44RE, and so on.

Note: *Dreamweaver users need only enter a custom pattern in the Pattern text box of the Property inspector, using the values provided in the preceding table. For example, AA00AA.*

Note: *Spry's validation mechanism supports only ASCII characters.*

Insert autocomplete characters

The following table shows a full list of values used for custom patterns.

Value	Description
"0"	Whole numbers between 0 and 9
"A"	Uppercase alphabetic characters
"a"	Lowercase alphabetic characters
"B"; "b"	Case-insensitive alphabetic characters
"X"	Uppercase alpha-numeric characters
"x"	Lowercase alpha-numeric characters
"Y"; "y"	Case-insensitive alpha-numeric characters
"?"	Any character

Any characters (letters, punctuation, and so on) other than the backslash (\) and those listed in the preceding table are considered autocomplete characters, and Spry can insert these kinds of characters at the appropriate time. For example, if you have a zip code like UT.99CW, you might want to have Spry insert the period automatically after the user types the first two capital letters.

❖ To use an autocomplete character, insert the character in the format pattern, as follows:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"zip_code", {format:"zip_custom", pattern:"AA.00AA"});
</script>
```

The preceding code results in a text field that accepts values such as UT.99CW, AB.87PP, GV.44RE, and so on, with the period appearing as an autocomplete character after the user types the first two capital letters.

You can also have Spry autocomplete letters (other than those in the preceding table), as in the following example:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"zip_code", {format:"zip_custom", pattern:"AA.00AAF3"});
</script>
```

The preceding code results in a text field that accepts values such as UT.99CWF3, AB.87PPF3, GV.44REF3, and so on, with the period appearing as an autocomplete character after the user types the first two capital letters, and the F3 appearing after the user types the last two capital letters.

To use any of the special characters listed in the preceding table as an autocomplete character, escape them with a double backslash (\\), as in the following example:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"zip_code", {format:"zip_custom", pattern:"AA.00AA\\B3"});
</script>
```

The preceding code results in a text field that accepts values such as UT.99CWB3, AB.87PPB3, GV.44REB3, and so on, with the period appearing as an autocomplete character after the user types the first two capital letters, and the B3 appearing after the user types the last two capital letters.

To use a backslash (\) as part of an autocomplete sequence, double it, and escape it using a double backslash—a total of four backslashes (\\\\)—as follows:

```
<script type="text/javascript">
    var textfieldwidget1 = new Spry.Widget.ValidationTextField("textfieldwidget1",
"zip_code", {format:"zip_custom", pattern:"AA\\\\00AA"});
</script>
```

The preceding code results in a text field that accepts values such as UT\99CW, AB\87PP, GV\44RE, and so on, with the backslash appearing as an autocomplete character after the user types the first two capital letters.

Customize state-related class names

While you can replace error-message-related class names with class names of your own by changing the rules in the CSS code and the class names in the HTML code, you cannot change or replace state-related class names, because the behaviors are hard-coded as part of the Spry framework. You can, however, override the default state-related class name with your own class name by specifying a new value in the third parameter of the widget constructor.

❖ To change widget state-related class names, add one of the overriding options to the third parameter of the widget constructor, and specify your custom class name, as follows:

```
<script type="text/javascript">
    var sprytextfield1 = new Spry.Widget.ValidationTextField("sprytextfield1", "none",
{requiredClass:"required"});
</script>
```

The following table provides a list of options you can use to override built-in state-related class names.

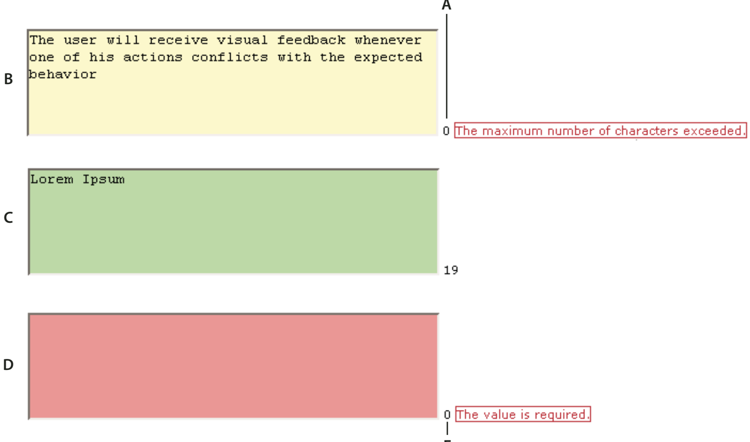
Option	Description
requiredClass	Overrides the "textfieldRequiredState" built-in value
invalidFormatClass	Overrides the "textfieldInvalidFormatState" built-in value
validClass	Overrides the "textfieldValidState" built-in value
focusClass	Overrides the "textfieldFocusState" built-in value
invalidFormatClass	Overrides the "textfieldInvalidFormatState" built-in value
invalidRangeMinClass	Overrides the "textfieldMinValueState" built-in value
invalidRangeMaxClass	Overrides the "textfieldMaxValueState" built-in value
invalidCharsMinClass	Overrides the "textfieldMinCharsState" built-in value
invalidCharsMaxClass	Overrides the "textfieldMaxCharsState" built-in value
textfieldFlashTextClass	Overrides the "textfieldFlashText" built-in value

Working with the Validation Text Area widget

Validation Text Area widget overview and structure

A Spry Validation Text Area widget is a text area that displays valid or invalid states when the user enters a few sentences of text. If the text area is a required field and the user fails to enter any text, the widget returns a message stating that a value is required.

The following example shows a Validation Text Area widget in various states.



***A.** Characters remaining counter* ***B.** Text Area widget in focus, maximum number of characters state* ***C.** Text Area widget in focus, valid state* ***D.** Text Area widget, required state* ***E.** Characters typed counter*

The Validation Text Area widget includes a number of states (for example, valid, invalid, required value, and so on). You can alter the properties of these states using the Property inspector, depending on the desired validation results. A Validation Text Area widget can validate at various points—for example, when the user clicks outside the widget, as the user types, or when the user tries to submit the form.

Initial state When the page loads in the browser, or when the user resets the form.

Focus state When the user places the insertion point in the widget.

Valid state When the user enters information correctly, and the form can be submitted.

Required state When the user fails to enter any text.

Minimum Number Of Characters state When the user enters fewer than the minimum number of characters required in the text area.

Maximum Number Of Characters state When the user enters greater than the maximum number of characters allowed in the text area.

Whenever a Validation Text Area widget enters one of these states through user interaction, the Spry framework logic applies a specific CSS class to the HTML container for the widget at run time. For example, if a user tries to submit a form, but did not enter text in the text area, Spry applies a class to the widget that causes it to display the error message, "A value is required." The rules that control the style and display states of error messages exist in the SpryValidationTextarea.css file that accompanies the widget.

The default HTML code for the Validation Text Area widget, usually inside a form, is made up of a container `span` tag that surrounds the `textarea` tag of the text area. The HTML code for the Validation Text Area widget also includes `script` tags in the head of the document and after the widget's HTML code.

The HTML code for the Validation Text Area widget also includes `script` tags in the head of the document and after the widget's HTML code. The `script` tag in the head of the document defines all of the JavaScript functions related to the Text Area widget. The `script` tag after the widget code creates a JavaScript object that makes the text area interactive.

Following is the HTML code for a Validation Text Area widget:

```
<head>
...
<!-- Link the Spry Validation Text Area JavaScript library -->
<script src="SpryAssets/SpryValidationTextarea.js" type="text/javascript"></script>
<!-- Link the CSS style sheet that styles the widget -->
<link href="SpryAssets/SpryValidationTextarea.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" name="form1" method="post" action="">
        <!-- Create the text area widget and assign a unique id-->
        <span id="sprytextarea1">
            <textarea name="textarea1" id="textarea1" cols="45" rows="5"></textarea>
            <!--Display an error message-->
            <span class="textareaRequiredMsg">A value is required.</span>
        </span>
    </form>
<!-- Initialize the Validation Text Area widget object-->
<script type="text/javascript">
var sprytextarea1 = new Spry.Widget.ValidationTextarea("sprytextarea1");
</script>
</body>
```

In the code, the `new` JavaScript operator initializes the Text Area widget object, and transforms the `span` content with the ID of `sprytextarea1` from static HTML code into an interactive page element.

The `span` tag for the error message in the widget has a CSS class applied to it. This class (which is set to `display:none;` by default), controls the style and visibility of the error message, and exists in the accompanying SpryValidationTextarea.css file. When the widget enters different states as a result of user interaction, Spry places different classes on the container for the widget, which in turn affects the error-message class.

You can add other error messages to a Validation Text Area widget by creating a `span` tag (or any other type of tag) to hold the text of the error message. Then, by applying a CSS class to it, you can hide or show the message, depending on the widget state.

You can change the default appearance of the Validation Text Area widget's states by editing the corresponding CSS rule in the SpryValidationTextarea.css file. For example, to change the background color for a state, edit the corresponding rule or add a new rule (if it’s not already present) in the style sheet.

Variation on tags used for Text Area widget structure

In the preceding example, `span` tags create the structure for the widget:

```
Container SPAN
    TEXTAREA tag
    Error message SPAN
```

You can, however, use almost any container tag to create the widget:

```
Container DIV
    TEXTAREA tag
    Error Message P
```

Spry uses the tag ID (not the tag itself) to create the widget. Spry also displays error messages using CSS code that is indifferent to the actual tag used to contain the error message.

The ID passed into the widget constructor identifies a specific HTML element. The constructor finds this element and looks inside the identified container for a corresponding `textarea` tag. If the ID passed to the constructor is the ID of the `textarea` tag (rather than a container tag), the constructor attaches validation triggers directly to the `textarea` tag. If there is no container tag, however, the widget cannot display error messages, and different validation states will only alter the appearance of the `textarea` tag element (for example, its background color).

***Note:** Multiple `textarea` tags do not work inside the same HTML widget container. Each text field should be its own widget.*

CSS code for the Validation Text Area widget

The SpryValidationTextarea.css file contains the rules that style the Validation Text Area widget and its error messages. You can edit these rules to style the look and feel of the widget and error messages. The names of the rules in the CSS file correspond to the names of the classes specified in the widget's HTML code.

The following is the CSS code for the SpryValidationTextarea.css file:

```
/*Validation Textarea styling classes*/
.textareaRequiredMsg, .textareaMinCharsMsg, .textareaMaxCharsMsg, .textareaValidMsg {
    display:none;
}
.textareaRequiredState .textareaRequiredMsg,
.textareaMinCharsState .textareaMinCharsMsg,
.textareaMaxCharsState .textareaMaxCharsMsg{
    display: inline;
    color: #CC3333;
    border: 1px solid #CC3333;
}
.textareaValidState textarea, textarea.textareaValidState {
    background-color:#B8F5B1;
}
textarea.textareaRequiredState, .textareaRequiredState textarea,
textarea.textareaMinCharsState, .textareaMinCharsState textarea,
textarea.textareaMaxCharsState, .textareaMaxCharsState textarea {
    background-color:#FF9F9F;
}
.textareaFocusState textarea, textarea.textareaFocusState {
    background-color:#FFFFCC;
}
.textareaFlashState textarea, textarea.textareaFlashState{
    color:red !important;
}
```

The SpryValidationTextField.css file also contains extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Insert the Validation Text Area widget

1 Locate the SpryValidationTextarea.js file and add it to your web site. You can find the SpryValidationTextarea.js file in the widgets/textareavalidation directory, located in the Spry directory that you downloaded from Adobe Labs. For more information, see "Prepare your files" on page 3.

For example, create a folder called *SpryAssets* in the root folder of your web site, and upload the SpryValidationTextarea.js file to it. The SpryValidationTextarea.js file contains all of the information necessary for making the Text Area widget interactive.

2 Locate the SpryValidationTextarea.css file and add it to your web site. You might choose to add it to the main directory, a SpryAssets directory, or to a CSS directory if you have one.

3 Open the web page to add the Text Area widget to and link the SpryValidationTextarea.js file to the page by inserting the following `script` tag in the page's head tag:

```
<script src="SpryAssets/SpryValidationTextarea.js" type="text/javascript"></script>
```

Make sure that the file path to the SpryValidationTextarea.js file is correct. This path varies depending on where you've placed the file in your web site.

4 Link the SpryValidationTextarea.css file to your web page by inserting the following `link` tag in the page's head tag:

```
<link href="SpryAssets/SpryValidationTextarea.css" rel="stylesheet" type="text/css" />
```

Make sure that the file path to the SpryValidationTextarea.css file is correct. This path varies depending on where you've placed the file in your web site.

5 Add a text area to the page and give it a name and a unique ID:

```
<textarea name="mytextarea" id="textarea"></textarea>
```

6 Add a container around the text area by inserting `span` tags around the `<textarea>` tags, and assigning the widget a unique ID, as follows:

```
<span id="sprytextarea1">
    <textarea name="mytextarea"></textarea>
</span>
```

7 Initialize the text area object by inserting the following script block after the HTML code for the widget:

```
<script type="text/javascript">
    var sprytextarea1 = new Spry.Widget.ValidationTextarea("sprytextarea1");
</script>
```

The `new` JavaScript operator initializes the Text Area widget object, and transforms the `span` tag content with the ID of sprytextarea1 from static HTML code into an interactive text field object. The Spry.Widget.ValidationTextarea method is a constructor in the Spry framework that creates text area objects. The information necessary to initialize the object is contained in the JavaScript library, SpryValidationTextarea.js, that you linked to at the beginning of this procedure.

Make sure that the ID of the text area's container `span` tag matches the ID parameter you specified in the Spry.Widgets.ValidationTextarea method. Make sure that the JavaScript call comes after the HTML code for the widget.

8 Save the page.

The complete code looks as follows:

```
<head>
...
<script src="SpryAssets/SpryValidationTextarea.js" type="text/javascript"></script>
<link href="SpryAssets/SpryValidationTextarea.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" name="form1" method="post" action="">
        <span id="sprytextarea1">
            <textarea name="textarea1" id="textarea1" cols="45" rows="5"></textarea>
        </span>
    </form>
<script type="text/javascript">
var sprytextarea1 = new Spry.Widget.ValidationTextarea("sprytextarea1");
</script>
</body>
```

Display error messages

❖ Create a span tag (or any other type of tag) to display the error message, and assign the appropriate class to it, as follows:

```
<span id="sprytextarea1">
    <textarea name="textarea1" id="textarea1" cols="45" rows="5"></textarea>
    <span class="textareaRequiredMsg">Please enter a description</span>
</span>
```

The `.textareaRequiredMsg` rule is located in the SpryValidationTextarea.css file, and is set to `display:none` by default. When the widget enters a different state through user interaction, Spry applies the appropriate class —the state class—to the container of the widget. This action affects the error message class, and by extension, the appearance of the error message.

For example, the following shows a portion of the CSS from the SpryValidationTextarea.css file:

```
.textareaRequiredMsg,
.textareaMinCharsMsg,
.textareaMaxCharsMsg,
.textareaValidMsg {
    display:none;
}
.textareaRequiredState .textareaRequiredMsg,
.textareaMinCharsState .textareaMinCharsMsg,
.textareaMaxCharsState .textareaMaxCharsMsg {
    display: inline;
    color: #CC3333;
    border: 1px solid #CC3333;
}
```

By default, there is no state class applied to the widget container, so that when the page loads in a browser, the error message text in the preceding HTML code example only has the `.textareaRequiredMsg` class applied to it. (The property and value pair for this rule is `display:none`, so the message remains hidden.) If the user fails to enter text in a required text area, however, Spry applies the appropriate class to the widget container, as follows:

```
<span id="sprytextarea1" class="textareaRequiredState">
    <input type="text" name="mytextarea" id="mytextarea" />
    <span class="textareaRequiredMsg">Please enter a description</span>
</span>
```

In the CSS in the preceding code, you can see that the state rule with the contextual selector `.textareaRequired-State . textareaRequiredMsg` overrides the default error-message rule responsible for hiding the error message text. Thus, when Spry applies the state class to the widget container, the state rule determines the appearance of the widget, and displays the error message inline in red with a 1-pixel solid border.

Following is a list of default error-message classes and their descriptions. You can change these classes and rename them to anything you want. If you do so, don't forget to change them in the contextual selector also.

Error message class	Description
.textareaRequiredMsg	Causes error message to display when the widget enters the required state
.textareaMinCharsMsg	Causes error message to display when the widget enters the minimum number of characters state
.textareaMaxCharsMsg	Causes error message to display when the widget enters the maximum number of characters state
.textareaValidMsg	Causes error message to display when the widget enters the valid state

***Note:** You cannot rename state-related class names because they are hard-coded as part of the Spry framework.*

Specify when validation occurs

By default, the Validation Text Area widget validates when the user clicks the submit button. You can, however, set two other options: blur or change. The `validateOn:["blur"]` parameter causes the widget to validate whenever the user clicks outside the text area. The `validateOn:["change"]` parameter causes the widget to validate as the user changes text inside the text area.

❖ To specify when validation occurs, add a `validateOn` parameter to the constructor as follows:

```
<script type="text/javascript">
    var sprytextarea1 = new Spry.Widget.ValidationTextarea("sprytextarea1",
{validateOn:["blur"]});
</script>
```

As a convenience, you can discard the brackets if your `validateOn` parameter contains a single value (for example, `validateOn: "blur"`). If the parameter contains both values, however (`validateOn:["blur", "change"]`), include brackets in the syntax.

Specify a minimum and maximum number of characters

❖ To specify a minimum or maximum number of characters, add the `minChars` property or `maxChars` property (or both) and a value to the constructor, as follows:

```
<script type="text/javascript">
    var textareawidget1 = new
Spry.Widget.ValidationTextarea("textareawidget1",{minChars:value, maxChars:value});
</script>
```

Add a character counter

You can add a character counter that lets your users know how many characters they have typed, or how many characters are remaining when entering text in the text area.

1 Add a `span` tag after the `textarea` tag for the widget, and assign a unique ID to the tag, as follows:

```
<form id="form1" name="form1" method="post" action="">
    <span id="sprytextarea1">
        <textarea name="textarea1" id="textarea1" cols="45" rows="5"></textarea>
        <span id="my_counter_span"></span>
        <span class="textareaRequiredMsg">Maximum number of characters exceeded</span>
    </span>
</form>
<script type="text/javascript">
    var sprytextarea1 = new Spry.Widget.ValidationTextarea("sprytextarea1" {maxChars:100});
</script>
```

Leave the new tag empty. Spry provides the content of the tag later as the user types in text.

Note: *The counter tag must appear within the HTML container tag for the widget.*

2 Add the `counterType` option and a value to the widget constructor, as follows:

```
<form id="form1" name="form1" method="post" action="">
    <span id="sprytextarea1">
        <textarea name="textarea1" id="textarea1" cols="45" rows="5"></textarea>
        <span id="my_counter_span"></span>
        <span class="textareaRequiredMsg">Maximum number of characters exceeded</span>
    </span>
</form>
<script type="text/javascript">
    var sprytextarea1 = new Spry.Widget.ValidationTextarea("sprytextarea1" {maxChars:100,
counterType:"chars_remaining"});
</script>
```

The `counterType` option defines the type of counter to use and can take two values: `"chars_count"`, or `"chars_remaining"`. The `"chars_count"` value results in a counter that counts the number of characters typed in the text area. The "chars_remaining" value results in a counter that displays the number of characters remaining before the maximum number of characters is reached. The second option must be used in conjunction with the `maxChars` option, as in the preceding example. For more information, see "Specify a minimum and maximum number of characters" on page 71.

3 Add the `counterId` option to the widget constructor, and set its value equal to the unique idea you assigned to the counter `span` tag, as follows:

```
<form id="form1" name="form1" method="post" action="">
    <span id="sprytextarea1">
        <textarea name="textarea1" id="textarea1" cols="45" rows="5"></textarea>
        <span id="my_counter_span"></span>
        <span class="textareaRequiredMsg">Maximum number of characters exceeded</span>
    </span>
</form>
<script type="text/javascript">
    var sprytextarea1 = new Spry.Widget.ValidationTextarea("sprytextarea1" {maxChars:100,
counterType:"chars_remaining", counterId:"my_counter_span"});
</script>
```

Change required status of a text area

By default, Validation Text Area widgets require user input when published on a web page. You can, however, make the completion of text areas optional for the user.

❖ To change the required status of a text area, add the `isRequired` property to the constructor and set its value to false, as follows:

```
<script type="text/javascript">
    var textareawidget1 = new Spry.Widget.ValidationTextarea("textareawidget1",
{isRequired:false});
</script>
```

Create a hint for a text area

The `hint` option lets you display a hint that lets your user know what kind of text they should enter (for example, “Enter your address here”). The hint appears in the text area when the user loads the page in a browser and no predefined value exists.

❖ To create a hint for a text area, add the `hint` property to the constructor and the text of your hint as the value, as follows:

```
<script type="text/javascript">
    var textareawidget1 = new Spry.Widget.ValidationTextarea("textareawidget1",
{hint:"Enter your address here"});
</script>
```

Block extra characters

You can prevent your users from entering more than the maximum number of allowed characters in a Validation Text Area widget. For example, if you set the useCharacterMasking option so that a widget can accept no more than 20 characters, the user cannot type more than 20 characters in the text area.

Use this option in conjunction with the `maxChars` option. For information, see “Specify a minimum and maximum number of characters” on page 71.

❖ To block extra characters, add the `useCharacterMasking` property to the constructor and set its value to `true`, as follows:

```
<script type="text/javascript">
    var textareawidget1 = new Spry.Widget.ValidationTextarea("textareawidget1",
maxChars:20, {useCharacterMasking:true});
</script>
```

Customize the Validation Text Area widget

The SpryValidationTextarea.css file provides the default styling for the Validation Text Area widget. You can, however, customize the widget by changing the appropriate CSS rule. The CSS rules in the SpryValidationTextarea.css file use the same class names as the related elements in the widget’s HTML code, so it’s easy for you to know which CSS rules correspond to the widget and its error states.

The SpryValidationTextarea.css file should already be included in your website before you start customizing. For more information, see “Prepare your files” on page 3.

Style a Validation Text Area widget (general instructions)

1 Open the SpryValidationTextarea.css file.

2 Locate the CSS rule for the part of the widget to change. For example, to change the background color of the Text Area widget's required state, edit the `textareaRequiredState` rule in the SpryValidationTextarea.css file.

3 Make your changes to the CSS rule and save the file.

The SpryValidationTextarea.css file contains extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Style Validation Text Area widget error message text

By default, error messages for the Validation Text Area widget appear in red with a 1-pixel solid border surrounding the text.

❖ To change the text styling of Validation Text Area widget error messages, use the following table to locate the appropriate CSS rule, and then change the default properties, or add your own text styling properties and values.

Text to change	Relevant CSS rule	Relevant properties to change
Error message text	.textareaRequiredState .textareaRequiredMsg, .textareaMinCharsState .textareaMinCharsMsg, .textareaMaxCharsState .textareaMaxCharsMsg	color: #CC3333; border: 1px solid #CC3333;

Change Validation Text Area widget background colors

❖ To change the background colors of the Validation Text Area widget in various states, use the following table to locate the appropriate CSS rule, and then change the default background color values.

Background color to change	Relevant CSS rule	Relevant property to change
Background color of widget in valid state	.textareaValidState textarea, textarea.textareaValidState	background-color: #B8F5B1;
Background color of widget in invalid state	textarea.textareaRequiredState, .textareaRequiredState textarea, textarea.textareaMinCharsState, .textareaMinCharsState textarea, textarea.textareaMaxCharsState, .textareaMaxCharsState textarea	background-color: #FF9F9F;
Background color of widget in focus	.textareaFocusState textarea, textarea.textareaFocusState	background-color: #FFFFCC;

Customize state-related class names

While you can replace error message-related class names with class names of your own by changing the rules in the CSS and the class names in the HTML code, you cannot change or replace state-related class names, because the behaviors are hard-coded as part of the Spry framework. You can, however, override the default state-related class name with your own class name by specifying a new value in the third parameter of the widget constructor.

❖ To change widget state-related class names, add one of the overriding options to the third parameter of the widget constructor, and specify your custom class name, as follows:

```
<script type="text/javascript">
    var sprytextarea1 = new Spry.Widget.ValidationTextarea("sprytextarea1",
{requiredClass:"required"});
</script>
```

The following table provides a list of options you can use to override built-in state-related class names.

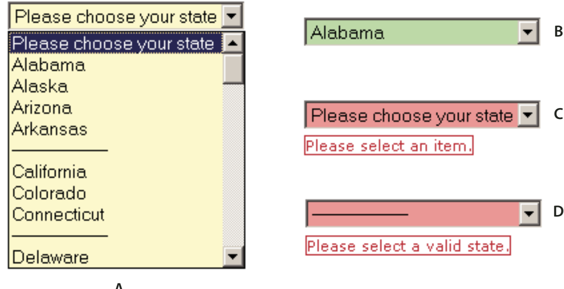
Option	Description
requiredClass	Overrides the "textareaRequiredState" built-in value
validClass	Overrides the "textareaValidState" built-in value
focusClass	Overrides the "textareaFocusState" built-in value
invalidCharsMinClass	Overrides the "textareaMinCharsState" built-in value
invalidCharsMaxClass	Overrides the "textareaMaxCharsState" built-in value
textareaFlashTextClass	Overrides the "textareaFlashText" built-in value

Working with the Validation Select widget

Validation Select widget overview and structure

A Spry Validation Select widget is a drop-down menu that displays valid or invalid states when the user makes a selection. For example, you can insert a Validation Select widget that contains a list of states, grouped into different sections and divided by horizontal lines. If the user accidentally selects one of the divider lines as opposed to one of the states, the Validation Select widget returns a message to the user stating that their selection is invalid.

The following example shows an expanded Validation Select widget, as well as the collapsed form of the widget in various states.



A. *Validation Select widget in focus* *B.* *Select widget, valid state* *C.* *Select widget, required state* *D.* *Select widget, invalid state*

The Validation Select widget includes a number of states (for example, valid, invalid, required value, and so on). You can alter the properties of these states by using the Property inspector, depending on the desired validation results. A Validation Select widget can validate at various points (for example, when the user clicks outside the widget, as the user makes selections, or when the user tries to submit the form).

Initial state When the page loads in the browser, or when the user resets the form.

Focus state When the user clicks the widget.

Valid state When the user selects a valid item, and the form can be submitted.

Invalid state When the user selects an invalid item.

Required state When the user fails to select a valid item.

Whenever a Validation Select widget enters one of the preceding states through user interaction, the Spry framework logic applies a specific CSS class to the HTML container for the widget at run time. For example, if a user tries to submit a form, but did not select an item from the menu, Spry applies a class to the widget so that it displays the error message, “Please select an item.” The rules that control the style and display states of error messages reside in the SpryValidationSelect.css file that accompanies the widget.

The default HTML code for the Validation Select widget, usually inside of a form, is made up of a container `span` tag that surrounds the `select` tag of the text area. The HTML code for the Validation Select widget also includes `script` tags in the head of the document and after the widget’s HTML code.

The HTML code for the Validation Select widget also includes `script` tags in the head of the document and after the widget’s HTML code. The `script` tag in the head of the document defines all of the JavaScript functions related to the Select widget. The `script` tag after the widget code creates a JavaScript object that makes the widget interactive.

Following is the HTML code for a Validation Select widget:

```
<head>
...
<!-- Link the Spry Validation Select JavaScript library -->
<script src="SpryAssets/SpryValidationSelect.js" type="text/javascript"></script>
<!-- Link the CSS style sheet that styles the widget -->
<link href="SpryAssets/SpryValidationSelect.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" name="form1" method="post" action="">
        <!-- Create the select widget and assign a unique id-->
        <span id="spryselect1">
        <select name="select1" id="select1">
            <!-- Add items and values to the widget-->
            <option>--Please select an item--</option>
            <option value="item1">Item 1</option>
            <option value="item2">Item 2</option>
            <option value="-1">Invalid Item</option>
            <option value="item3">Item 3</option>
            <option>Empty Item</option>
        </select>
        <!--Add an error message-->
        <span class="selectRequiredMsg">Please select an item.</span>
        </span>
    </form>
<!-- Initialize the Validation Select widget object-->
<script type="text/javascript">
var spryselect1 = new Spry.Widget.ValidationSelect("spryselect1");
</script>
</body>
```

In the code, the `new` JavaScript operator initializes the Select widget object, and transforms the `span` content with the ID of `spryselect1` from static HTML code into an interactive page element.

The `span` tag for the error message in the widget has a CSS class applied to it. This class (which is set to `display:none;` by default), controls the style and visibility of the error message, and exists in the accompanying SpryValidationSelect.css file. When the widget enters different states as a result of user interaction, Spry places different classes on the container for the widget, which in turn affects the error-message class.

You can add other error messages to a Validation Select widget by creating a `span` tag (or any other type of tag) to hold the text of the error message. Then, by applying a CSS class to it, you can hide or show the message, depending on the widget state.

You can change the default appearance of the Validation Select widget's state by editing the corresponding CSS rule in the SpryValidationSelect.css file. For example, to change the background color for a state, edit the corresponding rule or add a new rule (if it's not already present) in the style sheet.

Variation on tags used for Select widget structure

In the preceding example, `span` tags create the structure for the widget:

```
Container SPAN
    SELECT tag
    Error message SPAN
```

You can, however, use almost any container tag to create the widget.

```
Container DIV
    SELECT tag
    Error Message P
```

Spry uses the tag ID (not the tag itself) to create the widget. Spry also displays error messages using CSS code that is indifferent to the actual tag used to contain the error message.

The ID passed into the widget constructor identifies a specific HTML element. The constructor finds this element and looks inside the identified container for a corresponding `select` tag. If the ID passed to the constructor is the ID of the `select` tag (rather than a container tag), the constructor attaches validation triggers directly to the `select` tag. If no container tag is present, however, the widget cannot display error messages, and different validation states will only alter the appearance of the `select` tag element (for example, its background color).

Note: *Multiple `select` tags do not work inside the same HTML widget container. Each select list should be its own widget.*

CSS code for the Validation Select widget

The SpryValidationSelect.css file contains the rules that style the Validation Select widget and its error messages. You can edit these rules to style the look and feel of the widget and error messages. The names of the rules in the CSS file correspond to the names of the classes specified in the widget's HTML code.

The following is the CSS code for the SpryValidationSelect.css file:

```
/*Validation Select styling classes*/
.selectRequiredMsg, .selectInvalidMsg {
    display: none;
}
.selectRequiredState .selectRequiredMsg,
.selectInvalidState .selectInvalidMsg {
    display: inline;
    color: #CC3333;
    border: 1px solid #CC3333;
}
.selectValidState select, select.selectValidState {
    background-color: #B8F5B1;
}
select.selectRequiredState, .selectRequiredState select,
select.selectInvalidState, .selectInvalidState select {
    background-color: #FF9F9F;
}
.selectFocusState select, select.selectFocusState {
    background-color: #FFFFCC;
}
```

The SpryValidationSelect.css file also contains extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Insert the Validation Select widget

1 Locate the SpryValidationSelect.js file and add it to your web site. You can find the SpryValidationSelect.js file in the widgets/selectvalidation directory, located in the Spry directory that you downloaded from Adobe Labs. For more information, see “Prepare your files” on page 3.

For example, create a folder called *SpryAssets* in the root folder of your web site, and upload the SpryValidation-Select.js file to it. The SpryValidationSelect.js file contains all of the information necessary for making the Select widget interactive.

2 Locate the SpryValidationSelect.css file and add it to your web site. You might choose to add it to the main directory, a SpryAssets directory, or to a CSS directory if you have one.

3 Open the web page to add the Select widget to and link the SpryValidationSelect.js file to the page by inserting the following `script` tag in the page’s head tag:

```
<script src="SpryAssets/SpryValidationSelect.js" type="text/javascript"></script>
```

Make sure that the file path to the SpryValidationSelect.js file is correct. This path varies depending on where you’ve placed the file in your web site.

4 Link the SpryValidationSelect.css file to your web page by inserting the following `link` tag in the page’s head tag:

```
<link href="SpryAssets/SpryValidationSelect.css" rel="stylesheet" type="text/css" />
```

Make sure that the file path to the SpryValidationSelect.css file is correct. This path varies depending on where you’ve placed the file in your web site.

5 Add a select list to the page and give it a name and a unique ID:

```
<select name="myselect" id="myselect"></select>
```

6 Add options to the select list, as follows:

```
<select name="myselect" id="myselect">
    <option>--Please select an item--</option>
    <option value="item1">Item 1</option>
    <option value="item2">Item 2</option>
    <option value="-1">Invalid Item</option>
    <option value="item3">Item 3</option>
    <option>Empty Item</option>
</select>
```

7 To add a container around the select list, insert `span` tags around the `select` tags, and assign a unique ID to the widget, as follows:

```
<span id="spryselect1">
    <select name="myselect" id="myselect">
        <option>--Please select an item--</option>
        <option value="item1">Item 1</option>
        <option value="item2">Item 2</option>
        <option value="-1">Invalid Item</option>
        <option value="item3">Item 3</option>
        <option>Empty Item</option>
    </select>
</span>
```

8 To initialize the Spry validation select object, insert the following `script` block after the HTML code for the widget:

```
<script type="text/javascript">
    var spryselect1 = new Spry.Widget.ValidationSelect("spryselect1");
</script>
```

The `new` JavaScript operator initializes the Select widget object, and transforms the `span` tag content with the ID of `spryselect1` from static HTML code into an interactive select object. The `Spry.Widget.ValidationSelect` method is a constructor in the Spry framework that creates select objects. The information necessary to initialize the object is contained in the SpryValidationSelect.js JavaScript library that you linked to at the beginning of this procedure.

Make sure that the ID of the select list's container `span` tag matches the ID parameter you specified in the Spry.Widgets.ValidationSelect method. Make sure that the JavaScript call comes after the HTML code for the widget.

9 Save the page.

The complete code looks as follows:

```
<head>
...
<script src="SpryAssets/SpryValidationSelect.js" type="text/javascript"></script>
<link href="SpryAssets/SpryValidationSelect.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <span id="spryselect1">
        <select name="myselect" id="myselect">
            <option>--Please select an item--</option>
            <option value="item1">Item 1</option>
            <option value="item2">Item 2</option>
            <option value="-1">Invalid Item</option>
            <option value="item3">Item 3</option>
            <option>Empty Item</option>
        </select>
    </span>
<script type="text/javascript">
    var spryselect1 = new Spry.Widget.ValidationSelect("spryselect1");
</script>
</body>
```

Display error messages

❖ Create a `span` tag (or any other type of tag) to display the error message, and assign the appropriate class to it, as follows:

```
<span id="spryselect1">
    <select name="select1" id="select1">
        <option>--Please select an item--</option>
        <option value="item1">Item 1</option>
        . . .
    </select>
    <span class="selectRequiredMsg">Please select an item.</span>
</span>
```

The `selectRequiredMsg` rule is located in the SpryValidationSelect.css file, and is set to `display:none` by default. When the widget enters a different state through user interaction, Spry applies the appropriate class—the state class—to the container of the widget. This action affects the error-message class, and by extension, the appearance of the error message.

For example, the following shows a portion of the CSS rule from the SpryValidationSelect.css file:

```
.selectRequiredMsg, .selectInvalidMsg {
    display: none;
}
.selectRequiredState .selectRequiredMsg,
.selectInvalidState .selectInvalidMsg {
    display: inline;
    color: #CC3333;
    border: 1px solid #CC3333;
}
```

By default, no state class is applied to the widget container, so that when the page loads in a browser, the error message text in the preceding HTML code example only has the `.selectRequiredMsg` class applied to it. (The property and value pair for this rule is `display:none`, so the message remains hidden.) If the user fails to make a selection, however, Spry applies the appropriate class to the widget container, as follows:

```
<span id="spryselect1" class="selectRequiredState">
    <select name="select1" id="select1">
        <option>--Please select an item--</option>
        <option value="item1">Item 1</option>
        . . .
    </select>
    <span class="selectRequiredMsg">Please select an item.</span>
</span>
```

In the preceding CSS code, the state rule with the contextual selector `.selectRequiredState . selectRequiredMsg` overrides the default error-message rule responsible for hiding the error-message text. Thus, when Spry applies the state class to the widget container, the state rule determines the appearance of the widget, and displays the error message inline in red with a 1-pixel solid border.

Following is a list of default error-message classes and their descriptions. You can change these classes and rename them. If you do so, don't forget to change them in the contextual selector also.

Error message class	Description
.selectRequiredMsg	Causes error message to display when the widget enters the required state
.selectInvalidMsg	Causes error message to display when the widget enters the invalid state

***Note:** You cannot rename state-related class names because they are hard-coded as part of the Spry framework.*

Specify when validation occurs

By default, the Validation Select widget validates when the user clicks the submit button. You can, however, set two other options: `blur` or `change`. The `validateOn:["blur"]` parameter causes the widget to validate whenever the user clicks outside the select list. The `validateOn:["change"]` parameter causes the widget to validate as the user makes selections.

❖ To specify when validation occurs, add a `validateOn` parameter to the constructor as follows:

```
<script type="text/javascript">
    var spryselect1 = new Spry.Widget.ValidationSelect("spryselect1",
{validateOn:["blur"]});
</script>
```

As a convenience, you can discard the brackets if your `validateOn` parameter contains a single value (for example, `validateOn: "blur"`). If the parameter contains both values, however (`validateOn:["blur", "change"]`), include brackets in the syntax.

Change required status of a select list

By default, Validation Select widgets require the user to make a selection before submitting the form. You can, however, make selections optional for the user.

❖ To change the required status of a select list, add the `isRequired` property to the constructor and set its value to false, as follows:

```
<script type="text/javascript">
    var selectwidget1 = new Spry.Widget.ValidationSelect("selectwidget1",
{isRequired:false});
</script>
```

Specify an invalid value

You can specify a value that registers as invalid if the user selects a menu item that is associated with that value. For example, if you specify -1 as an invalid value, and you assign the value to an option tag, the widget returns an error message if the user selects that menu item.

1 Assign a negative value to an option tag, for example:

```
<option value="-1"> ------------------- </option>
```

2 Add the invalid option and the value you specified to the widget constructor, as follows:

```
<script type="text/javascript">
    var selectwidget1 = new Spry.Widget.ValidationSelect("selectwidget1", {invalidValue:"-
1"});
</script>
```

Customize the Validation Select widget

The SpryValidationSelect.css file provides the default styling for the Validation Select widget. You can, however, customize the widget by changing the appropriate CSS rule. The CSS rules in the SpryValidationSelect.css file use the same class names as the related elements in the widget's HTML code, so it's easy for you to know which CSS rules correspond to the widget and its error states.

The SpryValidationSelect.css file should already be included in your website before you start customizing. For more information, see "Prepare your files" on page 3.

Style a Validation Select widget (general instructions)

1 Open the SpryValidationSelect.css file.

2 Locate the CSS rule for the part of the widget to change. For example, to change the background color of the Select widget's required state, edit the `.selectRequiredState` rule in the SpryValidationSelect.css file.

3 Make your changes to the CSS rule and save the file.

The SpryValidationSelect.css file contains extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Style Validation Select widget error message text

By default, error messages for the Validation Select widget appear in red with a 1-pixel solid border surrounding the text.

❖ To change the text styling of Validation Select widget error messages, use the following table to locate the appropriate CSS rule, and then change the default properties, or add your own text styling properties and values.

Text to style	Relevant CSS rule	Relevant properties to change
Error message text	.selectRequiredState .selectRequiredMsg, .selectInvalidState .selectInvalidMsg	color: #CC3333; border: 1px solid #CC3333;

Change Validation Select widget background colors

❖ To change the background colors of the Validation Select widget in various states, use the following table to locate the appropriate CSS rule, and then change the default background color values.

Background color to change	Relevant CSS rule	Relevant property to change
Background color of widget in valid state	.selectValidState select, select.selectValidState	background-color: #B8F5B1;
Background color of widget in invalid state	select.selectRequiredState, .selectRequiredState select, select.selectInvalidState, .selectInvalidState select	background-color: #FF9F9F;
Background color of widget in focus	.selectFocusState select, select.selectFocusState	background-color: #FFFFCC;

Customize state-related class names

While you can replace error message-related class names with class names of your own by changing the rules in the CSS and the class names in the HTML code, you cannot change or replace state-related class names, because the behaviors are hard-coded as part of the Spry framework. You can, however, override the default state-related class name with your own class name by specifying a new value in the third parameter of the widget constructor.

❖ To change widget state-related class names, add one of the overriding options to the third parameter of the widget constructor, and specify your custom class name, as follows:

```
<script type="text/javascript">
    var spryselect1 = new Spry.Widget.ValidationSelect("spryselect1",
{requiredClass:"required"});
</script>
```

The following table provides a list of options you can use to override built-in state-related class names.

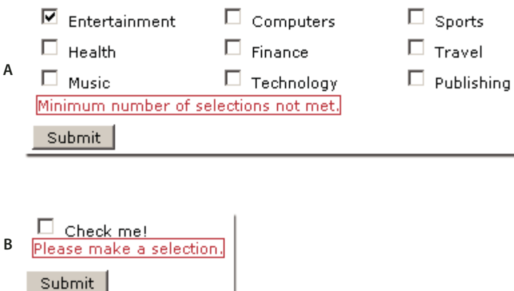
Option	Description
requiredClass	Overrides the "selectRequiredState" built-in value
validClass	Overrides the "selectValidState" built-in value
focusClass	Overrides the "selectFocusState" built-in value
invalidClass	Overrides the "selectInvalidState" built-in value

Working with the Validation Checkbox widget

Validation Checkbox widget overview and structure

A Spry Validation Checkbox widget is a checkbox or group of checkboxes in an HTML code form that display valid or invalid states when the user selects or fails to select a checkbox. For example, you can add a Validation Checkbox widget to a form in which a user might be required to make three selections. If the user fails to make all three selections, the widget returns a message stating that the minimum number of selections was not met.

The following example shows a Validation Checkbox widget in various states.



A. Validation checkbox widget group, minimum number of selections state B. Validation Checkbox widget, required state

The Validation Checkbox widget includes a number of states (for example, valid, invalid, required value, and so on). You can alter the properties of these states by using the Property inspector, depending on the desired validation results. A Validation Checkbox widget can validate at various points (for example, when the user clicks outside the widget, as the user makes selections, or when the user tries to submit the form).

Initial state When the page loads in the browser, or when the user resets the form.

Valid state When the user makes a selection, or the correct number of selections, and the form can be submitted.

Required state When the user fails to make a required selection.

Minimum Number Of Selections state When the user selects fewer than the minimum number of checkboxes required.

Maximum Number Of Selections state When the user selects greater than the maximum number of checkboxes allowed.

Whenever a Validation Checkbox widget enters one of these states through user interaction, the Spry framework logic applies a specific CSS class to the HTML container for the widget at run time. For example, if a user tries to submit a form, but makes no selections, Spry applies a class to the widget that causes it to display the error message, “Please make a selection.” The rules that control the style and display states of error messages reside in the SpryValidationCheckbox.css file that accompanies the widget.

The default HTML code for the Validation Checkbox widget, usually inside a form, is made up of a container `span` tag that surrounds the `input type="checkbox"` tag of the checkbox. The HTML code for the Validation Checkbox widget also includes `script` tags in the head of the document and after the widget’s HTML code.

The HTML code for the Validation Checkbox widget also includes `script` tags in the head of the document and after the widget’s HTML code. The `script` tag in the head of the document defines all of the JavaScript functions related to the Checkbox widget. The `script` tag after the widget code creates a JavaScript object that makes the checkbox interactive.

Following is the HTML code for a Validation Checkbox widget:

```
<head>
...
<!-- Link the Spry Validation Checkbox JavaScript library -->
<script src="SpryAssets/SpryValidationCheckbox.js" type="text/javascript"></script>
<!-- Link the CSS style sheet that styles the widget -->
<link href="SpryAssets/SpryValidationCheckbox.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" name="form1" method="post" action="">
        <!-- Create the checkbox widget and assign a unique id-->
        <span id="sprycheckbox1">
            <input type="checkbox" name="checkbox1" value="1"/>
            <input type="checkbox" name="checkbox2" value="2"/>
            <!--Add an error message-->
            <span class="checkboxRequiredMsg">Please make a selection.</span>
        </span>
    </form>
<!-- Initialize the Validation Checkbox widget object-->
<script type="text/javascript">
    var sprycheckbox1 = new Spry.Widget.ValidationCheckbox("sprycheckbox1");
</script>
</body>
```

In the code, the `new` JavaScript operator initializes the Checkbox widget object, and transforms the `span` content with the ID of `sprycheckbox1` from static HTML code into an interactive page element.

The `span` tag for the error message in the widget has a CSS class applied to it. This class (which is set to `display:none;` by default), controls the style and visibility of the error message, and exists in the accompanying SpryValidationCheckbox.css file. When the widget enters different states as a result of user interaction, Spry places different classes on the container for the widget, which in turn affects the error-message class.

You can add other error messages to a Validation Checkbox widget by creating a `span` tag (or any other type of tag) to hold the text of the error message. Then, by applying a CSS class to it, you can hide or show the message, depending on the widget state.

You can change the default appearance of the Validation Checkbox widget's states by editing the corresponding CSS rule in the SpryValidationCheckbox.css file. For example, to change the background color for a state, edit the corresponding rule or add a new rule (if it’s not already present) in the style sheet.

Variation on tags used for Text Field widget structure

In the preceding example, `span` tags create the structure for the widget:

```
Container SPAN
    INPUT type="checkbox"
    Error message SPAN
```

You can, however, use almost any container tag to create the widget.

```
Container DIV
    INPUT type="checkbox"
    Error Message P
```

Spry uses the tag ID (not the tag itself) to create the widget. Spry also displays error messages using CSS code that is indifferent to the actual tag used to contain the error message.

The ID passed into the widget constructor identifies a specific HTML element. The constructor finds this element and looks inside the identified container for a corresponding `input` tag. If the ID passed to the constructor is the ID of the `input` tag (rather than a container tag), the constructor attaches validation triggers directly to the `input` tag. If no container tag is present, however, the widget cannot display error messages, and different validation states alter only the appearance of the `input` tag element (for example, its background color).

CSS code for the Validation Checkbox widget

The SpryValidationCheckbox.css file contains the rules that style the Validation Checkbox widget and its error messages. You can edit these rules to style the look and feel of the widget and error messages. The names of the rules in the CSS file correspond to the names of the classes specified in the widget's HTML code.

The following is the CSS code for the SpryValidationCheckbox.css file:

```
/*Validation Checkbox styling classes*/
.checkboxRequiredMsg, .checkboxMinSelectionsMsg, .checkboxMaxSelectionsMsg{
    display: none;
}
.checkboxRequiredState .checkboxRequiredMsg,
.checkboxMinSelectionsState .checkboxMinSelectionsMsg,
.checkboxMaxSelectionsState .checkboxMaxSelectionsMsg {
    display: inline;
    color: #CC3333;
    border: 1px solid #CC3333;
}
```

The SpryValidationCheckbox.css file also contains extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Insert the Validation Checkbox widget

1 Locate the SpryValidationCheckbox.js file and add it to your web site. You can find the SpryValidationCheckbox.js file in the widgets/checkboxvalidation directory, located in the Spry directory that you downloaded from Adobe Labs. For more information, see "Prepare your files" on page 3.

For example, create a folder called *SpryAssets* in the root folder of your web site, and upload the SpryValidationCheckbox.js file to it. The SpryValidationCheckbox.js file contains all of the information necessary for making the Checkbox widget interactive.

2 Locate the SpryValidationCheckbox.css file and add it to your web site. You might choose to add it to the main directory, a SpryAssets directory, or to a CSS directory if you have one.

3 Open the web page to add the Checkbox widget to and link the SpryValidationCheckbox.js file to the page by inserting the following `script` tag in the page's head tag:

```
<script src="SpryAssets/SpryValidationCheckbox.js" type="text/javascript"></script>
```

Make sure that the file path to the SpryValidationCheckbox.js file is correct. This path varies depending on where you've placed the file in your web site.

4 Link the SpryValidationCheckbox.css file to your web page by inserting the following `link` tag in the page's head tag:

```
<link href="SpryAssets/SpryValidationCheckbox.css" rel="stylesheet" type="text/css" />
```

Make sure that the file path to the SpryValidationCheckbox.css file is correct. This path varies depending on where you've placed the file in your web site.

5 Add checkboxes to the page and assign them names and values. You can add an unlimited number of checkboxes.

```
<input type="checkbox" name="checkbox1" value="1"/>
<input type="checkbox" name="checkbox2" value="2"/>
```

6 Add a container around the checkboxes by inserting `span` tags around the `input` tags, and assigning a unique ID to the widget, as follows:

```
<span id="sprycheckbox1">
    <input type="checkbox" name="checkbox1" value="1"/>
    <input type="checkbox" name="checkbox2" value="2"/>
</span>
```

7 To initialize the Spry validation checkbox object, insert the following `script` block after the HTML code for the widget:

```
<script type="text/javascript">
    var sprycheckbox1 = new Spry.Widget.ValidationCheckbox("sprycheckbox1");
</script>
```

The `new` JavaScript operator initializes the Checkbox widget object, and transforms the `span` tag content with the ID of `sprycheckbox1` from static HTML code into an interactive checkbox object. The `Spry.Widget.ValidationCheckbox` method is a constructor in the Spry framework that creates checkbox objects. The information necessary to initialize the object is contained in the SpryValidationCheckbox.js JavaScript library that you linked to at the beginning of this procedure.

Make sure that the ID of the checkbox widget's container `span` tag matches the ID parameter you specified in the `Spry.Widgets.ValidationCheckbox` method. Make sure that the JavaScript call comes after the HTML code for the widget.

8 Save the page.

The complete code looks as follows:

```
<head>
...
<script src="SpryAssets/SpryValidationCheckbox.js" type="text/javascript"></script>
<link href="SpryAssets/SpryValidationCheckbox.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <span id="sprycheckbox1">
        <input type="checkbox" name="checkbox1" value="1"/>
        <input type="checkbox" name="checkbox2" value="2"/>
    </span>
<script type="text/javascript">
    var sprycheckbox1 = new Spry.Widget.ValidationCheckbox("sprycheckbox1");
</script>
</body>
```

Display error messages

❖ Create a `span` tag (or any other type of tag) to display the error message, and assign the appropriate class to it, as follows:

```
<span id="sprycheckbox1">
    <input type="checkbox" name="checkbox1" value="1"/>
    <input type="checkbox" name="checkbox2" value="2"/>
    <span class="checkboxRequiredMsg">Please make a selection.</span>
</span>
```

The `checkboxRequiredMsg` rule is located in the SpryValidationCheckbox.css file, and is set to `display:none` by default. When the widget enters a different state through user interaction, Spry applies the appropriate class—the state class—to the container of the widget. This action affects the error-message class, and by extension, the appearance of the error message.

For example, the following shows the CSS rule from the SpryValidationCheckbox.css file:

```
.checkboxRequiredMsg, .checkboxMinSelectionsMsg, .checkboxMaxSelectionsMsg{
    display: none;
}
.checkboxRequiredState .checkboxRequiredMsg,
.checkboxMinSelectionsState .checkboxMinSelectionsMsg,
.checkboxMaxSelectionsState .checkboxMaxSelectionsMsg {
    display: inline;
    color: #CC3333;
    border: 1px solid #CC3333;
}
```

By default, no state class is applied to the widget container, so that when the page loads in a browser, the error message text in the preceding HTML code example only has the `checkboxRequiredMsg` class applied to it. (The property and value pair for this rule is `display:none`, so the message remains hidden.) If the user fails to make a selection, however, Spry applies the appropriate class to the widget container, as follows:

```
<span id="sprycheckbox1" class="checkboxRequiredState">
    <input type="checkbox" name="checkbox1" value="1"/>
    <input type="checkbox" name="checkbox2" value="2"/>
    <span class="checkboxRequiredMsg">Please make a selection.</span>
</span>
```

In the preceding CSS code, the state rule with the contextual selector `.checkboxRequiredState .checkboxRequiredMsg` overrides the default error-message rule responsible for hiding the error message text. Thus, when Spry applies the state class to the widget container, the state rule determines the appearance of the widget, and displays the error message inline in red with a 1-pixel solid border.

Following is a list of default error-message classes and their descriptions. You can change these classes and rename them. If you do so, don't forget to change them in the contextual selector also.

Error message class	Description
.checkboxRequiredMsg	Causes error message to display when the widget enters the required state
.checkboxMinSelectionsMsg	Causes error message to display when the widget enters the minimum number of selections state
.checkboxMaxSelectionsMsg	Causes error message to display when the widget enters the maximum number of selections state

***Note:** You cannot rename state-related class names because they are hard-coded as part of the Spry framework.*

Specify when validation occurs

By default, the Validation Checkbox widget validates when the user clicks the submit button. You can, however, set two other options: blur or change. The `validateOn:["blur"]` parameter causes the widget to validate whenever the user clicks outside the widget. The `validateOn:["change"]` parameter causes the widget to validate as the user makes selections.

❖ To specify when validation occurs, add a `validateOn` parameter to the constructor as follows:

```
<script type="text/javascript">
    var sprycheckbox1 = new Spry.Widget.ValidationCheckbox("sprycheckbox1",
{validateOn:["blur"]});
</script>
```

As a convenience, you can discard the brackets if your `validateOn` parameter contains a single value (for example, `validateOn: "blur"`). If the parameter contains both values, however (`validateOn:["blur", "change"]`), include brackets in the syntax.

Specify a minimum and maximum number of selections

By default, a Validation Checkbox widget is set to `required`. If you insert a number of checkboxes on your page, however, you can specify a minimum and maximum selection range. For example, if you have six checkboxes within the `span` tag for a single Validation Checkbox widget, and you want to make sure that the user selects at least three checkboxes, you can set such a requirement for the entire widget.

❖ To specify a minimum or maximum number of selections, add the `minSelections` property or `maxSelections` property (or both) and a value to the constructor, as follows:

```
<script type="text/javascript">
    var checkboxwidget1 = new
Spry.Widget.ValidationCheckbox("checkboxwidget1",{minSelections:value,
maxSelections:value});
</script>
```

Change required status of checkboxes

By default, Validation Checkbox widgets require the user to make at least one selection before submitting the form. You can, however, make selections optional for the user.

❖ To change the required status of a checkbox, add the `isRequired` property to the constructor and set its value to false, as follows:

```
<script type="text/javascript">
    var checkboxwidget1 = new Spry.Widget.ValidationCheckbox("checkboxwidget1",
{isRequired:false});
</script>
```

Customize the Validation Checkbox widget

The SpryValidationCheckbox.css file provides the default styling for the Validation Checkbox widget. You can customize the widget by changing the appropriate CSS rule. The CSS rules in the SpryValidationCheckbox.css file use the same class names as the related elements in the widget's HTML code, so it's easy for you to know which CSS rules correspond to the widget and its error states.

The SpryValidationCheckbox.css file should already be included in your website before you start customizing. For more information, see "Prepare your files" on page 3.

Style a Validation Checkbox widget (general instructions)

1 Open the SpryValidationCheckbox.css file.

2 Locate the CSS rule for the part of the widget to change. For example, to change the background color of the Checkbox widget's required state, edit the `.checkboxRequiredState` rule in the SpryValidationCheckbox.css file.

3 Make your changes to the CSS rule and save the file.

The SpryValidationCheckbox.css file contains extensive comments, explaining the code and the purpose for certain rules. For further information, see the comments in the file.

Style Validation Checkbox widget error message text

By default, error messages for the Validation Checkbox widget appear in red with a 1-pixel solid border surrounding the text.

❖ To change the text styling of Validation Checkbox widget error messages, use the following table to locate the appropriate CSS rule, and then change the default properties, or add your own text styling properties and values.

Text to style	Relevant CSS rule	Relevant properties to change
Error message text	.checkboxRequiredState .checkboxRequiredMsg, .checkboxMinSelectionsState .checkboxMinSelectionsMsg, .checkboxMaxSelectionsState .checkboxMaxSelectionsMsg	color: #CC3333; border: 1px solid #CC3333;

Customize state-related class names

While you can replace error message-related class names with class names of your own by changing the rules in the CSS and the class names in the HTML code, you cannot change or replace state-related class names, because the behaviors are hard-coded as part of the Spry framework. You can, however, override the default state-related class name with your own class name by specifying a new value in the third parameter of the widget constructor.

❖ To change widget state-related class names, add one of the overriding options to the third parameter of the widget constructor, and specify your custom class name, as follows:

```
<script type="text/javascript">
    var sprycheckbox1 = new Spry.Widget.ValidationCheckbox("sprycheckbox1",
{requiredClass:"required"});
</script>
```

The following table provides a list of options you can use to override built-in state-related class names.

Option	Description
requiredClass	Overrides the "checkboxRequiredState" built-in value
minSelectionsClass	Overrides the "checkboxMinSelectionsState" built-in value
maxSelectionsClass	Overrides the "checkboxMaxSelectionsState" built-in value

Chapter 3: Working with Spry XML Data Sets

A Spry XML Data Set is a JavaScript object that you can use to display data from an XML data source file on a web page. You can then use this data to create master and detail regions on the page that update as site visitors make different selections.

About Spry XML Data Sets and dynamic regions

Spry XML Data Set basic overview

A Spry data set is fundamentally a JavaScript object. With a few snippets of code in your web page, you can create this object and load data from an XML source into the object when the user opens the page in a browser. The data set results in a flattened array of XML data that can be represented as a standard table containing rows and columns.

For example, suppose you have an XML source file, cafetownsend.xml, that contains the following information:

```
<?xml version="1.0" encoding="UTF-8"?>
<specials>
    <menu_item id="1">
        <item>Summer Salad</item>
        <description>organic butter lettuce with apples, blood oranges, gorgonzola, and 
raspberry vinaigrette.</description>
        <price>7</price>
    </menu_item>
    <menu_item id="2">
        <item>Thai Noodle Salad</item>
        <description>lightly sauteed in sesame oil with baby bok choi, portobello mushrooms, 
and scallions.</description>
        <price>8</price>
    </menu_item>
    <menu_item id="3">
        <item>Grilled Pacific Salmon</item>
        <description>served with new potatoes, diced beets, Italian parlsey, and lemon 
zest.</description>
        <price>16</price>
    </menu_item>
</specials>
```

Using XPath in your web page to indicate the data you're interested in (in this example, the specials/menu_item node of the XML file), the data set flattens the XML data into an array of objects (rows) and properties (columns), represented by the following table.

@id	item	description	price
1	Summer salad	organic butter lettuce with apples, blood oranges, gorgonzola, and raspberry vinaigrette.	7
2	Thai Noodle Salad	lightly sauteed in sesame oil with baby bok choi, portobello mushrooms, and scallions.	8
3	Grilled Pacific Salmon	served with new potatoes, diced beets, Italian parlsey, and lemon zest.	16

The data set contains a row for each menu item and the following columns: @id, item, description, and price. The columns represent the child nodes of the `specials/menu_item` node in the XML, plus any attributes contained in the `menu_item` tag, or in any of the child tags of the `menu_item` tag.

The data set also contains a built-in data reference called *ds_RowID* (not shown) that can be useful later when you display your data. Additionally the data set includes other built-in data references, for example, *ds_RecordCount*, *ds_CurrentRow*, and others that you can use to manipulate the data display.

You create a Spry data set object by using XPath in the `Spry.Data.XMLDataSet` constructor. The XPath defines the default structure of the data set. For example, if you use XPath to select a repeating XML node that includes three child nodes, the data set will have a row for each repeating node, and a column for each of the three child nodes. (If any of the repeating nodes or child nodes contain attributes, the data set also creates a column for each attribute.)

If you do not specify an XPath, all of the data in the XML source will be included in the data set.

After the data set is created, the data set object lets you easily display and manage the data. For example, you can create a simple table that displays the XML data, and then use simple methods and properties to reload, sort and filter, or page through data.

The following example creates a Spry data set called `dsSpecials`, and loads data from an XML file called *cafetownsend.xml*:

```
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
    <title>Spry Example</title>
    <!--Link the Spry libraries-->
    <script type="text/javascript" src="includes/xpath.js"></script>
    <script type="text/javascript" src="includes/SpryData.js"></script>
    <!--Create a data set object-->
    <script type="text/javascript">
        var dsSpecials = new Spry.Data.XMLDataSet("data/cafetownsend.xml",
"specials/menu_item");
    </script>
</head>
. . .
<body>
</body>
```

***Note:** The examples in this document are for reading purposes only and not intended for execution. For working samples, see the demos folder in the Spry folder on Adobe Labs.*

In the example, the first `script` tag links an open-source XPath library to the page where you'll eventually display XML data. The XPath library allows for the specification of a more complex XPath when you create a data set:

```
<script type="text/javascript" src="includes/xpath.js"></script>
```

The second `script` block links the SpryData.js Spry data library, which is stored in a folder called *includes* on the server:

```
<script type="text/javascript" src="includes/SpryData.js"></script>
```

The Spry data library depends on the XPath library, so it's important that you always link the XPath library first.

The third `script` block contains the statement that creates the `dsSpecials` data set. The cafetownsend.xml XML source file is stored in a folder called *data* on the server:

```
var dsSpecials = new Spry.Data.XMLDataSet("data/cafetownsend.xml", "specials/menu_item");
```

***Note:** Remember that JavaScript and XML are case-sensitive languages, so it's important that you make sure that the scripts and column names you specify are capitalized (or not capitalized) appropriately.*

In JavaScript the new operator is used to create objects. The `Spry.Data.XMLDataSet` method is a constructor in the Spry data library that creates new Spry data set objects. The constructor takes two parameters: the source of the data (`"data/cafetownsend.xml"`, in this case, a relative URL) and an XPath expression that specifies the node or nodes in the XML to supply the data (`"specials/menu_item"`).

You can also specify an absolute URL as the source of the XML data, as follows:

```
var dsSpecials = new
Spry.Data.XMLDataSet("http://www.somesite.com/somefolder/cafetownsend.xml",
"specials/menu_item");
```

***Note:** The URL you decide to use (whether absolute or relative) is subject to the browser's security model, which means that you can only load data from an XML source that is on the same server domain as the HTML page you're linking from. You can avoid this limitation by providing a cross-domain service script. For more information, consult your server administrator.*

In the preceding example, the constructor creates a new `dsSpecials` Spry data set object. The data set obtains data from the `specials/menu_item` node (specified by XPath) in the cafetownsend.xml XML file and converts the data to a flattened array of objects and properties, similar to the rows and columns of a table. (For an example of the table, see the beginning of this section.)

Each data set maintains the notion of a *current row*. By default, the current row is set to the first row in the data set. Later, you can change the current row programmatically by calling the `setCurrentRow()` method on the data set object. For more information, see "Set or change current row" on page 124.

***Note:** The data set contains no data after you've created it with the `new` JavaScript operator. To load data into the dataset, first call the data set's `loadData()` method, which executes a request to load the XML data. Spry regions and detail regions do this automatically for the data sets they depend on, but if you are not using one of these regions, call the `loadData()` method manually in your page code. This loading is asynchronous, so the data might still be unavailable if you try to access it immediately after calling `loadData()`.*

Spry XML Data Set advanced examples

Spry XML data sets use the `XMLHTTPRequest` object to asynchronously load the specified URL. When the XML data arrives, it is actually in two formats: a text format, and a document object model (DOM) tree format.

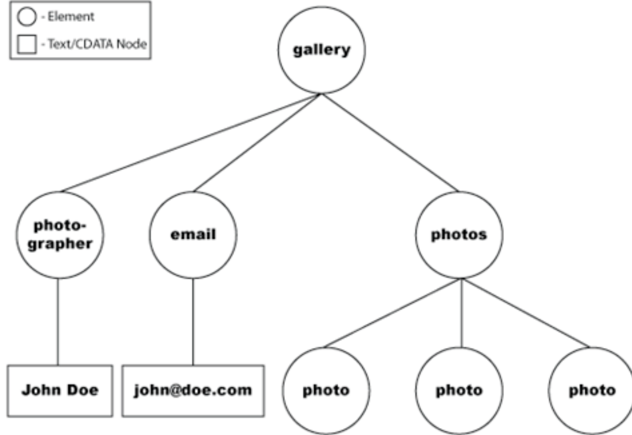
For example, say that you've specified `"/photos.php?galleryid=2000"` as your data source. (This is a path to a web service that retrieves XML data).

```
<script type="text/javascript">
    var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
</script>
```

The following code represents the data as it arrives in text format:

```
<gallery id="12345">
    <photographer id="4532">John Doe</photographer>
    <email>john@doe.com</email>
    <photos id="2000">
        <photo path="sun.jpg" width="16" height="16" />
        <photo path="tree.jpg" width="16" height="16" />
        <photo path="surf.jpg" width="16" height="16" />
    </photos>
</gallery>
```

The following example represents the data as it arrives in DOM-tree format.

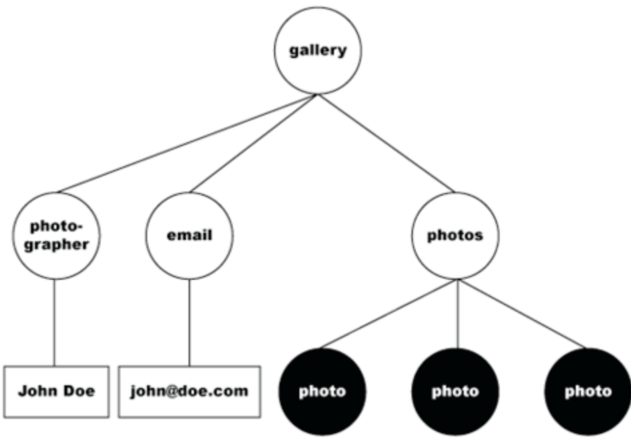


The data set then uses the XPath specified in the constructor to navigate the XML DOM tree to find the particular nodes that represent the data you are interested in.

The following code shows data selected with the `/gallery/photos/photo` XPath, in bold:

```
<gallery id="12345">
    <photographer id="4532">John Doe</photographer>
    <email>john@doe.com</email>
    <photos id="2000">
        <photo path="sun.jpg" width="16" height="16"/>
        <photo path="tree.jpg" width="16" height="16"/>
        <photo path="surf.jpg" width="16" height="16"/>
    </photos>
</gallery>
```

The following example is the DOM-tree representation of the selected nodes.



The data set then flattens the set of nodes into a tabular format, which the following table represents.

@path	@width	@height
sun.jpg	16	16
tree.jpg	16	16
surf.jpg	16	16

In this instance, Spry derives the column names of the flattened table from the selected nodes and their attributes. The way Spry determines column names, however, can vary, depending on the XPath you specify.

Spry uses the following guidelines when flattening data and creating columns:

- If the selected node has attributes, Spry creates a column for each attribute and places the value of the attribute in that column. The names for these columns are the names of the attributes preceded by an @ sign. For example, if a node has an `id` attribute, the column name is `@id`.
- If the selected node has no element children, and has text or CDATA underneath it, then Spry creates a column and places that text or CDATA in that column. The name of this column is the tag name of the node for normal XML elements.
- If the selected node has element children, Spry creates a column for the value of each element and its attributes, but only for each element child *that has no element children of its own*. The names of the columns are the tag names of the children elements, or in the case of an attribute on a child element, the format `"child-TagName/@attrName"`.
- If the selected node is an attribute, Spry creates a column for the attribute, and the column name is the name of the attribute preceded by an @ sign.
- Spry ignores element children that have their own element children.

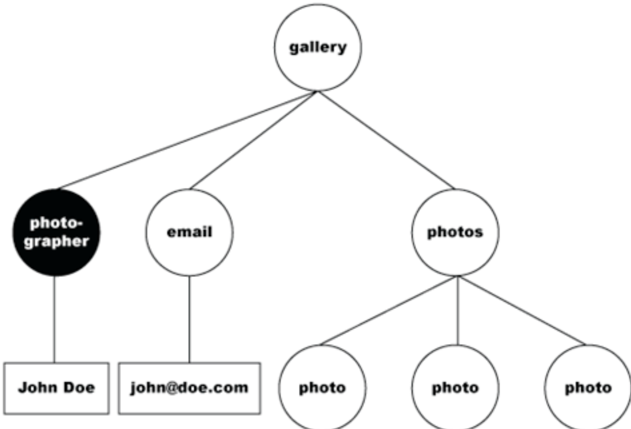
The examples that follow provide more details on the flattening process and how Spry generates column names for data sets.

Example of selecting and flattening an element with attributes and a text value

The following code shows data selected with the `/gallery/photographer` XPath, in bold:

```
<gallery id="12345">
    <photographer id="4532">John Doe</photographer>
    <email>john@doe.com</email>
    <photos id="2000">
        <photo path="sun.jpg" width="16" height="16" />
        <photo path="tree.jpg" width="16" height="16" />
        <photo path="surf.jpg" width="16" height="16" />
    </photos>
</gallery>
```

The following is the DOM-tree representation of the selected node.



The data set then flattens the selected data into the following table.

photographer	@id
John Doe	16

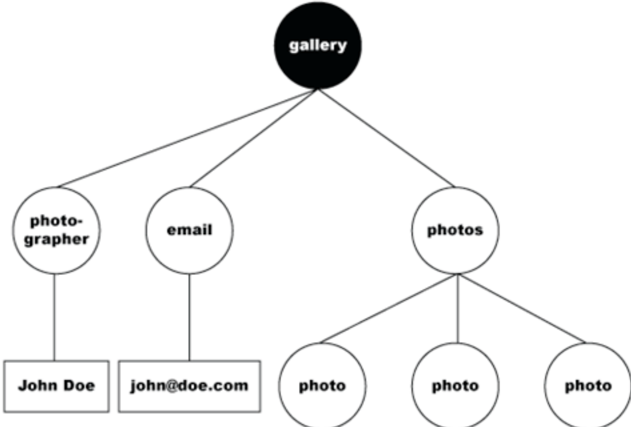
In this particular case, only one node is selected, so we only get one row in our data set. The value of the `photographer` node is the text `"John Doe"`, so a column named *photographer* is created to store that value. The `id` attribute is an attribute of the `photographer` node, so its value is placed in the @id column. All attribute names are preceded by an @ sign.

Example of selecting and flattening an element with attributes and element children

The following code shows data selected with the `/gallery` XPath:

```
<gallery id="12345">
    <photographer id="4532">John Doe</photographer>
    <email>john@doe.com</email>
    <photos id="2000">
        <photo path="sun.jpg" width="16" height="16" />
        <photo path="tree.jpg" width="16" height="16" />
        <photo path="surf.jpg" width="16" height="16" />
    </photos>
</gallery>
```

The following is the DOM-tree representation of the selected node:



The data set then flattens the selected data into the following table.

@id	photographer	photographer/@id	email
12345	John Doe	4532	john@doe.com

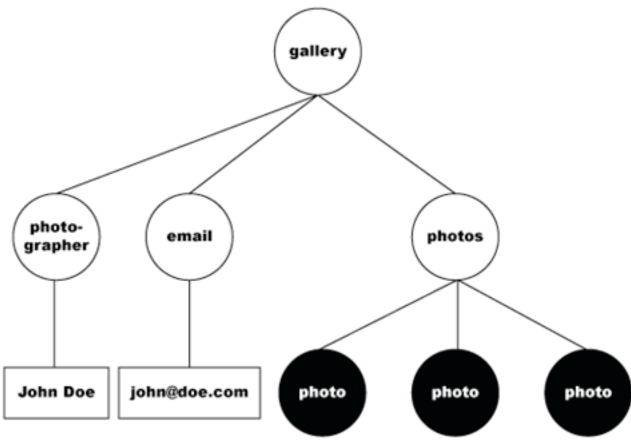
Notice that the column names for attributes of children elements are prefixed with the tag name of the child element. In this particular case, `photographer` is a child element of the selected `gallery` node, so its `id` attribute is prefixed with `photographer/@`. Also notice that nothing was added to the table for the `photos` element, even though it is a child of the `gallery` node. That is because Spry does not flatten any child elements that contain other elements.

Example of selecting an attribute of a single element and flattening it

With XPath you can also select attributes of nodes. The following code shows data selected with the `gallery/photos/photo/@path` XPath, in bold:

```
<gallery id="12345">
    <photographer id="4532">John Doe</photographer>
    <email>john@doe.com</email>
    <photos id="2000">
        <photo path="sun.jpg" width="16" height="16" />
        <photo path="tree.jpg" width="16" height="16" />
        <photo path="surf.jpg" width="16" height="16" />
    </photos>
</gallery>
```

The following is the DOM-tree representation of the selected nodes.



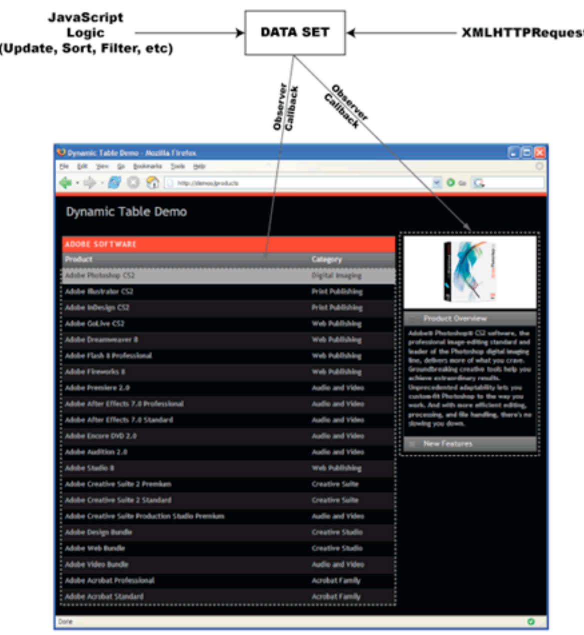
The data set then flattens the selected data into the following table.

@path
sun.jpg
tree.jpg
surf.jpg

Spry dynamic region overview and structure

After you've created a Spry data set, you can display the data in a Spry dynamic region. A Spry dynamic region is an area on a web page that's bound to a data set. The region displays the XML data from the data set and automatically updates the data display whenever the data set is modified.

Dynamic regions regenerate because they register themselves as observers or listeners of the data sets to which they are bound. Whenever data in any of these data sets is modified (loaded, updated, sorted, filtered, and so on), the data sets send notifications to all of their observers, triggering an automatic regeneration by the listening dynamic regions.



To declare a Spry dynamic region in a container tag, use the `spry:region` attribute. Most HTML elements can act as dynamic-region containers, however, the following tags cannot be used:

- `col`
- `colgroup`
- `frameset`
- `html`
- `iframe`
- select
- `style`
- `table`
- `tbody`
- `tfoot`
- `thead`
- `title`
- `tr`

While you cannot use any of the preceding HTML elements as Spry dynamic region containers, you can use them inside Spry dynamic-region containers.

Note: *Dynamic regions are limited to regions within the body tag. You can't add the* `spry:region` *attribute to any tag that is outside the body tag.*

In the following example, a container for a dynamic region called `Specials_DIV` is created using a `div` tag that includes a standard HTML table. Tables are typical HTML elements used for dynamic regions because the first row of the table can contain headings, and the second row can contain repeated XML data.

```
<!--Create the Spry dynamic region-->
<div id="Specials_DIV" spry:region="dsSpecials">
    <!--Display the data in a table-->
    <table id="Specials_Table">
        <tr>
            <th>Item</th>
            <th>Description</th>
            <th>Price</th>
        </tr>
        <tr spry:repeat="dsSpecials">
            <td>{item}</td>
            <td>{description}</td>
            <td>{price}</td>
        </tr>
    </table>
</div>
```

In the example, the `div` tag that creates the container for the dynamic region needs only two attributes: a `spry:region` attribute that declares the dynamic region and specifies the data set to use in it, and an `id` attribute that names the region:

```
<div id="Specials_DIV" spry:region="dsSpecials">
```

The new region is an observer of the `dsSpecials` data set. Any time the `dsSpecials` data set changes, the new dynamic region regenerates itself with the updated data.

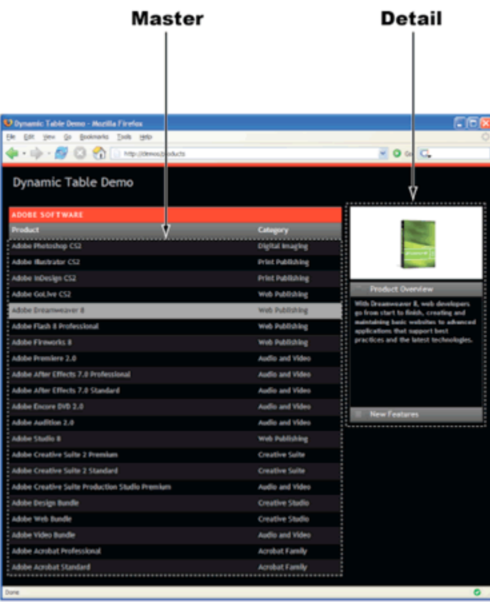
An HTML table displays the data:

```
<table id="Specials_Table">
    <tr>
        <th>Item</th>
        <th>Description</th>
        <th>Price</th>
    </tr>
    <tr spry:repeat="dsSpecials">
        <td>{item}</td>
        <td>{description}</td>
        <td>{price}</td>
    </tr>
</table>
```

The values in curly braces in the second row of the table—the data references—specify the columns in the data set. The data references bind the table cells to the data in specific columns of the data set. Because XML data often includes repeating nodes, the example also declares a `spry:repeat` attribute in the second table row tag. This causes all of the rows in the data set to appear when the user loads the page (instead of just the data set’s current row).

Spry basic master and detail region overview and structure

When working with Spry data sets, you can create master and detail dynamic regions to display more detail about your data. One region on the page (the master), drives the display of the data in another region on the page (the detail).



Typically, the master region displays an abbreviated form of a set of records from the data set, and the detail region displays more information about a selected record. Because the detail region depends on the master region, any time the data in the master region changes, the data in the detail region changes as well.

This section covers basic master and detail relationships where both regions are dependent on the same data set. For information on master and detail regions that use more than one data set, see “Spry advanced master and detail region overview and structure” on page 104.

In the following example, a master dynamic region displays data from the dsSpecials data set, and a detail dynamic region displays more detail about the row of data that’s been selected in the master region:

```
<head>
. . .
<script type="text/javascript" src="../includes/xpath.js"></script>
<script type="text/javascript" src="../includes/SpryData.js"></script>
<script type="text/javascript">
    var dsSpecials = new Spry.Data.XMLDataSet("data/cafetownsend.xml",
"specials/menu_item");
</script>
</head>
. . .
<body>
<!--Create a master dynamic region-->
<div id="Specials_DIV" spry:region="dsSpecials">
    <table id="Specials_Table">
        <tr>
            <th>Item</th>
            <th>Description</th>
            <th>Price</th>
        </tr>
        <!--User clicks to reset the current row in the data set-->
        <tr spry:repeat="dsSpecials" spry:setrow="dsSpecials">
            <td>{item}</td>
            <td>{description}</td>
            <td>{price}</td>
        </tr>
    </table>
</div>
<!--Create the detail dynamic region-->
<div id="Specials_Detail_DIV" spry:detailregion="dsSpecials">
    <table id="Specials_Detail_Table">
        <tr>
            <th>Ingredients</th>
            <th>Calories</th>
        </tr>
        <tr>
            <td>{ingredients}</td>
            <td>{calories}</td>
        </tr>
    </table>
</div>
. . .
</body>
```

Note: *The example XML file in “Spry XML Data Set basic overview” on page 91 does not contain nodes for ingredients or calories. These nodes are added to the data set for this example.*

In the example, the first `div` tag contains the `id` and `spry:region` attributes that create a container for the master dynamic region:

```
<div id="Specials_DIV" spry:region="dsSpecials">
```

The first table-row tag of the master region contains a `spry:setrow` attribute that sets the value of the current row in the data set.

```
<tr spry:repeat="dsSpecials" spry:setrow="dsSpecials">
```

The second `div` tag contains the attributes that create a container for the detail dynamic region:

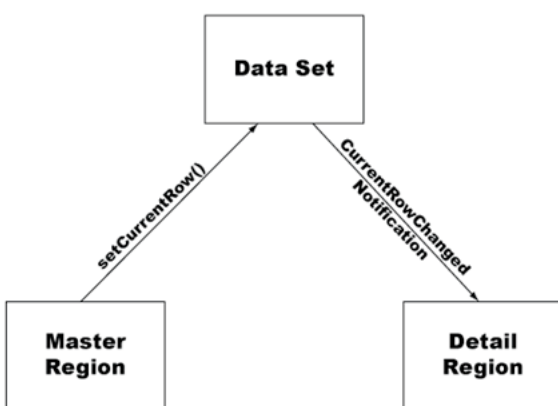
```
<div id="Specials_Detail_DIV" spry:detailregion="dsSpecials">
```

Every Spry data set maintains the notion of a *current row*. By default, the current row is set to the first row in the data set. A `spry:detailregion` works in exactly the same way as a `spry:region` except that when the data set's current row changes, the detail region updates automatically.

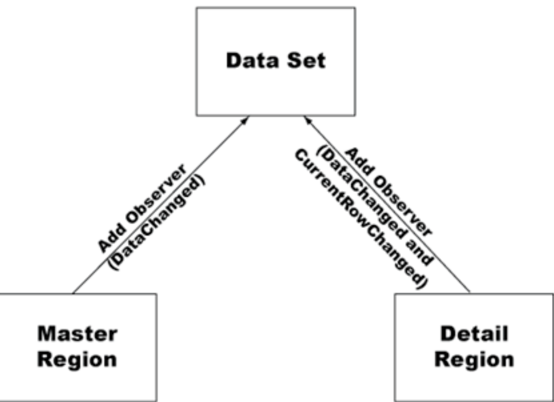
The binding expressions in the detail region (`{ingredients}` and `{calories}`) display data from the data set's current row when the page loads in a browser. When a user clicks a row in the master region table, however, the `spry:setrow` attribute changes the current row in the data set to the row the user selected.

The `{ds_RowID}` data reference is a built-in part of the Spry framework that points to an automatically generated unique ID for each row in the data set. (See "Data reference options" on page 135.) When the user selects a row in the master region table, the `spry:setrow` attribute supplies the unique ID to the `setCurrentRow` method, which sets the current row in the data set.

Whenever the data set is modified, all dynamic regions bound to that data set regenerate themselves and display the updated data. Because the detail region, like the master region, is an observer of the `dsSpecials` data set, it also changes as a result of the modification, and displays data related to the row the user selected (the new current row).



The difference between a `spry:region` and a `spry:detailregion` is that the `spry:detailregion` specifically listens for `CurrentRowChange` notifications (in addition to `DataChanged` notifications) from the data set, and updates itself when it receives one. Normal `spry:regions`, on the other hand, ignore the `CurrentRowChange` notification, and only update when they receive a `DataChanged` notification from the data set.



Spry advanced master and detail region overview and structure

In some cases, you might want to create master and detail relationships that involve more than one data set. For example, you might have a list of menu items that has a great deal of detail information associated with it. (This section uses a list of ingredients to illustrate the point.) Fetching all of the information associated with every menu item in a single query might be an inefficient use of bandwidth not to mention unnecessary, given that many users might not even be interested in the details of everything on the menu. Instead, it is more efficient to download only the detail data that the user is interested in when the user requests it, thus improving performance and reducing bandwidth. Limiting the amount of data exchange in this way is a common technique used to improve performance in AJAX applications.

Following is the XML source code for a sample file called *cafetownsend.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<specials>
    <menu_item id="1">
        <item>Summer Salad</item>
        <description>organic butter lettuce with apples, blood oranges, gorgonzola, and
raspberry vinaigrette.</description>
        <price>7</price>
        <url>summersalad.xml</url>
    </menu_item>
    <menu_item id="2">
        <item>Thai Noodle Salad</item>
        <description>lightly sauteed in sesame oil with baby bok choi, portobello mushrooms,
and scallions.</description>
        <price>8</price>
        <url>thainoodles.xml</url>
    </menu_item>
    <menu_item id="3">
        <item>Grilled Pacific Salmon</item>
        <description>served with new potatoes, diced beets, Italian parlsey, and lemon
zest.</description>
        <price>16</price>
        <url>salmon.xml</url>
    </menu_item>
</specials>
```

Note: *This XML sample code is different from the code used in “Spry XML Data Set basic overview” on page 91.*

The cafetownsend.xml file supplies the data for the master data set. The *url* node of the cafetownsend.xml file points to a unique XML file (or URL) for each menu item. These unique XML files contain a list of ingredients for the corresponding menu items. The summersalad.xml file, for example, might look as follows:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<item>
    <item_name>Summer salad</item_name>
    <ingredients>
        <ingredient>
            <name>butter lettuce</name>
        </ingredient>
        <ingredient>
            <name>Macintosh apples</name>
        </ingredient>
        <ingredient>
            <name>Blood oranges</name>
        </ingredient>
        <ingredient>
            <name>Gorgonzola cheese</name>
        </ingredient>
        <ingredient>
            <name>raspberries</name>
        </ingredient>
        <ingredient>
            <name>Extra virgin olive oil</name>
        </ingredient>
```

```
        <ingredient>
            <name>balsamic vinegar</name>
        </ingredient>
        <ingredient>
            <name>sugar</name>
        </ingredient>
        <ingredient>
            <name>salt</name>
        </ingredient>
        <ingredient>
            <name>pepper</name>
        </ingredient>
        <ingredient>
            <name>parsley</name>
        </ingredient>
        <ingredient>
            <name>basil</name>
        </ingredient>
    </ingredients>
</item>
```

When you are familiar with the structure of your XML code, you can create two data sets to use to display data in master and detail dynamic regions. In the following example, a master dynamic region displays data from the `dsSpecials` data set, and a detail dynamic region displays data from the dsIngredients data set:

```
<head>
. . .
<script type="text/javascript" src="../includes/xpath.js"></script>
<script type="text/javascript" src="../includes/SpryData.js"></script>
<script type="text/javascript">
<!--Create two separate data sets-->
    var dsSpecials = new Spry.Data.XMLDataSet("data/cafetownsend.xml",
"specials/menu_item");
    var dsIngredients = new Spry.Data.XMLDataSet("data/{dsSpecials::url}",
"item/ingredients/ingredient");
</script>
</head>
. . .
<body>
<!--Create a master dynamic region-->
<div id="Specials_DIV" spry:region="dsSpecials">
    <table id="Specials_Table">
        <tr>
            <th>Item</th>
            <th>Description</th>
            <th>Price</th>
        </tr>
        <!--User clicks to reset the current row in the data set-->
```

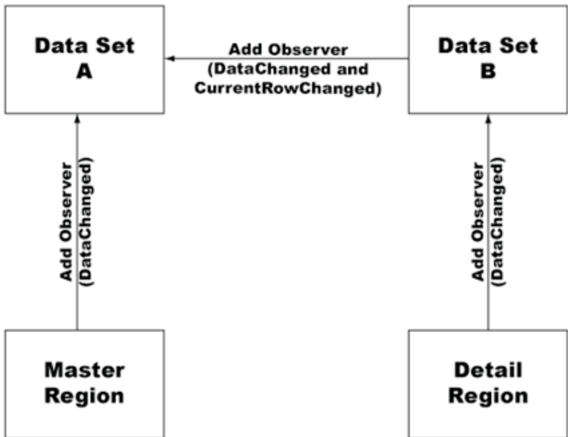
```
        <tr spry:repeat="dsSpecials" spry:setrow="dsSpecials">
            <td>{item}</td>
            <td>{description}</td>
            <td>{price}</td>
        </tr>
    </table>
</div>
<!--Create the detail dynamic region-->
<div id="Specials_Detail_DIV" spry:region="dsIngredients">
    <table id="Specials_Detail_Table">
        <tr>
            <th>Ingredients</th>
        </tr>
        <tr spry:repeat="dsIngredients">
            <td>{name}</td>
        </tr>
    </table>
</div>
. . .
</body>
```

In the example, the third `script` block contains the statement that creates two data sets, one called *dsSpecials* and one called *dsIngredients*:

```
var dsSpecials = new Spry.Data.XMLDataSet("data/cafetownsend.xml", "specials/menu_item");
var dsIngredients = new Spry.Data.XMLDataSet("data/{dsSpecials::url}",
"item/ingredients/ingredient");
```

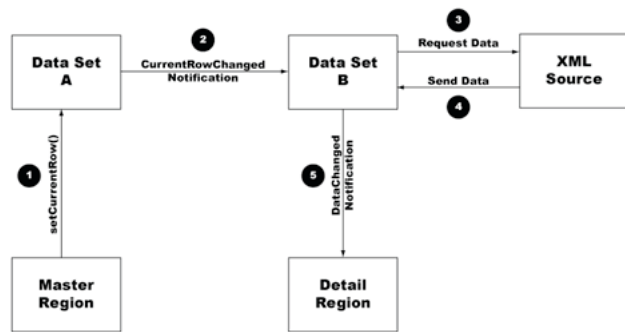
The URL for the second data set, `dsIngredients`, contains a data reference (`{dsSpecials::url}`) to the first data set, `dsSpecials`. More specifically, it contains a data reference to the url column in the `dsSpecials` data set. When the URL or XPath argument in the constructor that creates a data set contains a reference to another data set, the data set being created automatically becomes an observer of the data set it's referencing. The new data set depends on the original data set, and reloads its data or reapplies its XPath whenever the data or current row changes in the original data set.

The following example shows the observer relationships established between data sets and master and detail dynamic regions. In the preceding example, the `dsIngredients` data set (data set B) is an observer of the `dsSpe-cials` data set (data set A).



In the example, changing the current row in the `dsSpecials` data set sends a notification to the `dsIngredients` data set that it also needs to change. Because each row of the `dsSpecials` data set contains a distinct URL in the url column, the `dsIngredients` data set must update to include the correct URL for the selected row.

By default, the `dsIngredients` data set (whose data is displayed in the detail region) is created using the data it obtains from the URL specified in the constructor—in this case a reference to the data in the url column of the `dsSpecials` data set. The default current row in the dsSpecials data set (the first row) contains a unique path to the summersalad.xml file, and thus the detail region displays the information from that file when the page loads in a browser. When the current row of the `dsSpecials` data set changes, however, the URL also changes—to salmon.xml for example—and the `dsIngredients` data set (and by association, the detail dynamic region) updates accordingly.



This process is functionally equivalent to the one illustrated in “Spry basic master and detail region overview and structure” on page 101, the technical difference being that in the advanced case, the second (or detail) *data set* is listening for data and row changes in the master data set, whereas in the basic example, the *detail region* is listening for data and row changes in the master data set.

In the example code, `spry:region` is used for the detail region instead of `spry:detailregion`. The difference between a `spry:region` and a `spry:detailregion` is that the `spry:detailregion` specifically listens for `CurrentRowChange` notifications (in addition to `DataChanged` notifications) from the data set, and updates itself when it receives one. Because the current row of the `dsIngredients` data set never changes (it's the current row of the dsSpecials data set that changes), a `spry:detailregion` attribute is not needed. In this case, the `spry:region` attribute, which defines a region that only listens for DataChanged notifications, suffices.

About progressive enhancement and data set accessibility

Progressive enhancement writes code for a document for the lowest common denominator of browser functionality and then enhances the presentation and behavior of the page, using CSS, JavaScript, Flash, Java, and SVG code, and so on. Pages created with this approach provide an enhanced experience in modern browsers, but the data is still accessible and the page still functional in the absence of these technologies.

All of the Spry examples in this document up to this point have dealt with using JavaScript to dynamically load XML data and generate regions of the document. You can also use Spry in a progressive enhancement manner using the Hijax methodology (see http://domscripting.com/blog/display/41). This progressive enhancement methodology uses JavaScript to unobtrusively attach event handlers to items on the page, such as links to catch user events (for example, clicks). Progressive enhancement also lets you replace parts of your document with code fragments that are delivered asynchronously from the server to avoid having to load an entire page.

As a trivial example of using Spry with this methodology, you could start with an HTML page filled with static data and links:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:spry="http://ns.adobe.com/spry">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Hijax Demo - Notes 1</title>
</head>
<body>
<a href="notes1.html">Note 1</a>
<a href="notes2.html">Note 2</a>
<a href="notes3.html">Note 3</a>
<div>
    <p>This is some <b>static content</b> for note 1.</p>
</div>
</body>
</html>
```

To progressively enhance this page with Spry, so that you don't have to load an entirely new page when you click the links, you first use XML to make sure that the page fragments of each page that the links refer to are accessible. The following example is one way to externalize the static data:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<notes>
    <note><![CDATA[<p>This is some <b>dynamic content</b> for note 1.</p>]]></note>
    <note><![CDATA[<p>This is some <b>dynamic content</b> for note 2.</p>]]></note>
    <note><![CDATA[<p>This is some <b>dynamic content</b> for note 3.</p>]]></note>
</notes>
```

You can then apply Spry to the HTML page in the following manner:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:spry="http://ns.adobe.com/spry">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Hijax Demo - Notes 1</title>
<script language="JavaScript" type="text/javascript" src="includes/xpath.js"></script>
<script language="JavaScript" type="text/javascript" src="includes/SpryData.js"></script>
<script language="JavaScript" type="text/javascript">
<!--
var dsNotes = new Spry.Data.XMLDataSet('data/notes.xml', "/notes/note");
-->
</script>
</head>
<body>
<a href="note1.html" onclick="dsNotes.setCurrentRowNumber(0); return false;">Note 1</a>
<a href="note2.html" onclick="dsNotes.setCurrentRowNumber(1); return false;">Note 2</a>
<a href="note3.html" onclick="dsNotes.setCurrentRowNumber(2); return false;">Note 3</a>
<div spry:detailregion="dsNotes" spry:content="{note}">
    <p>This is some <b>static content</b> for note 1.</p>
</div>
</body>
</html>
```

The Spy code added in the preceding code looks familiar, but you'll notice that the `div` block that includes the `spry:detailregion` attribute also includes a `spry:content` attribute. This `spry:content` attribute tells the Spry dynamic-region processing code to replace the static data, currently contained in the region, with the data represented by the data reference in its attribute value, if any data is in the data set that the region is bound to.

If this page is loaded in a browser with JavaScript disabled, it degrades and you get the same functionality as for the original page with static content and traditional link navigation. If JavaScript is enabled, the data set loads the XML data and replaces the static content in the region. Clicking the links updates the region with code from the data set.

In the preceding example, Hijax promotes the use of unobtrusively attaching event handlers to links. The preceding example intentionally uses `onclick` attributes to quickly illustrate the point of attaching a JavaScript event handler.

Building dynamic pages with Spry

Prepare your files

Before you begin creating Spry data sets, obtain the necessary files (xpath.js and SpryData.js). The xpath.js file allows you to specify complex XPath expressions when creating your data set; the SpryData.js file contains the Spry data library.

Link both files to whatever HTML page you're creating.

1 Locate the Spry ZIP file on the Adobe Labs website.

2 Download and unzip the Spry ZIP file to your hard drive.

3 Open the unzipped Spry folder and locate the includes folder. This folder contains the xpath.js and SpryData.js files necessary for running the Spry framework.

4 Copy the includes folder and either paste or drag a copy of it to the root directory of your web site.

Note: If you drag the original includes folder out of the unzipped Spry folder, the demos in the Spry folder won't work properly.

5 In Code view (View > Code), link the Spry data library files to your web page by inserting the following `script` tags within the page's head tag:

```
<script type="text/javascript" src="includes/xpath.js"></script>
<script type="text/javascript" src="includes/SpryData.js"></script>
```

The SpryData.js file depends on the xpath.js file, so it's important that the xpath.js file comes first in the code.

When you've linked the Spry data library, you can create a Spry data set.

6 Add the Spry name-space declaration to the HTML tag so that the HTML tag looks as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:spry="http://ns.adobe.com/spry/">
```

The Spry name space declaration is necessary to validate the code.

Note: Spry features work locally as long as the Spry data library files are linked to your HTML page. To publish the HTML page to a live server, upload the xpath.js and SpryData.js files as dependent files.

Create a Spry XML data set

1 Open a new or existing HTML page.

2 Make sure that you've linked the Spry data library files to the page and declared the Spry namespace. For more information, see "Prepare your files" on page 110.

3 Locate the XML source for the data set.

For example, you might want to use an XML file called *cafetownsend.xml* located in a folder called *data* in the site's root folder:

data/cafetownsend.xml

You could also specify a URL to an XML file, as follows:

http://www.somesite.com/somefolder/cafetownsend.xml

Note: The URL you decide to use (whether absolute or relative) is subject to the browser's security model, which means that you can only load data from an XML source that is on the same server domain as the HTML page you're linking from. You can avoid this limitation by providing a cross-domain service script. For more information, consult your server administrator.

4 Because you'll need to specify the repeating XML node that supplies data to the data set, make sure you understand the structure of the XML before you create the data set.

In the following example, the cafetownsend.xml file consists of a parent node called *specials* that contains a repeating child node called *menu_item*.

```
<?xml version="1.0" encoding="UTF-8"?>
<specials>
    <menu_item id="1">
        <item>Summer Salad</item>
        <description>organic butter lettuce with apples, blood oranges, gorgonzola, and
raspberry vinaigrette.</description>
        <price>7</price>
    </menu_item>
    <menu_item id="2">
        <item>Thai Noodle Salad</item>
        <description>lightly sauteed in sesame oil with baby bok choi, portobello mushrooms,
and scallions.</description>
        <price>8</price>
    </menu_item>
    <menu_item id="3">
        <item>Grilled Pacific Salmon</item>
        <description>served with new potatoes, diced beets, Italian parlsey, and lemon
zest.</description>
        <price>16</price>
    </menu_item>
</specials>
```

5 To create the data set, insert the following `script` block after the `script` tags importing the library:

```
<script type="text/javascript">
    var datasetName = new Spry.Data.XMLDataSet("XMLsource", "XPathToRepeatingChildNode");
</script>
```

In the Cafe Townsend example, you create a data set with the following statement:

```
var dsSpecials = new Spry.Data.XMLDataSet("data/cafetownsend.xml", "specials/menu_item");
```

The statement creates a new data set called `dsSpecials` that retrieves data from the `specials/menu_item` node in the specified XML file. The data set has a row for each menu item and the following columns: @id, item, description, and price that the following table represents.

@id	item	description	price
1	Summer salad	organic butter lettuce with apples, blood oranges, gorgonzola, and raspberry vinaigrette.	7
2	Thai Noodle Salad	lightly sauteed in sesame oil with baby bok choi, portobello mushrooms, and scallions.	8
3	Grilled Pacific Salmon	served with new potatoes, diced beets, Italian parlsey, and lemon zest.	16

You can also specify a URL as the source of the XML data, as follows:

```
var dsSpecials = new
Spry.Data.XMLDataSet("http://www.somesite.com/somefolder/cafetownsend.xml",
"specials/menu_item");
```

Note: *The URL you decide to use (whether absolute or relative) is subject to the browser's security model, which means that you can only load data from an XML source that is on the same server domain as the HTML page you're linking from. You can avoid this by providing a cross-domain service script. For more information, consult your server administrator.*

The completed example code might look as follows:

```
<head>
...
<script type="text/javascript" src="includes/xpath.js"></script>
<script type="text/javascript" src="includes/SpryData.js"></script>
<script type="text/javascript">
    var dsSpecials = new Spry.Data.XMLDataSet("data/cafetownsend.xml",
"specials/menu_item");
</script>
...
</head>
```

6 (Optional) If the values in your data set include numbers (as in this example), reset the column types for the columns that contain those numerical values. This becomes important later if you want to sort data.

To set column types, add the setColumnType data set method to the head tag of your document, after you've created the data set, as follows (in bold):

```
<script type="text/javascript">
    var dsSpecials = new Spry.Data.XMLDataSet("data/cafetownsend.xml",
"specials/menu_item");
    dsSpecials.setColumnType("price", "number");
</script>
```

In the example, the expression calls the `setColumnType` method on the dsSpecials data set object, which you've already defined. The `setColumnType` method takes two parameters: the name of the data set column to retype (`"price"`) and the desired data type (`"number"`).

For more information, see "Let users sort data" on page 115.

After you've created the data set, create a dynamic region so that you can display the data.

Create a Spry dynamic region and display data

After you create a Spry data set, you bind a Spry dynamic region to the data set. A Spry dynamic region is an area on the page that displays the data and automatically updates the data display whenever the data set is modified.

1 Make sure that you've linked the Spry data library files to the page, declared the Spry namespace, and created a data set. For more information, see "Prepare your files" on page 110 and "Create a Spry XML data set" on page 111.

2 In Code view, create a Spry dynamic region by adding the `spry:region` attribute to the tag that will contain the region. The attribute uses the syntax `spry:region="`*`datasetName`*`"`.

Note: *Most, but not all, HTML elements can act as containers for dynamic regions. For more information, see "Spry dynamic region overview and structure" on page 98.*

For example, to use a `div` tag as the container for the dynamic region displaying data from the `dsSpecials` data set, add the `spry:region` attribute to the tag as follows:

```
<div id="Specials_DIV" spry:region="dsSpecials"></div>
```

Note: *Dynamic regions can depend on more than one data set. To add more data sets to the region, list them as additional values of the `spry:region` attribute, separated by a space. For example, you can create a dynamic region by using `spry:region="dsSpecials dsSpecials2 dsSpecials3"`.*

3 Within the tag containing the dynamic region (this example uses a `div` tag), insert an HTML element to display the first row of the data set. You can use any HTML element to display data. One of the most typical elements used for this purpose, however, is a two-row HTML table, where the first row contains static column headings and the second row contains the data:

```
<table id="Specials_Table">
    <tr>
        <th>Item</th>
        <th>Description</th>
        <th>Price</th>
    </tr>
    <tr spry:repeat="dsSpecials">
        <td>{item}</td>
        <td>{description}</td>
        <td>{price}</td>
    </tr>
</table>
```

The values in curly braces in the second row—the data references—specify columns in the data set. The data references bind the table cells to data in specific columns of the data set.

Note: *If the Spry region depends on more than one data set, specify the data set to which you're binding the dynamic region. The full syntax of the data reference takes the form of* `{datasetName::columnName}`*. For example, to bind the dynamic region to two or three different data sets, enter the data in the preceding example as follows:* `{dsSpecials::item}`*,* `{dsSpecials::description}`*, and so forth. Spry regions also support multiple data sets which you can specify by adding the data set names to the value of the attribute, separated by a space (i.e. <div spry:region="ds1 ds2 ds3">.*

4 Make the HTML element repeat automatically to display all the rows of the data set by adding the `spry:repeat` attribute and value to the HTML element tag using the following syntax:

```
spry:repeat="datasetName"
```

In the example, you add the `spry:repeat` attribute to the table row tag as follows (in bold):

```
<tr spry:repeat="dsSpecials">
    <td>{item}</td>
    <td>{description}</td>
    <td>{price}</td>
</tr>
```

In the example, the completed code binding the dynamic region to the data set would look as follows:

```
<div id="Specials_DIV" spry:region="dsSpecials">
    <table id="Specials_Table">
        <tr>
            <th>Item</th>
            <th>Description</th>
            <th>Price</th></tr>
        <tr spry:repeat="dsSpecials">
            <td>{item}</td>
            <td>{description}</td>
            <td>{price}</td>
        </tr>
    </table>
</div>
```

5 You can make the dynamic region more interactive by defining click events that allow users to sort data. For instructions, see "Let users sort data" on page 115.

Sample code: Spry data set and dynamic region

The following sample code creates a Spry data set and dynamic region to display a list of menu specials in an HTML table.

```
<head>
...
<script type="text/javascript" src="includes/xpath.js"></script>
<script type="text/javascript" src="includes/SpryData.js"></script>
<script type="text/javascript">
    var dsSpecials = new Spry.Data.XMLDataSet("data/cafetownsend.xml",
"specials/menu_item");
</script>
...
</head>
<body>
...
<div id="Specials_DIV" spry:region="dsSpecials">
    <table id="Specials_Table">
        <tr>
            <th>Item</th>
            <th>Description</th>
            <th>Price</th>
        </tr>
        <tr spry:repeat="dsSpecials">
            <td>{item}</td>
            <td>{description}</td>
            <td>{price}</td>
        </tr>
    </table>
</div>
...
</body>
```

Let users sort data

You can spry:sort attributes to your dynamic region that allow users to interact with the data. This section uses the table code from "Create a Spry dynamic region and display data" on page 113 as an example.

1 Locate the place in the code where you want to add the spry:sort attribute or attributes. In this example, the attributes are added to two column headers in a table that displays the XML data.

2 Add a spry:sort attribute to the appropriate column header tags, using the following form:

```
spry:sort="columnName"
```

The value defined in the spry:sort attribute tells the data set which column to use when sorting the data

For example, adding the following spry:sort attributes (in bold) to column header tags sorts the dynamic region data according to the specified value whenever the user clicks a column header on the page.

```
<div id="Specials_DIV" spry:region="dsSpecials">
    <table id="Specials_Table" class="main">
        <tr>
            <th spry:sort="item">Item</th>
            <th spry:sort="description">Description</th>
            <th>Price</th>
        </tr>
        <tr spry:repeat="dsSpecials">
            <td>{item}</td>
            <td>{description}</td>
            <td>{price}</td>
        </tr>
    </table>
</div>
```

Clicking Item on the page sorts the data alphabetically according to the menu item name, and clicking Description on the page sorts the data alphabetically according to the menu item's description.

Numerical sorting

By default, all data in the data set (including numbers) is considered text, and sorts alphabetically. To sort numerically (for example, to sort by price of menu item), you can use the `setColumnType` data set method to change the data type of the price column from text to numbers. The method takes the following form:

```
datasetName.setColumnType("columnName", "number");
```

Using the preceding example, you would add the `setColumnType` method to the head tag of your document, after you create the data set (in bold):

```
<script type="text/javascript">
    var dsSpecials = new Spry.Data.XMLDataSet("data/cafetownsend.xml",
"specials/menu_item");
    dsSpecials.setColumnType("price", "number");
</script>
```

The expression calls the `setColumnType` method on the `dsSpecials` data set object, which you've already defined. The `setColumnType` method takes two parameters: the name of the data set column to retype (`"price"`) and the desired data type (`"number"`).

You can now add the spry:sort attribute to the price column so that all three columns in the HTML table are sortable when the user clicks any of the table headers:

```
<div id="Specials_DIV" spry:region="dsSpecials">
    <table id="Specials_Table" class="main">
        <tr>
            <th spry:sort="item">Item</th>
            <th spry:sort="description">Description</th>
            <th spry:sort="price">Price</th>
        </tr>
        <tr spry:repeat="dsSpecials">
            <td>{item}</td>
            <td>{description}</td>
            <td>{price}</td>
        </tr>
    </table>
</div>
```

Create a basic master and detail page

When working with Spry data sets, you can create master and detail dynamic regions to display more detail about your data. One region on the page (the master), drives the display of the data in another region on the page (the detail).

For an overview of how basic master and detail dynamic regions work, see “Spry basic master and detail region overview and structure” on page 101.

1 Create a data set. See “Create a Spry XML data set” on page 111.

2 Create the master region by adding the `spry:region` attribute to the HTML element that will act as the container tag for the region. See “Create a Spry dynamic region and display data” on page 113.

In the following example, a master dynamic region displays repeated data from the `dsSpecials` data set:

```
<head>
. . .
<script type="text/javascript" src="../includes/xpath.js"></script>
<script type="text/javascript" src="../includes/SpryData.js"></script>
<script type="text/javascript">
    var dsSpecials = new Spry.Data.XMLDataSet("data/cafetownsend.xml",
"specials/menu_item");</script>
</head>
. . .
<body>
    <div id="Specials_DIV" spry:region="dsSpecials">
        <table id="Specials_Table">
            <tr>
                <th>Item</th>
                <th>Description</th>
                <th>Price</th>
            </tr>
            <tr spry:repeat="dsSpecials">
                <td>{item}</td>
                <td>{description}</td>
                <td>{price}</td>
            </tr>
        </table>
    </div>
</body>
```

3 Add an attribute that will allow users to change the current row in the data set. In the following example, a spry:setrow attribute (in bold) changes the current row in the data set whenever a user clicks a row in the master region table:

```
<tr spry:repeat="dsSpecials" spry:setrow="ds_Specials">
    <td>{item}</td>
    <td>{description}</td>
    <td>{price}</td>
</tr>
```

4 Create the detail dynamic region on the page by adding the `spry:detailregion` attribute to the tag that will contain the region. The attribute uses the following syntax: `spry:detailregion="`*`datasetName`*`"`.

In the following example, a `div` tag contains the detail dynamic region:

```
<div id="Specials_Detail_DIV" spry:detailregion="dsSpecials">
</div>
```

5 Within the tag containing the detail dynamic region, insert an HTML element to display the detail data from the current row of the data set.

In the example, an HTML table displays detail data from the `{ingredients}` column and `{calories}` column in the `dsSpecials` data set.

```
<div id="Specials_Detail_DIV" spry:detailregion="dsSpecials">
    <table id="Specials_Detail_Table">
        <tr>
            <th>Ingredients</th>
            <th>Calories</th>
        </tr>
        <tr>
            <td>{ingredients}</td>
            <td>{calories}</td>
        </tr>
    </table>
</div>
```

The completed example code binding both the master and detail dynamic regions to the dsSpecials data set would look as follows:

```
<div id="Specials_DIV" spry:region="dsSpecials">
    <table id="Specials_Table">
        <tr>
            <th>Item</th>
            <th>Description</th>
            <th>Price</th>
        </tr>
        <tr spry:repeat="dsSpecials" spry:setrow="dsSpecials">
            <td>{item}</td>
            <td>{description}</td>
            <td>{price}</td>
        </tr>
    </table>
</div>
<div id="Specials_Detail_DIV" spry:detailregion="dsSpecials">
    <table id="Specials_Detail_Table">
        <tr>
            <th>Ingredients</th>
            <th>Calories</th>
        </tr>
        <tr>
            <td>{ingredients}</td>
            <td>{calories}</td>
        </tr>
    </table>
</div>
```

Create an advanced master and detail page

You can create master and detail relationships that involve more than one data set. For an overview of how such relationships work, see “Spry advanced master and detail region overview and structure” on page 104.

1 Familiarize yourself with the structure of the XML files used in creating the data set. You’ll need to understand the structure to make one data set depend on another.

2 Create a data set by adding the appropriate code to the head of your document. (See “Create a Spry XML data set” on page 111.) This is the master data set.

3 Create a second data set (the detail data set) immediately following the master data set you just created. The URL or XPath in the constructor of the detail data set contains a data reference to one or more of the columns in the master data set. The data reference uses the following syntax: `{MasterDatasetName::columnName}`.

In the following example, the third `script` block contains the statement that creates two data sets, one called *dsSpecials* (the master) and one called *dsIngredients* (the detail):

```
<head>
. . .
<script type="text/javascript" src="../includes/xpath.js"></script>
<script type="text/javascript" src="../includes/SpryData.js"></script>
<script type="text/javascript">
        var dsSpecials = new Spry.Data.XMLDataSet("data/cafetownsend.xml",
"specials/menu_item");
        var dsIngredients = new Spry.Data.XMLDataSet("data/{dsSpecials::url}",
"item/ingredients/ingredient");
</script>
</head>
```

The path to the XML file for the dsIngredients detail data set contains a data reference (`{dsSpecials::url}`) to the `dsSpecials` master data set. More specifically, it contains a data reference to the url column in the `dsSpecials` data set. When the url or XPath argument in the constructor that creates a data set contains a reference to another data set, the data set being created automatically becomes an observer of the data set it’s referencing.

4 Create the master region by adding the `spry:region` attribute to the HTML element that will act as the container tag for the region. See “Create a Spry dynamic region and display data” on page 113.

In the following example, a master dynamic region displays repeated data from the `dsSpecials` data set:

```
<body>
    <div id="Specials_DIV" spry:region="dsSpecials">
        <table id="Specials_Table">
            <tr>
                <th>Item</th>
                <th>Description</th>
                <th>Price</th>
            </tr>
            <tr spry:repeat="dsSpecials">
                <td>{item}</td>
                <td>{description}</td>
                <td>{price}</td>
            </tr>
        </table>
    </div>
</body>
```

5 Add an attribute that will allow users to change the current row in the master data set. In the following example, a spry:setrow attribute (in bold) changes the current row in the `dsSpecials` data set whenever a user clicks a row in the master region table:

```
<tr spry:repeat="dsSpecials" spry:setrow="dsSpecials">
    <td>{item}</td>
    <td>{description}</td>
    <td>{price}</td>
</tr>
```

6 Create the detail dynamic region on the page by adding the `spry:region` attribute to the tag that will contain the region. The attribute uses the syntax `spry:region="datasetName"`.

***Note:** When creating master and detail relationships using two or more data sets, it is not necessary to use the `spry:detailregion` attribute as is outlined in "Create a basic master and detail page" on page 117. Because the current row of the detail data set never changes (it's the current row of the master data set that changes), the `spry:region` attribute suffices. For more information, see "Spry advanced master and detail region overview and structure" on page 104.*

In the following example, a `div` tag contains the detail dynamic region:

```
<div id="Specials_Detail_DIV" spry:region="dsSpecials">
</div>
```

7 Within the tag containing the detail dynamic region, insert an HTML element to display the detail data from the current row of the master data set.

In the example, an HTML table displays detail data from the `{name}` column in the `dsIngredients` data set. The `dsIngredients` data set creates the `{name}` column based on the information it receives from the `dsSpecials` data set.

```
<div id="Specials_Detail_DIV" spry:region="dsIngredients">
    <table id="Specials_Detail_Table">
        <tr>
            <th>Ingredients</th>
        </tr>
        <tr spry:repeat=”dsIngredients”>
            <td>{name}</td>
        </tr>
    </table>
</div>
```

The completed example code binding the master region to the `dsSpecials` data set, and detail region to the `dsIngredients` data set, would look as follows:

```
<head>
. . .
<script type="text/javascript" src="../includes/xpath.js"></script>
<script type="text/javascript" src="../includes/SpryData.js"></script>
<script type="text/javascript">
        var dsSpecials = new Spry.Data.XMLDataSet("data/cafetownsend.xml",
"specials/menu_item");
        var dsIngredients = new Spry.Data.XMLDataSet("data/{dsSpecials::url}",
"item/ingredients/ingredient");
</script>
</head>
. . .
<body>
    <div id="Specials_DIV" spry:region="dsSpecials">
        <table id="Specials_Table">
            <tr>
                <th>Item</th>
                <th>Description</th>
                <th>Price</th>
            </tr>
            <tr spry:repeat="dsSpecials" spry:setrow="dsSpecials">
                <td>{item}</td>
                <td>{description}</td>
                <td>{price}</td>
            </tr>
        </table>
    </div>
    <div id="Specials_Detail_DIV" spry:region="dsIngredients">
        <table id="Specials_Detail_Table">
            <tr>
                <th>Ingredients</th>
            </tr>
            <tr spry:repeat="dsIngredients">
                <td>{name}</td>
            </tr>
        </table>
    </div>
. . .
</body>
```

Getting and manipulating data

Data Set retrieval options

By default, Spry data sets use the HTTP GET method to retrieve XML data. The data set can also retrieve data using the HTTP POST method, by specifying additional constructor options, as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Dynamic Region Example</title>
<script type="text/javascript" src="../../includes/xpath.js"></script>
<script type="text/javascript" src="../../includes/SpryData.js"></script>
<script type="text/javascript">
    var dsPhotos = new Spry.Data.XMLDataSet("/photos.php", "/gallery/photos/photo", {
method: "POST", postData: "galleryid=2000&offset=20&limit=10", headers: { "Content-Type":
"application/x-www-form-urlencoded; charset=UTF-8" } });
</script>
</head>
. . .
<body>
</body>
</html>
```

If you use the POST method, but don't specify a content type, the default content type is set to `"application/x-www-form-urlencoded; charset=UTF-8"`.

The following table shows the HTTP-related constructor options that you can specify.

Option	Description
method	The HTTP method to use when fetching the XML data. Must be the string "GET" or "POST".
postData	Can be a string containing url encode form arguments or any value supported by the XMLHttpRequest object. If a "Content-Type" header is not specified in conjunction with the postData option, the `"application/x-www-form-urlencoded; charset=UTF-8"` content type is used.
username	The server username to use when accessing the XML data.
password	The password to use in conjunction with username when accessing the XML data.
headers	An object or associative array that uses the HTTP request field name as its property and key to store values.

Turn off data caching

By default, Spry caches any XML datathat a data set loaded on the client. If a data set attempts to load the XML data for a given URL that is already in the cache, Spry returns a reference to the cached data to the data set. If multiple data sets attempt to load the same URL at the same time, all load requests are combined into a single HTTP request to save bandwidth.

Use of data caching and of combining requests can greatly improve performance, especially when multiple data sets refer to the same XML data; at times you might need to load the data directly from the server (for example, if you have a URL that might return different data each time someone accesses it).

❖ To force a data set to load XML data directly from the server, set the `useCache` XMLDataSet constructor option to false, as follows:

```
var dsPhotos = new  Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo", {useCache: false })
```

Get data

After the XML data is loaded, it is flattened into a tabular format. Inside the data set, the data is actually stored as an array of objects (rows) with properties (columns) and values.

The following example shows data selected with the `/gallery/photos/photo` XPath, in bold:

```
<gallery id="12345">
    <photographer id="4532">John Doe</photographer>
    <email>john@doe.com</email>
    <photos id="2000">
        <photo path="sun.jpg" width="16" height="16"/>
        <photo path="tree.jpg" width="16" height="16"/>
        <photo path="surf.jpg" width="16" height="16"/>
    </photos>
</gallery>
```

The data set then flattens the set of nodes into a tabular format, that the following table represents.

@path	**@width**	**@height**
sun.jpg	16	16
tree.jpg	16	16
surf.jpg	16	16

You can get all of the rows in the data set by calling `getData()`. The data is loaded asynchronously, so you can only access the data after it is loaded. You might need to register an observer to be notified when the data is ready. For more information, see “Work with observer notifications” on page 126.

❖ Use the `getData()` method to fetch all the rows in the data set. To get the value of a specific column, index into the row with the column name.

```
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
...
var rows = dsPhotos.getData(); // Get all rows.
var path = rows[0]["@path"];   // Get the data in the "@path" column of the first row.
```

Sort data

❖ To sort the rows of a data set using the values in a given column, call the `sort()` method on the data set:

```
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
...
dsPhotos.sort("@path"); // Sort the rows of the data set using the "@path" column.
```

Calling the `sort()` method of a data set sorts the rows in ascending order, using data in the specified column.

The `sort()` method takes two arguments. The first argument can be the name of the column, or an array of column names to use when sorting. If the first argument is an array, the first element of the array serves as the primary sort column; each column after that is used for secondary and tertiary sorting, and so on. The second argument is the sort order to use, and must be one of the following strings: `"ascending"`, `"descending"`, or `"toggle"`. If the second argument is not specified, the sort order defaults to "ascending". Specifying `"toggle"` as the sort order causes the data sorted in `"ascending"` order to be sorted in `"descending"` order, and the reverse. If the column was never sorted before, and the sort order used is `"toggle"`, the data is initially sorted in `"ascending"` order.

```
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
...
dsPhotos.sort("@path", "toggle"); // Toggle the Sort order of the rows of the data set using
the "@path" column.
```

To have the data in the data set sorted automatically whenever data is loaded, specify the column to sort by and the sort order to use as an option to the data set constructor. Use the `"sortOnLoad"` option to the data set constructor to specify a column to sort by. In the following example, the data set automatically sorts in descending order by using the data in the @path column.

```
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo", { sortOnLoad: "@path", sortOrderOnLoad: "descending" });
```

By default, all values in the data set are treated as text. This can be problematic when sorting numeric or date values. You can specify the data type for a given column by calling the `setColumnType()` method.

```
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
dsPhotos.setColumnType("@width", "number");
dsPhotos.setColumnType("@height", "number");
...
dsPhotos.sort("@width"); // Sort the rows of the data set using the "@width" column.
```

Current supported data types are `"number"`, `"date"`, and `"string"`.

Set or change current row

Each data set maintains the notion of a *current row*. By default, the current row is set to the first row in the data set. To change the current row programmatically, call the `setCurrentRowNumber()` method and pass the row number of the row you want to make the current row. The index of the first row is always zero, so if a data set contains 10 rows, the index of the last row is 9.

❖ Use `setCurrentRowByNumber()` or `setCurrentRow()` to change the current row:

```
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
...
dsPhotos.setCurrentRowNumber(2); // Make the 3rd row in the data set the current row.
```

Each row in the data set is also assigned a unique ID that allows you to refer to a specific row in the data set even after the order of the rows in the data set change. You can get the ID of a row by accessing its `"ds_RowID"` property. You can also change the current row by calling `setCurrentRow()` and passing the row ID of the row to make the current row.

```
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
...
var id = dsPhotos.getData()[2]["ds_RowID"]; // Get the ID of the 3rd row.
...
dsPhotos.setCurrentRow(id); // Make the 3rd row the current row by using its ID.
```

Remove duplicate rows

❖ Use the `distinct()` method to remove duplicate rows in a data set:

```
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
...
dsPhotos.distinct(); // Remove all duplicate rows.
```

In this context, the term *duplicate row* applies to a situation in which every column in the data set contains identical information in 2 or more rows.

To run the `distinct()` method automatically whenever data is loaded into a data set, use the `"distinctOnLoad"` option to the constructor.

```
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo", { distinctOnLoad: true });
```

The `distinct()` method is destructive, so it discards any nondistinct rows. The only way to get the data back is to reload the XML data.

Filter data

Data sets support both destructive and non-destructive filtering.

Before using either method of filtering, supply a filter function that takes a data set, row object and rowNumber. This function is invoked by the data sets filtering methods for each row in the data set. The function must return either the row object passed to the function, or a new row object, meant to replace the row passed into the function. For the function to filter out the row, it should return a null value.

The data set's destructive filter method is `filterData()`. This method actually replaces or discards the rows of the data set. The only way to get the original data back is to reload the XML data of the data set.

❖ Use the destructive `filterData()` method to permanently discard rows in a data set:

```
...
// Filter out all rows that don't have a path that begins
// with the letter 's'.
var myFilterFunc = function(dataSet, row, rowNumber)
{
    if (row["@path"].search(/^s/) != -1)
        return row; // Return the row to keep it in the data set.
    return null; // Return null to remove the row from the data set.
}dsPhotos.filterData(myFilterFunc); // Filter the rows in the data set.
```

The filter function remains active, even when loading XML data from another URL, until you call `filterData()` with a `null` argument. Call `filterData()` with a `null` argument to uninstall your filter function.

```
dsPhotos.filterData(null); // Turn off destructive filtering.
```

The data set's nondestructive filter method is `filter()`. Unlike `filterData()`, `filter()` creates a new array of rows that reference the original data. As long as the filter function does not modify the row object passed into it, you can get the original data back by calling `filter()` and passing a `null` argument. Use the nondestructive `filter()` method to filter the rows in a data set.

```
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
...
// Filter out all rows that don't have a path that begins
// with the letter 's'.
var myFilterFunc = function(dataSet, row, rowNumber)
{
    if (row["@path"].search(/^s/) != -1)
        return row; // Return the row to keep it in the data set.
    return null; // Return null to remove the row from the data set.
}dsPhotos.filter(myFilterFunc); // Filter the rows in the data set.
```

To get the original data back, call `filter()` and pass a `null` argument

```
dsPhotos.filter(null); // Turn off non-destructive filtering.
```

Refresh data

Data sets can reload their data at a specified interval expressed in milliseconds. This can be handy when the data at a given URL changes periodically.

❖ To tell a data set to load at a given interval, pass the optional `loadInterval` option when calling the `XMLDataSet` constructor:

```
// Load the data every 10 seconds. Turn off the cache to  make sure we get it directly from
the server.
var dsPhotos = new  Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo", { useCache:  false, loadInterval: 10000 });
```

You can also turn on this interval loading programatically with the `startLoadInterval()` method, and stop it with the `stopLoadInterval()` method.

```
dsPhotos.startLoadInterval(10000); // Start loading data every 10 seconds.
...
dsPhotos.stopLoadInterval(); // Turn off interval loading.
```

Work with observer notifications

Observer notifications overview

The XML data set supports an observer mechanism that allows an object or callback function to receive event notifications.

Notification	Description
onPreLoad	The data set is about to send a request for data. If the data set depends on other data sets, this event notification will not send until they are all loaded successfully.
onPostLoad	The request for data was successful. The data is accessible.
onLoadError	An error occurred while requesting the data.
onDataChanged	The data in the data set has been modified.

Notification	Description
onPreSort	The data in the data set is about to be sorted.
onPostSort	The data in the data set has been sorted.
onCurrentRowChanged	The data set's notion of the current row has changed.

Objects as observers

To receive notifications, an object must define a method for each notification it is interested in receiving, and then register itself as an observer on the data set. For example, an object interested in `onDataChanged` notifications must define an `onDataChanged()` method and then call `addObserver()`.

```
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
...
var myObserver = new Object;
myObserver.onDataChanged = function(dataSet, data)
{
    alert("onDataChanged called!");
};
dsPhotos.addObserver(myObserver);
```

The first argument for each notification method is the object that is sending the notification. For data set observers, this is always the `dataSet` object. The second argument is either undefined, or an object that depends on the type of notification.

Notification	Data passed into notification
onPreLoad	undefined
onPostLoad	undefined
onLoadError	The Spry.Utils.loadURL.Request object that was used when making the request
onDataChanged	undefined
onPreSort	Object with the following properties:
	oldSortColumns: An array of columns used during the last sort.
	oldSortOrder: The sort order used during the last sort.
	newSortColumns: The array of columns about to be used for the sort.
	newSortOrder: The sort order about to be used for the sort.
onPostSort	Object with the following properties:
	oldSortColumns: An array of the columns used in the previous sort.
	oldSortOrder: The sort order used during the previous sort.
	newSortColumns: The array of columns used for the sort.
	newSortOrder: The sort order used for the sort.

Notification	Data passed into notification
onCurrentRowChanged	Object with the following properties:
	oldRowID: The ds_RowID of the last current row.
	newRowID: The ds_RowID of the current row.

To stop an object from receiving notifications, the object must be removed from the list of observers with a call to `removeObserver()`.

```
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
...
dsPhotos.removeObserver(myObserver);
```

Functions as observers

Functions can also be registered as observers. The main difference between object and function observers is that an object is only notified for the notification methods it defines, whereas a function observer is called for every type of event notification.

```
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
...
function myObserverFunc(notificationType, dataSet, data)
{
    if (notificationType == "onDataChanged")
        alert("onDataChanged called!";
    else if (notificationType == "onPostSort")
        alert("onPostSort called!";
};
dsPhotos.addObserver(myObserverFunc);
```

A function observer is registered with the same call to `addObserver`.

When the function is called, the first argument to be passed into it is the notification type. This is a string that is the name of the notification. The second argument is the notifier, which in this case is the data set, and the third argument is the data for the notification.

To stop a function observer from receiving notifications, it must be removed from the list of observers with a call to `removeObserver()`.

```
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
...
dsPhotos.removeObserver(myObserverFunc);
```

Working with dynamic regions

Looping constructs

Dynamic regions currently support two looping constructs. One allows you to repeat an element and all of its content for each row in a given data set (`spry:repeat`), and another allows you to repeat all of the children of a given element for each row in a given data set (`spry:repeatchildren`).

To designate an element as something that repeats, add a `spry:repeat` attribute to the element with a value that is the name of the data set to repeat. The following example shows an `li` block that repeats for every row in the `dsPhotos` data set

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:spry="http://ns.adobe.com/spry">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Dynamic Region Example</title>
<script type="text/javascript" src="../../includes/xpath.js"></script>
<script type="text/javascript" src="../../includes/SpryData.js"></script>
<script type="text/javascript">
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
</script>
</head>
<body>
    <div spry:region="dsPhotos">
        <ul>
            <li spry:repeat="dsPhotos">{@path}</li>
        </ul>
    </div>
</body>
</html>
```

To repeat just the children of an element, use the `spry:repeatchildren` attribute instead. In the following example, the children of the `ul` tag repeats for every row in the `dsPhotos` data set:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:spry="http://ns.adobe.com/spry">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Dynamic Region Example</title>
<script type="text/javascript" src="../../includes/xpath.js"></script>
<script type="text/javascript" src="../../includes/SpryData.js"></script>
<script type="text/javascript">
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
</script>
</head>
<body>
    <div spry:region="dsPhotos">
        <ul spry:repeatchildren="dsPhotos">
            <li>{@path}</li>
        </ul>
    </div>
</body>
</html>
```

The preceding `"spry:repeat"` and `"spry:repeatchildren"` examples are functionally equivalent, but `"spry:repeatchildren"` becomes more useful when used in conjunction with conditional constructs. Both examples result in the following output:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:spry="http://ns.adobe.com/spry">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Dynamic Region Example</title>
<script type="text/javascript" src="../../includes/xpath.js"></script>
<script type="text/javascript" src="../../includes/SpryData.js"></script>
<script type="text/javascript">
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
</script>
</head>
<body>
    <div>
        <ul>
            <li>sun.jpg</li>
            <li>tree.jpg</li>
            <li>surf.jpg</li>
        </ul>
    </div>
</body>
</html>
```

If you do not want to output the content in a repeat region for every row in the data set, limit what gets written out during the loop processing by adding a `spry:test` attribute to the element that has the `spry:repeat` or `spry:repeatchildren` attribute on it.

The following example shows an `li` block that is only output if the first letter of the value of `{@path}` starts with the letter *s*:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:spry="http://ns.adobe.com/spry">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Dynamic Region Example</title>
<script type="text/javascript" src="../../includes/xpath.js"></script>
<script type="text/javascript" src="../../includes/SpryData.js"></script>
<script type="text/javascript">
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
</script>
</head>
<body>
    <div spry:region="dsPhotos">
        <ul>
            <li spry:repeat="dsPhotos" spry:test="'{@path}'.search(/^s/) != -
1;">{@path}</li>
        </ul>
    </div>
</body>
</html>
```

The value of this `spry:test` attribute can be any JavaScript expression that evaluates to zero or `false` or some nonzero value. If the expression returns a nonzero value, the content is output. Because you are using XHTML, any special characters like &, <, or > that might be used in a JavaScript expression need to be converted to HTML entities. You can also use data references inside this JavaScript expression and the dynamic region processing engine provides the actual values from a data set just before evaluating the `spry:test` expression.

The following code shows the final output of the preceding example:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:spry="http://ns.adobe.com/spry">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Dynamic Region Example</title>
<script type="text/javascript" src="../../includes/xpath.js"></script>
<script type="text/javascript" src="../../includes/SpryData.js"></script>
<script type="text/javascript">
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
</script>
</head>
<body>
    <div>
        <ul>
            <li>sun.jpg</li>
            <li>surf.jpg</li>
        </ul>
    </div>
</body>
</html>
```

Conditional constructs

Dynamic regions currently support two conditional constructs. The first is `"spry:if"`. In the following example, the `li` tag is only written out if the value of `{@path}` begins with the letter *s*:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:spry="http://ns.adobe.com/spry">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Dynamic Region Example</title>
<script type="text/javascript" src="../../includes/xpath.js"></script>
<script type="text/javascript" src="../../includes/SpryData.js"></script>
<script type="text/javascript">
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
</script>
</head>
<body>
    <div spry:region="dsPhotos">
        <ul class="spry:repeat">
            <li spry:if="'{@path}'.search(/^s/) != -1;">{@path}</li>
        </ul>
    </div>
</body>
</html>
```

To make an element conditional, add an `spry:if` attribute to the element with a value that is a JavaScript expression that returns zero or nonzero values. A nonzero value that the JavaScript expression returns results in the element being written to the final output.

If you need an if-else construct, use the `"spry:choose"` construct. The following example uses the `spry:choose` construct to color every other `div`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:spry="http://ns.adobe.com/spry">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Dynamic Region Example</title>
<script type="text/javascript" src="../../includes/xpath.js"></script>
<script type="text/javascript" src="../../includes/SpryData.js"></script>
<script type="text/javascript">
var dsPhotos = new Spry.Data.XMLDataSet("/photos.php?galleryid=2000",
"/gallery/photos/photo");
</script>
</head>
<body>
    <div spry:region="dsPhotos">
        <div spry:choose="spry:choose">
            <div spry:when="'{@path}' == 'surf.gif'">{@path}</div>
            <div spry:when="'{@path}' == 'undefined'">Path was not defined.</div>
            <div spry:default="spry:default">Unexpected value for path!</div>
        </div>
    </div>
</body>
</html>
```

The `spry:choose` construct provides functionality equivalent to a case statement, or an if-else if-else construct. To create a `spry:choose` structure add a `spry:choose` attribute to an element. Next, add one or more child elements with `spry:when` attributes on them. The value of a `spry:when` attribute should be a JavaScript expression that returns a zero or nonzero value. To have a default case, in case all of the JavaScript expressions for each `spry:when` attribute return zero or `false`, add an element with a `spry:default` attribute. The `spry:default` attribute doesn't require a value, but XHTML states that all attributes must have a value, therefore, set the value of the attribute equal to its name.

The region processing engine evaluates the `spry:when` attribute of each node in the order they are listed under their parent element. The `spry:default` element is always evaluated last, and only if no `spry:when` expression returns a nonzero value.

Region states

Spry supports the notion of region states. That is, at any given time, a region is either loading data, ready to display the data, or in an error state because one or more of the data sets it is bound to failed to load its data. You can place a `spry:state` attribute, with a value of `"loading"`, `"error"`, or `"ready"` on elements *inside* a region container to associate it with a specific region state. Doing so can be quite useful for displaying a loading message as the data for a region loads, or notifying the user that the region failed to get its data. As the region changes states, it automatically regenerates its code and displays any elements with a `spry:state` attribute that matches the current state.

The following example uses the `spry:state` attribute to display loading and error messages:

```
<div spry:region="dsEmployees">
    <div spry:state="loading">Loading employee data ...</div>
    <div spry:state="error">Failed to load employee data!</div>
    <ul spry:state="ready">
        <li spry:repeat="dsEmployees">{firstname} {lastname}</li>
    </ul>
</div>
```

Any content that does not have a `spry:state` attribute on it, or is not a child or descendent of an element that has a `spry:state` attribute on it, is always included in the output when the code is regenerated. Also, children or descendents of an element with a `spry:state` attribute cannot have `spry:state` attributes. That is, nesting elements with `spry:state` attributes is not supported.

Region observer notifications

Spry supports an observer mechanism that allows a developer to register an object or function to receive a notification whenever the state of a region changes. This mechanism is almost identical to what is used for data sets with the following exceptions:

- Adding and removing region observers is done through the `Spry.Data.Region.addObserver()` and `Spry.Data.Region.removeObserver` global namespaced functions. This practice is different from data sets because data set observers call `addObserver()` and `removeObserver()` methods that are on the data set object. The use of global `namespaced` functions allows a developer to register an observer before the document's `onload` event starts, and before Spry creates the JavaScript object that represents the region. Regions use `addObserver` and `removeObserver` functions because the developer might want to register observers before the JavaScript region object actually exists.
- Both `addObserver()` and `removeObserver()` require an ID to identify which region the developer wants to observe. For this reason, regions that developers want to observe must have an `id` attribute defined on their region container node.

Objects as region observers

To receive notifications, an object must define a method for each notification it is interested in receiving, and then register itself as an observer on the region.

The following example shows an object being registered as an observer on a dynamic region:

```
<script>
...
// Create an observer object and define the methods to receive the notifications
// it wants.
myObserver = new Object;
myObserver.onPostUpdate = function(notifier, data)
{
    alert("onPostUpdate called for " + data.regionID);
};
...
// Call addObserver() to register the object as an observer.
Spry.Data.Region.addObserver("employeeListRegion", myObserver);
...
// You can unregister your object so it stops recieving notifications
// with a call to removeObserver().
Spry.Data.Region.removeObserver("employeeListRegion", myObserver);
...
</script>
...
<ul id="employeeListRegion" spry:region="dsEmployees">
...
</ul>
```

The first argument for each notification method is the object that is sending the notification. For region observers, this is not the region object. The second argument is an object that contains a `regionID` property that identifies the region that triggered the notification.

To stop an object from receiving notifications, the object must be removed from the list of observers with a call to `removeObserver()`.

Functions as region observers

Functions can also be registered as observers. The main difference between object and function observers is that an object is only notified for the notification methods it defines, whereas a function observer is called for every type of event notification.

The following example shows a function being registered as an observer on a dynamic region:

```
<script>
...
function myRegionCallback(notificationState, notifier, data)
{
    if (notificationType == "onPreUpdate")
        alert(regionID + " is starting an update!");
    else if (notificationType == "onPostUpdate")
        alert(regionID + " is done updating!");
}
...
// Call addObserver() to register your function as an observer.
Spry.Data.Region.addObserver("employeeListRegion", MyRegionCallback);
...
// You can unregister your callback function so it stops recieving notifications
// with a call to removeObserver().
Spry.Data.Region.removeObserver("employeeListRegion", MyRegionCallback);
...
</script>
...
<ul id="employeeListRegion" spry:region="dsEmployees">
...
</ul>
```

A function observer is registered with the same call to `addObserver()`.

When the function is called, the first argument passed into it is the notification type. This is a string that is the name of the notification. The second argument is the notifier, which in this case is not the region object. The third argument is a data object that has a `regionID` property that tells us what region triggered the notification.

To stop a function observer from receiving notifications, it must be removed from the list of observers with a call to `removeObserver()`.

The following table describes the current set of supported notifications.

Region notification type	Description
onLoadingData	One or more of the region's bound data sets is loading its data.
onPreUpdate	All of the region's bound data sets have loaded successfully. The region is about to regenerate its code.
onPostUpdate	The region has regenerated its code and inserted it into the document.
onError	An error occurred while loading data.

Data reference options

Each data set has a set of built-in data references that can be useful during the dynamic region regeneration process. Like data set column names, these built-in data references must be prefixed with the name of the data set if the dynamic region is bound to more than one data set.

For example, to display the row number of the current row of the data set when the region regenerates, add the `ds_RowNumber` data reference to the dynamic region, as follows:

```
<tr spry:repeat="dsSpecials">
    <td>{item}</td>
    <td>{description}</td>
    <td>{price}</td>
    <td>{ds_RowNumber}</td>
</tr>
```

These options are also useful to pass a value in a JavaScript method, as follows:

```
<tr spry:repeat="dsSpecials" onclick="dsSpecials.setCurrentRow('{ds_RowID}')">
    <td>{item}</td>
    <td>{description}</td>
    <td>{price}</td>
</tr>
```

The following table provides a full list of built-in Spry data references.

Data reference	Description
ds_RowID	The ID of a row in the data set. This ID can be used to refer to a specific record in the data set. It does not change, even when the data is sorted.
ds_RowNumber	The row number of the current row of the data set. Within a loop construct, this number reflects the position of the row currently being evaluated.
ds_RowNumberPlus1	The same as `ds_RowNumber`, except that the first row starts at index 1 instead of index 0.
ds_RowCount	The number of rows in the data set. If a nondestructive filter is set on the data set, this is the total number of rows after the filter is applied.
ds_UnfilteredRowCount	The number of rows in the data set before any nondestructive filter is applied.
ds_CurrentRowID	The ID of the current row of the data set. This value does not change, even when used within a looping construct.
ds_CurrentRowNumber	The row number of the current row of the data set. This value does not change, even when used within a looping construct.
ds_SortColumn	The name of the column last used for sorting. If the data in the data set was never sorted, this outputs nothing (an empty string).
ds_SortOrder	The current sort order of the data in the data set. This data reference outputs the words *ascending*, *descending*, or nothing (an empty string).
ds_EvenOddRow	Looks at the current value of `ds_RowNumber` and returns the string `"even"` or `"odd"`. Useful for rendering alternate row colors.

Hiding data references

In some browsers, loading pages over a slow connection can result in the user briefly seeing the unprocessed regions and data references on the page before the document's `onload` notification is sent. To hide unprocessed regions and data references, you can provide a CSS rule for the `SpryHiddenRegion` class:

```
<style>.SpryHiddenRegion {visibility:hidden;}
</style>
...
<div spry:region="dsEmployees" class="SpryHiddenRegion">
...
</div>
```

Using this technique, the CSS rule hides the Spry regions marked with this class when the page loads. When the Spry data is finished processing, Spry strips off the `SpryHiddenRegion` class and the finished Spry content appears.

An alternative way to hide just the data references, as opposed to the whole tag, is to use the `spry:content` attribute as a substitute for a data reference. Because the data reference is the value of the `spry:content` attribute, it is not visible when the page loads.

The following example hides a data reference with a `spry:content` attribute on an element:

```
<!--Example of a normal region.-->
<div spry:region="dsEmployees">
Hello my name is {firstname}.
</div>
<!--Example of a region using spry:content.-->
Hello my name is <span spry:content="{firstname}"></span>.
</div>
```

The `spry:content` attribute replaces the entire contents of the `div` tag with the value of the attribute. In this case, the the value of `{firstname}` is inserted into the empty `span` tag. The result is the same, only in this case there is no visible data reference.

Behavior attributes

You can place behavior attributes on elements within a dynamic region to automatically enable common behaviors that would ordinarily require some manual programming.

spry:hover

The `spry:hover` attribute places a class name on an element whenever the mouse cursor enters the element, and removes that class name as the cursor exits the element.

The value for the `spry:hover` attribute is the name of the class to put on the element whenever the mouse enters or exits the element.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:spry="http://ns.adobe.com/spry">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Behavior Attributes Example</title>
<style>
.myHoverClass { background-color: yellow; }
</style>
<script type="text/javascript" src="../../includes/xpath.js"></script>
<script type="text/javascript" src="../../includes/SpryData.js"></script><script
type="text/javascript">
var dsEmployees = new Spry.Data.XMLDataSet("../../data/employees-01.xml",
"/employees/employee");
</script>
</head>
<body>
<div spry:region="dsEmployees">
    <ul>
        <li spry:repeat="dsEmployees" spry:hover="myHoverClass">{username}</li>
    </ul>
</div>
</body>
</html>
```

In the preceding example, whenever the mouse enters an `li` element, the `"myHoverClass"` class name is added to the element's class attribute. It is automatically removed when the mouse exits the element.

spry:select

The `spry:select` attribute places a class name on an element when the mouse clicks the element.

The value for the `spry:select` attribute is the name of the class to put on the element whenever the mouse clicks the element.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:spry="http://ns.adobe.com/spry">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Behavior Attributes Example</title>
<style>
.myHoverClass {
    background-color: yellow;
}
.mySelectClass {color: white;background-color: black;}
</style>
<script type="text/javascript" src="../../includes/xpath.js"></script>
<script type="text/javascript" src="../../includes/SpryData.js"></script><script
type="text/javascript">var dsEmployees = new Spry.Data.XMLDataSet("../../data/employees-
01.xml",
"/employees/employee");
</script>
</head>
<body>
<div spry:region="dsEmployees">
    <ul>
        <li spry:repeat="dsEmployees" spry:hover="myHoverClass"
spry:select="mySelectClass">{username}</li>
    </ul>
</div>
</body>
</html>
```

In the preceding example, whenever the mouse clicks an `li` element, the `mySelectClass` class name is added to the elements class attribute.

If an element on the page with a `spry:select` attribute was previously selected, the class name used as the value for its `spry:select` attribute is automatically removed, in effect unselecting that element.

You can use a `spry:selectgroup` attribute in conjunction with a `spry:select` attribute to allow you to have more than one set of selections on a page. For a working example of this, see the RSS Reader example in the demos folder of the Spry folder from Adobe Labs.

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:spry="http://ns.adobe.com/spry">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Behavior Attributes Example</title>
<style>
.myHoverClass {
    background-color: yellow;
}
.mySelectClass {color: white;background-color: black;}
.myOtherSelectClass {color: white;background-color: black;}
</style>
<script type="text/javascript" src="../../includes/xpath.js"></script><script
type="text/javascript" src="../../includes/SpryData.js"></script><script
type="text/javascript">
var dsEmployees = new Spry.Data.XMLDataSet("../../data/employees-01.xml",
"/employees/employee");
</script></head><body><div spry:region="dsEmployees">
    <ul>
        <li spry:repeat="dsEmployees" spry:hover="myHoverClass" spry:select="mySelectClass"
spry:selectgroup="username">{username}</li></ul>
    <ul>
        <li spry:repeat="dsEmployees" spry:hover="myHoverClass"
spry:select="myOtherSelectClass" spry:selectgroup="firstname">{firstname}</li>
    </ul>
</div>
</body></html>
```

The value of a `spry:selectgroup` attribute is an arbitrary name. Any element that uses the same name for its `spry:selectgroup` attribute is automatically unselected when another element with the same select group name is clicked. Other elements with differing `spry:selectgroup` values are unaffected.

Chapter 4: Working with Spry Effects

About Spry effects

About Spry effects

Effects are visual enhancements that you can apply to almost any element on an HTML page. For example, an effect might highlight information, create animated transitions, or visually alter a page element for a certain period of time. Effects are a simple but elegant way of enhancing the look and feel of your website.

Effects refer to JavaScript methods and functions that reside on the client, and don't require any server-side logic or scripting to work. Thus, when a user browses an HTML page and triggers an effect, only the object to which the effect is applied gets updated: there is no need to refresh the entire page.

The Spry framework for AJAX includes these effects:

Fade Makes an element appear or fade away.

Highlight Changes the background color of an element.

Blind up/down Simulates a window blind that moves up or down to hide or reveal the element.

Slide up/down Moves the element up or down.

Grow Increases or reduces the size of the element.

Shake Simulates shaking the element from left to right.

Squish Makes the element disappear into the upper-left corner of the page.

Adobe designed Spry effects to be easy to implement on the page while letting the framework do the real work. No new tags or strange syntaxes are required. The HTML element to which you apply the effect does not require any custom tags.

Note: *Several effects libraries are available and Adobe recognizes the community's acceptance of Script.aculo.us as a well-written library. With that in mind, Adobe has adopted the their list of effects as well as their nomenclature and has implemented several Script.aculo.us solutions for browser issues related to effects. Adobe acknowledges the work that Thomas Fuchs of Script.aculo.us has done and hopes that the community recognizes both libraries as useful and viable. Additionally, Adobe has worked to make sure that developers can use both libraries on a single page.*

About the Spry effects library

The Spry Effects library, in the SpryEffects.js file, includes all of the Spry effects that are available on Adobe Labs. The file has no other dependencies.

Before you add effects to a page, you link the SpryEffects.js file in the head of the HTML document, as follows:

```
<script type="text/javascript" src="../includes/SpryEffects.js>"></script>
```

Note: *The exact file path differs, depending on where you store the SpryEffects.js file.*

Both the JavaScript file and the HTML file that contains the effects must reside on your server for Spry effects to work.

Before you begin

Prepare your files

Before you attach Spry effects to elements on your web pages, download and link the appropriate file.

1 Locate the Spry ZIP file on the Adobe Labs website.

2 Download and unzip the Spry ZIP file to your hard drive.

3 Open the unzipped Spry folder and locate the includes folder. This folder contains the file necessary for attaching Spry effects.

4 Add the SpryEffects.js file to your website by doing one of the following:

- Copy the includes folder and paste or drag a copy of it to the root directory of your web site. This gives you all of the files necessary for creating Spry effects as well as Spry XML data sets.
- To create a folder on your website (for example, a folder called *SpryAssets*), open the includes folder, and copy the SpryEffects.js file to the new folder.

Note: *If you drag the original includes folder or individual files out of the unzipped Spry folder, the demos in the Spry folder won't work properly. Copy and paste to your website instead of dragging.*

5 When the SpryEffects.js file is part of your website, you are ready to link it and add Spry effects to your pages. For specific instructions on adding a particular effect to a page, see the individual effects sections.

Attach Effects

Attach a Highlight effect

The Highlight effect changes the background color of a target element.

You can attach the Highlight effect to any HTML element except `applet`, `body`, `frame`, `frameset`, or `noframes`.

1 To link the SpryEffects.js file on your web page, add the following code to the head of your document:

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

Note: *The exact file path differs differ, depending on where you store the SpryEffects.js file.*

The SpryEffects.js file is in the includes folder of the Spry folder that you downloaded from Adobe Labs. See "Prepare your files" on page 142.

2 Make sure your target element has a unique ID. The target element is the element that changes when the user interacts with the page to cause the effect.

```
<div class="demoDiv"  id="highlight1"> Lorem ipsum dolor sit amet, consetetur sadipscing
elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed
diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum.</div>
```

3 To create the effect, add a JavaScript event that causes the effect when the user interacts with the page. For example, if you want the user to click on a sentence that causes another paragraph to highlight, you might add the following event to the sentence's `p` tag:

```
<p><a onclick="Spry.Effect.DoHighlight('highlight1',{duration: 1000, from:'#CCCCCC',
to:'#FFCC33',restoreColor: '#FFCC33',toggle:true}); return false;" href="#"> Click here to
highlight the below paragraph.</a></p>
```

The first argument of the JavaScript method is always the target element's ID (`'highlight1'` in the preceding example).

The complete code looks as follows:

```
<head>
. . .
<script  src="../includes/SpryEffects.js" type="text/javascript" ></script>
</head>
<body>
<p><a onclick="Spry.Effect.DoHighlight('highlight1',{duration: 1000, from:'#CCCCCC',
to:'#FFCC33',restoreColor: '#FFCC33',toggle:true}); return false;" href="#"> Click here to
highlight the below paragraph.</a></p>
<div class="demoDiv"  id="highlight1"> Lorem ipsum dolor sit amet, consetetur sadipscing
elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed
diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum.</div>
</body>
```

Highlight effect options

The following table lists the available options for the Highlight effect.

Option	Description
duration	The duration of the effect in milliseconds. The default value is 1000.
from	Start color value in RGB color format (#RRGGBB). This value sets the color of the first frame of the highlight. The default is the background color of the target element.
to	End color value in RGB color format (#RRGGBB). This value sets the color of the last frame of the highlight.
toggle	Produces a toggle effect. The default value is `false`. If the value is set to `true`, the `restoreColor` option is ignored.
transition	Determines the type of transition: linear (transition speed is constant for the duration of the transition) or sinusoidal (effect begins slowly, then speeds up, then slows again at the end). The default is sinusoidal.
setup	Lets you define a function that is called before the effect begins, e.g., `setup:function (element,effect){/* ... */}`.
finish	Lets you define a function that is called after the effect finishes, e.g., `finish:function (element,effect){/* ... */}`.

Sample code:

```
Spry.Effect.DoHighlight('targetID', {duration:1000,from:'#00ff00',to:'#0000ff'});
```

Attach a Fade effect

The Fade effect makes an element appear or fade away.

You can attach the Fade effect to any HTML element except `applet`, `body`, `iframe`, `object`, `tr`, `tbody`, or `th`.

1 To link the SpryEffects.js file on your web page, add the following code to the head of your document:

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

***Note:** The exact file path differs, depending on where you store the SpryEffects.js file.*

The SpryEffects.js file is in the includes folder of the Spry folder that you downloaded from Adobe Labs. See "Prepare your files" on page 142.

2 Make sure your target element has a unique ID. The target element is the element that changes when the user interacts with the page to cause the effect.

```
<div class="demoDiv" id="fade1">Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed
diam nonumy eirmod tempor invidunt ut labore et dolore magna sea takimata sanctus est Lorem
ipsum dolor sit amet.</div>
```

3 To create the effect, add a JavaScript event that causes the effect when the user interacts with the page. For example, if you want the user to click on a sentence that causes another paragraph to fade, you might add the following event to the sentence's `p` tag:

```
<p><a onclick="Spry.Effect.DoFade('fade1', {duration:1000,from:100,to:20,toggle:true});
return false;" href="#"> Click here to make the paragraph fade from 100% to 20%.</a></p>
```

The first argument of the JavaScript method is always the target element's ID (`'fade1'` in the preceding example).

The complete code looks as follows:

```
<head>
. . .
<script  src="../includes/SpryEffects.js" type="text/javascript" ></script>
</head>
<body>
<p><a onclick="Spry.Effect.DoFade('fade1', {duration:1000,from:100,to:20,toggle:true});
return false;" href="#"> Click here to make the paragraph fade from 100% to 20%.</a></p>
<div class="demoDiv" id="fade1">Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed
diam nonumy eirmod tempor invidunt ut labore et dolore magna sea takimata sanctus est Lorem
ipsum dolor sit amet.</div>
</body>
```

Fade effect options

The following table lists available options for the Fade effect.

Option	Description
duration	The duration of the effect in milliseconds. The default value is 1000.
from	Start opacity value in %. The default value is 0.
to	End opacity value in %. The default value is 100.
toggle	Produces a toggle effect. The default value is `false`.

Option	Description
transition	Determines the type of transition: linear (transition speed is constant for the duration of the transition) or sinusoidal (effect begins slowly, then speeds up, then slows again at the end). The default is sinusoidal.
setup	Lets you define a function that is called before the effect begins, e.g., `setup:function (element,effect){/* ... */}`.
finish	Lets you define a function that is called after the effect finishes, e.g., `finish:function (element,effect){/* ... */}`.

Sample code:

```
Spry.Effect.DoFade('targetID',{duration: 1000,from: 0,to: 100,toggle: true});
```

Attach a Blind up/Blind down effect

The Blind up/Blind down effect simulates a window blind that moves up or down to hide or reveal the element. This effect is similar to the Slide effect, but the contents stay in place.

You can only attach this effect to the following HTML elements: `address`, `dd`, `div`, `dl`, `dt`, `form`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `p`, `ol`, `ul`, `li`, `applet`, `center`, `dir`, `menu`, or `pre`.

1 To link the SpryEffects.js file on your web page, add the following code to the head of your document:

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

Note: *The exact file path differs, depending on where you store the SpryEffects.js file.*

The SpryEffects.js file is in the includes folder of the Spry folder that you downloaded from Adobe Labs. See “Prepare your files” on page 142.

2 Make sure your target element has a unique ID. The target element is the element that changes when the user interacts with the page to cause the effect.

```
<div id="blindup1">
    <h4>HEADER</h4>
    <p class="sampleText"> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed
    diam nonumy eirmod tempor invidunt ut labore et dolore magna</p>
</div>
```

3 To create the effect, add a JavaScript event that causes the effect when the user interacts with the page. For example, if you want the user to click on a sentence that causes another paragraph to blind up, you might add the following event to the sentence’s `p` tag:

```
<p><a onclick="Spry.Effect.DoBlind('blindup1', {duration: 1000, from:  '100%', to:  '0%',
toggle: true}); return false;" href="#" >Click here to blind up from 100% to 0%</a></p>
```

The first argument of the JavaScript method is always the target element’s ID (`'blindup1'` in the preceding example).

The complete code looks as follows:

```
<head>
. . .
<script  src="../includes/SpryEffects.js" type="text/javascript" > </script>
<style type="text/css">
#blindup1{
    background-color: #CCCCCC;
    height: 200px;
    width: 300px;
    overflow: hidden;
}
</style>
</head>
<body>
<p><a onclick="Spry.Effect.DoBlind('blindup1', {duration: 1000, from:  '100%', to:  '0%',
toggle: true}); return false;" href="#" >Click here to blind up from 100% to 0%</a></p>
<div id="blindup1">
    <h4>HEADER</h4>
    <p class="sampleText"> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed
    diam nonumy eirmod tempor invidunt ut labore et dolore magna</p>
</div>
</body>
```

Blind up/Blind down effect options

The following table lists available options for the Blind up/Blind down effect.

Option	Description
duration	The duration of the effect in milliseconds. The default value is 1000.
from	Start size in % or pixels. The default value is 100%.
to	End size in % or pixels. The default value is 0%.
toggle	Produces a toggle effect. The default value is false.
transition	Determines the type of transition: linear (transition speed is constant for the duration of the transition) or sinusoidal (effect begins slowly, then speeds up, then slows again at the end). The default is sinusoidal.
setup	Lets you define a function that is called before the effect begins, e.g., `setup:function (element,effect){/* ... */}`.
finish	Lets you define a function that is called after the effect finishes, e.g., `finish:function (element,effect){/* ... */}`.

Sample code:

```
Spry.Effect.DoBlind('targetID',{duration: 1000,from: '100%',to: '0%'});
```

Attach a Slide effect

The Slide effect moves the target element up or down (or left to right). This effect is similar to the Blind effect but the contents move up and down (or left to right) instead of staying in the same place.

You can only attach this effect to the following HTML elements: `blockquote`, `dd`, `div`, `form`, `center`, `table`, `span`, `input`, `textarea`, `select`, or `image`.

1 To link the SpryEffects.js file on your web page, add the following code to the head of your document:

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

Note: *The exact file path differs, depending on where you store the SpryEffects.js file.*

The SpryEffects.js file is in the includes folder of the Spry folder that you downloaded from Adobe Labs. See “Prepare your files” on page 142.

2 Make sure your target element is wrapped in a `div` that has a unique ID. The target element is the element that changes when the user interacts with the page to cause the effect.

```
<div id="slide1">
    <div> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed
    diam nonumy eirmod tempor invidunt ut labore et dolore magna
    aliquyam erat, sed diam voluptua.</div>
</div>
```

3 To create the effect, add a JavaScript event that causes the effect when the user interacts with the page. For example, if you want the user to click on a sentence that causes another paragraph to slide up, you might add the following event to the sentence’s `p` tag:

```
<p><a onclick="Spry.Effect.DoSlide('slide1',{duration:1000,from:'100%',
to:'20%',toggle:true}); return false;" href="#">Click here to slide the paragraph up from
100% to 20%</a></p>
```

The first argument of the JavaScript method is always the target element’s ID (`'slide1'` in the preceding example).

The complete code looks as follows:

```
<head>
. . .
<script  src="../includes/SpryEffects.js" type="text/javascript" > </script>
</head>
<body>
<p><a onclick="Spry.Effect.DoSlide('slide1',{duration:1000,from:'100%',
to:'20%',toggle:true}); return false;" href="#"> Click here to slide the paragraph up from
100% to 20%</a></p>
<div id="slide1">
    <div> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed
    diam nonumy eirmod tempor invidunt ut labore et dolore magna
    aliquyam erat, sed diam voluptua.</div>
</div>
</body>
```

Slide effect options

The following table lists available options for the Slide effect.

Option	Description
duration	The duration of the effect in milliseconds. The default value is 2000.
from	Start size in % or pixels. The default value is '100%'.
to	End size in % or pixels. The default value is '0%'.
toggle	Produces a toggle effect. The default value is false.

Option	Description
transition	Determines the type of transition: linear (transition speed is constant for the duration of the transition) or sinusoidal (effect begins slowly, then speeds up, then slows again at the end). The default is sinusoidal.
horizontal	If set to true, slides the content horizontally instead of vertically. The default value is false.
setup	Lets you define a function that is called before the effect begins, e.g., `setup:function (element,effect){/* ... */}`.
finish	Lets you define a function that is called after the effect finishes, e.g., `finish:function (element,effect){/* ... */}`.

Sample code:

```
Spry.Effect.DoSlide('targetID',{duration: 1000,from: '100%',to: '0%'});
```

Attach a Grow effect

The Grow effect increases or reduces the size of the element. The movement is toward or away from the center of the element.

You can only attach this effect to the following HTML objects: `address`, `dd`, `div`, `dl`, `dt`, `form`, `p`, `ol`, `ul`, `applet`, `center`, `dir`, `menu`, `img` or `pre`.

1 To link the SpryEffects.js file on your web page, add the following code to the head of your document:

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

Note: *The exact file path differs, depending on where you store the SpryEffects.js file.*

The SpryEffects.js file is in the includes folder of the Spry folder that you downloaded from Adobe Labs. See “Prepare your files” on page 142.

2 Make sure your target element has a unique ID. The target element is the element that changes when the user interacts with the page to cause the effect:

```
<div class="demoDiv" id="shrink1">
<p>Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor
invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam
et justo duo dolores et ea rebum.</p>
</div>
```

3 To create the effect, add a JavaScript event that causes the effect when the user interacts with the page. For example, if you want the user to click on a sentence that causes another paragraph to shrink, you might add the following event to the sentence’s `p` tag:

```
<p><a onclick="Spry.Effect.DoGrow('shrink1',{duration:700, from:'100%', to:'20%',toggle:
true}); return false;" href="#">Click here to shrink the paragraph from 100% to 20%.</a></p>
```

The first argument of the JavaScript method is always the target element’s ID (`'shrink1'` in the preceding example).

The complete code looks as follows:

```
<head>
. . .
<script  src="../includes/SpryEffects.js" type="text/javascript" ></script>
</head>
<body>
<p><a onclick="Spry.Effect.DoGrow('shrink1',{duration:700, from:'100%', to:'20%',toggle:
true}); return false;" href="#">Click here to shrink the paragraph from 100% to 20%.</a></p>
<div class="demoDiv" id="shrink1">
<p>Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor
invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam
et justo duo dolores et ea rebum.</p>
</div>
</body>
```

Grow effect options

The following table lists available options for the Grow effect.

Option	Description
duration	The duration of the effect in milliseconds. The default value is 500.
from	Start size in % or pixels. The default value is 0%.
to	End size in % or pixels. The default value is 100%.
toggle	Produces a toggle effect. The default value is false.
growCenter	Growing and shrinking direction of the element. The default value is `true` (grow and shrink from center). If set to `false`, the element grows or shrinks from the top left corner.
transition	Determines the type of transition: linear (transition speed is constant for the duration of the transition) or sinusoidal (effect begins slowly, then speeds up, then slows again at the end). The default is sinusoidal.
setup	Lets you define a function that is called before the effect begins, e.g., `setup:function (element,effect){/* ... */}`.
finish	Lets you define a function that is called after the effect finishes, e.g., `finish:function (element,effect){/* ... */}`.

Sample code:

```
Spry.Effect.DoGrow('targetID',{duration: 1000,from: '0%', to: '100%'});
```

Attach a Squish effect

The Squish effect makes the target element disappear into the upper-left corner of the page. The Squish effect produces the same effect as the Grow effect when the Grow effect’s growCenter option is set to false.

You can only attach this effect to the following HTML elements: `address`, `dd`, `div`, `dl`, `dt`, `form`, `img`, `p`, `ol`, `ul`, `applet`, `center`, `dir`, `menu`, or `pre`.

1 To link the SpryEffects.js file on your web page, add the following code to the head of your document:

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

Note: The exact file path differs, depending on where you store the SpryEffects.js file.

The SpryEffects.js file is in the includes folder of the Spry folder that you downloaded from Adobe Labs. See “Prepare your files” on page 142.

2 Make sure your target element has a unique ID. The target element is the element that changes when the user interacts with the page to cause the effect.

```
<div id="squish1"><p>Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy
eirmod tempor invidunt ut labore et dolore magna aliqt</p></div>
```

3 To create the effect, add a JavaScript event that causes the effect when the user interacts with the page. For example, if you want the user to click on a sentence that causes another paragraph to squish, you might add the following event to the sentence’s `p` tag:

```
<p><a onclick="Spry.Effect.DoSquish('squish1'); return false;" href="#">Click here to squish
the paragraph.</a></p>
```

The first argument of the JavaScript method is always the target element’s ID (`'squish1'` in the preceding example).

The complete code looks as follows:

```
<head>
. . .
<script  src="../includes/SpryEffects.js" type="text/javascript" ></script>
</head>
<body>
<p><a onclick="Spry.Effect.DoSquish('squish1'); return false;" href="#">Click here to squish
the paragraph.</a></p>
<div id="squish1"><p>Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy
eirmod tempor invidunt ut labore et dolore magna aliqt</p></div>
</body>
```

Squish effect options

The following table lists optional available options:

Option	Description
duration	The duration of the effect in milliseconds. The default value is 1000.
toggle	Produces a toggle effect. The default value is false.
setup	Lets you define a function that is called before the effect begins, e.g., `setup:function (element,effect){/* ... */}`.
finish	Lets you define a function that is called after the effect finishes, e.g., `finish:function (element,effect){/* ... */}`.

Sample code:

```
Spry.Effect.DoSquish('targetID',{duration: 1000});
```

Attach a Shake effect

The Shake effect simulates rapidly shaking the target element 20 pixels from left to right.

You can only attach this effect to the following HTML elements: `address`, `blockquote`, `dd`, `div`, `dl`, `dt`, `fieldset`, `form`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `iframe`, `img`, `object`, `p`, `ol`, `ul`, `li`, `applet`, `dir`, `hr`, `menu`, `pre`, or `table`.

1 To link the SpryEffects.js file on your web page, add the following code to the head of your document:

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

Note: *The exact file path differs, depending on where you store the SpryEffects.js file.*

The SpryEffects.js file is in the includes folder of the Spry folder that you downloaded from Adobe Labs. See “Prepare your files” on page 142.

2 Make sure your target element has a unique ID. The target element is the element that changes when the user interacts with the page to cause the effect.

```
<div id="shake1">Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy
eirmod tempor invidunt ut labore et dolore magna aliqit amet.</div>
```

3 To create the effect, add a JavaScript event that causes the effect when the user interacts with the page. For example, if you want the user to click on a sentence that causes another paragraph to shake, you might add the following event to the sentence’s `p` tag:

```
<p><a onclick="Spry.Effect.DoShake('shake1'); return false;" href="#">Shake it!</a></p>
```

The first argument of the JavaScript method is always the target element’s ID (`'shake1'` in the preceding example).

The complete code looks as follows:

```
<head>
. . .
<script  src="../includes/SpryEffects.js" type="text/javascript" ></script>
</head>
<body>
<p><a onclick="Spry.Effect.DoShake('shake1'); return false;" href="#">Shake it!</a></p>
<div id="shake1">Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy
eirmod tempor invidunt ut labore et dolore magna aliqit amet.</div>
</body>
```

Shake effect options

The following table lists available options for the Shake effect.

Option	**Description**
duration	Fixed at 500 milliseconds in Spry 1.4. Only editable in Spry 1.5 and later.
setup	Lets you define a function that is called before the effect begins, e.g., `setup:function (element,effect){/* ... */}`.
finish	Lets you define a function that is called after the effect finishes, e.g., `finish:function (element,effect){/* ... */}`.

Sample code:

```
Spry.Effect.DoShake('targetID');
```

Index

A

accessibility 2, 109
Accordion widget
  about 4
  adding panels to 10
  customizing 11
  deleting panels from 10
  enabling keyboard navigation 10
  inserting 7
  setting default open panel 11

B

behavior attributes 137
Blind up/Blind down effect 145

C

Collapsible Panel widget
  about 14
  customizing 20
  enabling keyboard navigation 19
  inserting 17
  setting default open panel 19

D

data
  current row, setting and changing 124
  filtering 125
  getting 123
  observer notifications 126
  references, hiding 136
  refreshing 126
  removing duplicate rows 124
  retrieving 121
  sorting 123
  turning off caching 122
Dreamweaver CS3 1
dynamic regions
  and conditional constructs 131
  and looping constructs 128
  and region states 133
  creating 113
  displaying data in 113
  observer notifications 133
  overview 98

E

effects
  about 141
effects library
  about 141

F

Fade effect 143

G

Grow effect 148

H

Highlight effect 142

J

JavaScript degradation 2

M

master and detail regions
  creating 117, 118
  depending on more than one data set 104
  overview 101
Menu Bar widget
  about 33
  changing orientation of 41
  customizing 42
  inserting 37

S

Shake effect 150
Slide effect 146
Spry
  and Dreamweaver CS3 1
  framework, about 1
  widgets, about 2
Squish effect 149

T

Tabbed Panels widget
  about 23
  adding panels to 29
  customizing 31
  deleting panels from 30
  enabling keyboard navigation 30
  inserting 27
  setting default open panel 30

V

Validation Checkbox widget
  about 83
  changing required status 89
  customizing 89, 90
  inserting 86
  specifying minimum and maximum number of selections 89
  specifying validation occurrence 89
Validation Select widget
  about 75
  changing required status 81
  customizing 82
  inserting 78
  specifying invalid values 82
  specifying validation occurrence 81
Validation Text Area widget
  about 65
  adding character counter to 71
  blocking extra characters 73
  changing required status 73
  creating hints 73
  customizing 73
  inserting 68
  specifying validation occurrence 71
Validation Text Field widget
  about 44
  blocking invalid characters 62
  changing required status 61
  creating hints 61
  customizing 62
  inserting 47
  specifying minimum and maximum number characters 60
  specifying minimum and maximum values 61

specifying validation occurrence 60
specifying validation type and format 50

X

XML Data Sets 91
advanced examples 93
creating 111
letting user sort 115
overview 91