TEMPLATES

Topics in this Chapter

- Encapsulating Layout
- Optional Content
- Role-based Content
- Defining Regions Separately
- Nesting Regions
- Extending Regions
- Combining Features
- Region Tag Implementations
- Conclusion

Chapter 4

Window toolkits typically provide three types of objects that greatly facilitate the implementation of flexible, extensible, and reusable applications: components, containers, and layout managers. Components are graphic objects such as buttons, menus, or lists, and containers are groups of components. Layout managers position and size a container's components.[1]

Components, containers, and layout managers are typically implemented with two design patterns: Composite and Strategy.[2] The Composite design pattern, used to implement components and containers, specifies that a container is a component, so you can place any component—even if it's a container—within any container. That handy feature lets you nest components as deeply as you want in a tree structure.

Layout managers are implemented with the Strategy pattern, which defines a family of algorithms and encapsulates each one. That makes layout algorithms interchangeable. Encapsulating layout also lets you modify layout algorithms without changing the containers that use them.

1. See Geary. *Graphic Java Volume 1: AWT*, Prentice Hall, 1998.
2. See Gamma, Helms, Johnson, Vlissides. *Design Patterns*, Addison-Wesley, 1994.

JSP does not provide anything analogous to components, containers, or layout managers. But JSP has two features—custom tags and the ability to include web components—that let you implement your own components, containers, and layout managers.[3] This chapter shows you how to do that so you can create web applications that are easy to maintain, extend, and reuse.

The components, containers, and layout managers discussed in this chapter are a little different from their window toolkit counterparts. To reflect that difference, they have different names:

- *Section*: An object that renders HTML or JSP in a page context
- *Region*: An object that contains sections
- *Template*: A JSP page that defines how regions and sections are laid out

Regions and templates are similar to containers and layout managers, respectively. Sections are different from components because they do not handle events; sections, however, are similar to components because they render content. That content is either an HTML file or a JSP page.

This chapter is divided into two parts. The first part, which begins with "Encapsulating Layout" below, shows you how to use a custom tag library to implement JSP pages with sections, regions, and templates. The second section, which starts at "Region Tag Implementations" on page 116, shows how those custom tags and their associated beans are implemented.

Note: The techniques discussed in this chapter represent an implementation of the J2EE Composite View pattern, which lets you compose a single view from multiple sub-views, using beans and JSP custom tags. You can read more about that design pattern and others in Core J2EE Patterns by Alur, Crupi, and Malks, published by Prentice Hall and Sun Microsystems Press.

Encapsulating Layout

Because layout typically undergoes many changes over the course of development, it's important to encapsulate that functionality so it can be modified with minimal impact to the rest of the application. In fact, layout managers are an example of one of the tenets of object-oriented design: *encapsulate the concept that varies*, which is also a fundamental theme for many design patterns.

3. The term *web component* refers to a servlet, HTML file, or JSP page and is unrelated to the more general term *component* used in this chapter.

Most web pages contain multiple sections that display their own content; for example, Figure 4-1 shows a web page containing header, footer, sidebar, and main content sections.

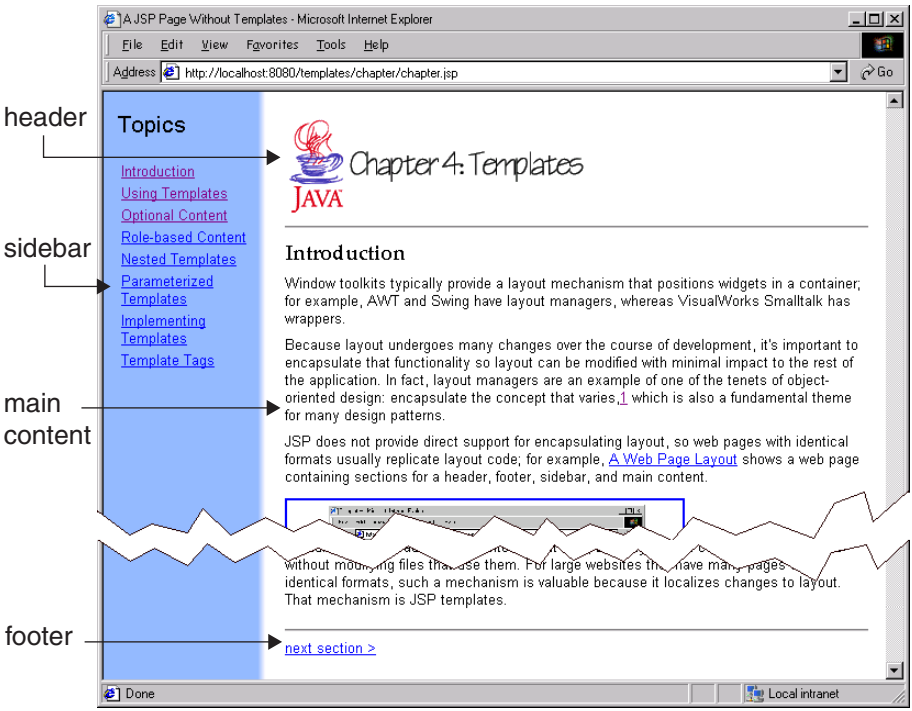

Figure 4-1 Web Page Layout

The layout of the page shown in Figure 4-1 can be implemented with HTML `table` tags, as listed in Example 4-1.

Example 4-1 Including Content

```
<html><head>
   <title>A JSP Page Without Templates</title>
</head>

<body background='graphics/blueAndWhiteBackground.gif'>

<table>
  <tr valign='top'>
    <td>
      <jsp:include page='sidebar.jsp' flush='true'/>
    </td>
```

```
    <td>
      <table>
        <tr>
          <td>
            <jsp:include page='header.jsp' flush='true'/>
          </td>
        </tr>
        <tr>
          <td>
            <jsp:include page='introduction.jsp' flush='true'/>
          </td>
        </tr>
        <tr>
          <td>
            <jsp:include page='footer.jsp' flush='true'/>
          </td>
        </tr>
      </table>
    </td>
  </tr>
</table>

</body>
</html>
```

In the JSP page listed in Example 4-1, content is included with the `jsp:include` action, which allows us to vary the content of that page—by changing the included files—without modifying the page itself. But because layout is hardcoded in the page, layout changes will require modifications to that page. If a web site has multiple pages with identical formats, even simple layout changes will require modifications to all of those pages.

Besides *separating content* from the JSP pages that display it, as is the case for the JSP page listed in Example 4-1, we can also *separate layout* from JSP pages. That separation means we can change layout without modifying the JSP files that use it.

Sections, Regions, and Templates

To separate layout from JSP pages, we will use one of the oldest programming tricks in the book: indirection. We will split the single JSP page listed in Example 4-1—which includes content and performs layout—into two JSP pages: a region that defines content and a template that performs layout. That region is listed in Example 4-2.a.

Example 4-2.a A JSP Page That Defines a Region

```
<%@ taglib uri='regions' prefix='region' %>

<region:render template='/template.jsp'>
  <region:put section='title'  content='Templates' direct='true'/>
  <region:put section='header' content='/header.jsp' />
  <region:put section='sidebar'content='/sidebar.jsp' />
  <region:put section='content'content='/introduction.jsp'/>
  <region:put section='footer' content='/footer.jsp' />
</region:render>
```

The JSP page listed above uses custom tags to define and render a region. That region contains the four sections shown in Example 4-1 on page 99.

Every region is associated with a single template, which is specified with the `template` attribute of the `region:render` tag. The `region:render` start tag creates a region—see "The Beans" on page 116 for more about how regions are implemented—and places it in application scope.

The `region:put` tags store a name/value pair in the region created by the `region:render` start tag. Those name/value pairs represent section names and content; for example, for the region listed in Example 4-2.a, a `header` section is defined whose content is `/header.jsp`.

Finally, the `region:render` end tag includes the template defined by that tag's `template` attribute. The template included by the region defined in Example 4-2.a is listed in Example 4-2.b.

Example 4-2.b The Template Used by the Region Defined in Example 4-2.a

```
<%@ taglib uri='regions' prefix='region' %>

<html><head>
   <title><region:render section='title'/></title>
</head>

<body background='graphics/blueAndWhiteBackground.gif'>

<table>
  <tr valign='top'>
    <td>
      <region:render section='sidebar'/>
    </td>
    <td>
      <table>
        <tr>
```

```
          <td>
            <region:render section='header'/>
          </td>
        </tr>
        <tr>
          <td>
            <region:render section='content'/>
          </td>
        </tr>
        <tr>
          <td>
            <region:render section='footer'/>
          </td>
        </tr>
      </table>
    </td>
  </tr>
</table>

</body>
</html>
```

Like all templates, the template listed in Example 4-2.b uses the `region:render` tag to render a region's sections. That tag accesses the region, stored in application scope, that included the template. From that region, the `region:render` tag obtains the name of the content associated with a section and includes it.[4]

A `direct` attribute can be specified for `region:render`; if that attribute is set to `true`, the content associated with that tag is not included by `region:render` but is printed directly to the implicit `out` variable. For example, in Example 4-2.a the `title` content—"Templates"—is used as the window title.

Web sites containing multiple pages with identical formats have one template, such as the one listed in Example 4-2.b, and many JSP pages, such as Example 4-2.a, that use the template. If the format is modified, *changes are restricted to the template*.

Another benefit of templates, and included content in general, is modular design. For example, the JSP file listed in Example 4-2.a ultimately includes `/header.jsp`, which is listed in Example 4-2.c.

4. The `region:render` tag pulls double duty by rendering regions and sections.

Example 4-2.c /header.jsp

```
<table>
  <tr>
     <td><img src='graphics/java.gif'/></td>
     <td><img src='graphics/templates.gif'/></td>
  </tr>
</table>
<hr>
```

Because `/header.jsp` is included content, it does not have to be replicated among pages that display a header. Also, notice that `/header.jsp` does not contain the usual preamble of HTML tags, such as `<html>`, `<head>`, `<body>`, etc., that most JSP pages contain; those tags are supplied by the template that includes that JSP file. The `/header.jsp` file is simple and easy to maintain.

This section has illustrated the basic capabilities of two custom tags—`region:render` and `region:put`—from the regions custom tag library. That tag library provides other features, such as optional and role-based content, that are explored in the sections that follow.

JSP Tip

Use Templates to Implement Web Applications with Modular Components

The techniques discussed in this chapter—using sections, regions, and templates to implement web pages—lets you construct web applications with modular components.

Encapsulating content lets you modify that content without modifying JSP pages that display it. Similarly, encapsulating layout lets you modify the layout used by multiple JSP pages by changing a single template.

Optional Content

All content rendered by a template is optional, which makes a single template useful to many regions; for example, Figure 4-2 shows two regions that use the same template. The region on the left specifies content for all four of its sections: sidebar, header, content, and footer. The region on the right only specifies content for three of its regions: sidebar, header, and footer. As you can see from Figure 4-2, if a template cannot locate content for a given section, it merely ignores that section.

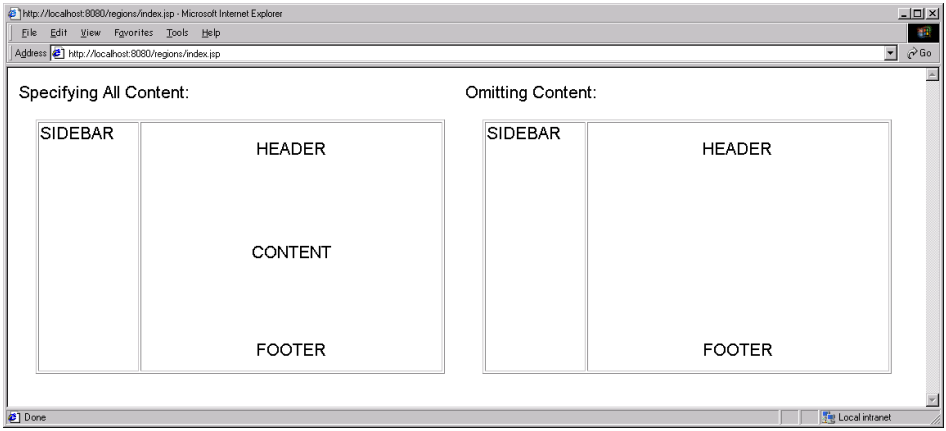

Figure 4-2 Optional Content

The JSP page shown in Figure 4-2 is listed in Example 4-3.a.

Example 4-3.a Specifying a Region with Inline Content and Omitted Content

```
<%@ taglib uri='regions' prefix='region' %>

<table>
   <tr>
      <td valign='top'>
         <font size='5'>Specifying All Content:</font>
         <table cellspacing='20'>
            <tr>
               <td>
                  <%--all content is specified for this region--%>
                  <region:render template='hscf.jsp'>
                     <region:put section='header'
                                 content='/header.jsp'/>

                     <region:put section='sidebar'
                                 content='/sidebar.jsp'/>

                     <region:put section='content'
                                 content='/content.jsp'/>

                     <region:put section='footer'
                                 content='/footer.jsp'/>
                  </region:render>
               </td>
         </table>
      </td>
```

```
        <td valign='top'>
           <font size='5'>Omitting Content:</font>
           <table cellspacing='20'>
              <tr>
                 <td>
                    <%--content is omitted for this region--%>
                    <region:render template='hscf.jsp'>
                       <region:put section='header'
                                   content='/header.jsp'/>

                       <region:put section='sidebar'
                                   content='/sidebar.jsp'/>

                       <region:put section='footer'
                                   content='/footer.jsp'/>
                    </region:render>
                 </td>
              </tr>
           </table>
           </td>
        </tr>
  </table>
```

In the JSP page listed above, the JSP pages specified as section content are very simple; for example, `/sidebar.jsp` is listed below:

```
<font size='5'>SIDEBAR</font>
```

The other JSP pages specified as content in Example 4-3.a are identical to `/sidebar.jsp`, except for the text they display; for example, `/header.jsp` is:

```
<font size='5'>HEADER</font>
```

In a real application, the JSP pages specified as section content would display more meaningful content, but the simple JSP pages used in Example 4-3.a serve to illustrate the use of regions and templates, so they are used throughout this chapter.

The template used by the regions listed in Example 4-3.a—`/hscf.jsp`—is listed in Example 4-3.b.[5]

5. `hscf` stands for **h**eader, **s**idebar, **c**ontent, and **f**ooter.

Example 4-3.b /hscf.jsp: The Template Used in Example 4-3.a

```
<html><head>
  <%@ taglib uri='regions' prefix='region' %>
</head>

<table border='1' width='500'>
   <tr> <%-- Sidebar --%>
      <td valign='top' width='25%'>
         <region:render section='sidebar'/>
      </td>
      <td valign='top' align='center' width='*'>
         <table height='300'>
            <tr> <%-- Header --%>
               <td align='center' height='20%'>
                  <region:render section='header'/>
               </td>
            </tr> <%-- Main Content --%>
               <td align='center' height='*'>
                  <region:render section='content'/>
               </td>
            </tr> <%-- Footer --%>
               <td align='center' height='15%'>
                  <region:render section='footer'/>
               </td>
            </tr>
         </table>
      </td>
   </tr>
</table>

</body></html>
```

The template listed in Example 4-3.b, like the template listed in Example 4-2.b on page 101, uses HTML table tags to lay out its four sections.

For the sake of illustration, the template listed above specifies a border width of 1 for its outermost table. Normally, templates should not render anything except their sections.

Role-based Content

Web applications often discriminate content based on a user's role. For example, the two pages shown in Figure 4-3 are produced by the same JSP template, which includes the edit panel only if the user's role is `curator`.

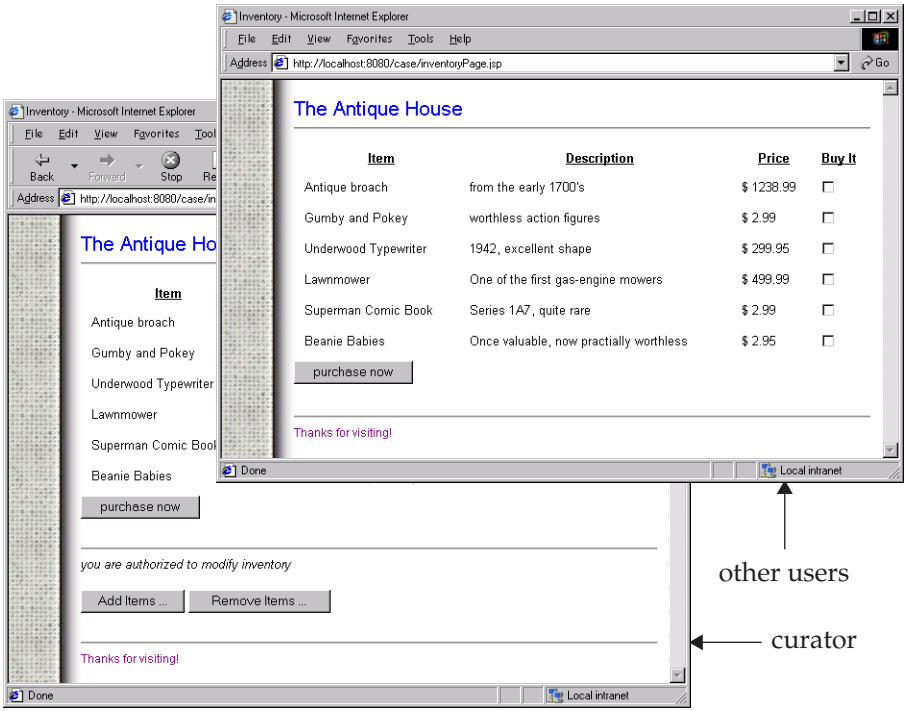

Figure 4-3 Role-based Content

Both of the JSP pages shown in Figure 4-3 use the same template, which is partially listed below.

```
<%@ taglib uri='regions.tld' prefix='region' %>
...
<table>
   ...
   <td><region:render section='editPanel' role='curator'/></td>
   ...
</table>
...
```

The `region:render` tag only renders a section's content if the user's role matches the `role` attribute. That `role` attribute is optional; if a role is not specified for a given section, that section, if it has content, is always rendered.

In the preceding code fragment, the template only renders the `editPanel` section if the user's role is `curator`. That means that every region that uses that template must abide by that restriction. Sometimes, you want individual regions to specify role-based content; to facilitate that intent, the `region:put` tag also has an optional role attribute. That attribute is used like this:

```
<region:render template='hscf.jsp'>
   ...
   <region:put section='header' content='/header.jsp'
               role='curator'/>
   ...
</region:render>
```

The `region:put` tag used in the code fragment above will only add the `header` section to the region created by the `region:render` start tag if the user's role is curator.

Defining Regions Separately

So far, all of the regions in this chapter have been defined and rendered in one place; for example, consider the JSP page shown in Figure 4-4.

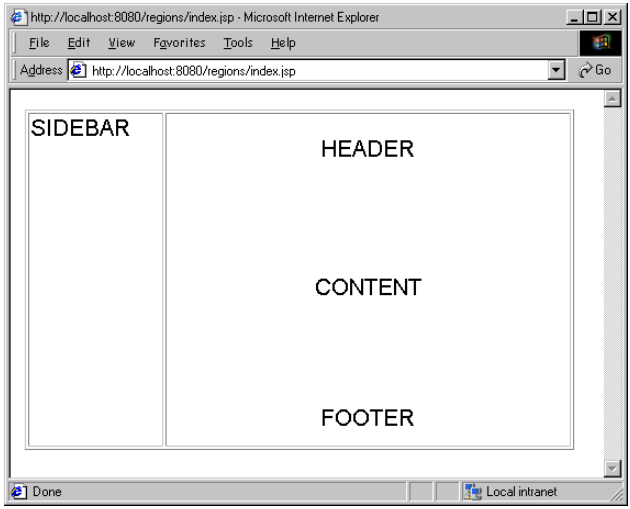

Figure 4-4 Defining Regions

The JSP page shown in Figure 4-4 can be created with a single region that has four sections. That region can be defined and rendered all at once, as listed in Example 4-4.

Example 4-4 Defining and Rendering a Region in One Place

```
<%@ taglib uri='regions' prefix='region' %>

<table>
   <tr>
      <td>
         <%--content for this region is specified inline--%>
         <region:render template='hscf.jsp'>
            <region:put section='header'  content='/header.jsp'/>
            <region:put section='sidebar' content='/sidebar.jsp'/>
            <region:put section='content' content='/content.jsp'/>
            <region:put section='footer'  content='/footer.jsp'/>
         </region:render>
      </td>
   </tr>
</table>
```

Instead of defining regions inline, as is the case for the region listed in Example 4-4, regions can be defined someplace other than where they are rendered; for example, Example 4-5.a shows how the JSP page shown in Figure 4-4 can be rendered from an existing region.

Example 4-5.a A JSP File That Uses an Existing Region

```
<%@ taglib uri='regions' prefix='region' %>

<%@ include file='/regionDefinitions.jsp' %>

<region:render region='SIDEBAR_REGION'/>
```

The JSP page listed in Example 4-5.a renders an existing region. That region—`SIDEBAR_REGION`—is defined in `/regionDefinitions.jsp`, which is included by the JSP page listed in Example 4-5.a. `/regionDefinitions.jsp` is listed in Example 4-5.b.

Example 4-5.b /regionDefinitions.jsp: Defining a Region

```
<%@ taglib uri='regions' prefix='region' %>

<region:define id='SIDEBAR_REGION' scope='application'
         template='hscf.jsp'>
   <region:put section='header'  content='/header.jsp'/>
   <region:put section='sidebar' content='/sidebar.jsp'/>
   <region:put section='content' content='/content.jsp'/>
   <region:put section='footer'  content='/footer.jsp'/>
</region:define>
```

The JSP page listed in Example 4-5.b uses the `region:define` tag to define a region named `SIDEBAR_REGION` that's stored in application scope. That region is created by the `region:define` start tag and its name and scope are specified with the `region:define` tag's `id` and `scope` attributes, respectively. The `region:define` tag also has a `template` attribute that specifies the template used by a region.

Like the `region:render` tag, `region:define` can contain `region:put` tags, which store section names and content in the region created by the `region:define` start tag.

Why would you want to define regions somewhere other than where they are rendered? Because that separation allows you to group region definitions in a single file, thereby giving you access to all of the regions defined by an application. That makes maintaining those regions significantly easier. Also, because regions can be nested—see "Nesting Regions" below—and can inherit from one another—see "Extending Regions" on page 112—it's easier to maintain multiple regions if they are all defined in one file.

Nesting Regions

Because sections and regions are implemented with the Composite design pattern, you can specify a region for a section's content; for example, the JSP page shown in Figure 4-5 nests one region inside another.

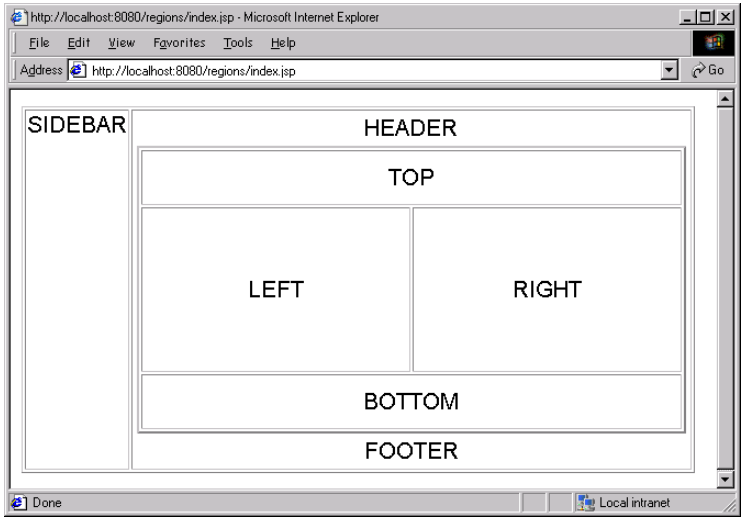

Figure 4-5 Nesting Regions

The JSP page shown in Figure 4-5 is identical to the JSP page listed in Example 4-5.a on page 109. That JSP page renders the preexisting SIDEBAR_REGION, which is defined in Example 4-5.c.

Example 4-5.c /regionDefinitions.jsp

```
<%@ taglib uri='regions' prefix='region' %>

<region:define id='SIDEBAR_REGION' scope='application'
         template='hscf.jsp'>
   <region:put section='header'  content='/header.jsp'/>
   <region:put section='sidebar' content='/sidebar.jsp'/>
   <region:put section='content' content='BORDER_REGION'/>
   <region:put section='footer'  content='/footer.jsp'/>
</region:define>

<region:define id='BORDER_REGION' scope='application'
         template='tlbr.jsp'>
   <region:put section='top'    content='/top.jsp'/>
   <region:put section='left'   content='/left.jsp'/>
   <region:put section='right'  content='/right.jsp'/>
   <region:put section='bottom' content='/bottom.jsp'/>
</region:define>
```

The SIDEBAR_REGION specifies the BORDER_REGION as the content for its content section, which nests the BORDER_REGION inside the SIDEBAR_REGION.

Notice that the BORDER_REGION does not need to be defined before it is specified in the SIDEBAR_REGION. That's because the SIDEBAR_REGION does not look for the BORDER_REGION until the SIDEBAR_REGION is rendered.

For completeness, the template used by the BORDER_REGION is listed in Example 4-5.d.

Example 4-5.d The Template Used by the BORDER_REGION in Example 4-5.c

```
<html><head>
  <%@ taglib uri='regions' prefix='region' %>
</head>

<table border='2' width='500'>
     <tr> <%-- Top Row --%>
        <td colspan='2' align='center' height='50'>
           <region:render section='top'/>
        </td>
     </tr>
```

```
      <tr> <%-- Left and Right Rows --%>
         <td align='center' width='50%' height='150'>
            <region:render section='left'/>
         </td>
         <td align='center' width='50%' height='150'>
            <region:render section='right'/>
         </td>
      </tr>

      <tr> <%-- Bottom Row --%>
         <td colspan='2' align='center' height='50'>
            <region:render section='bottom'/>
         </td>
      </tr>
</table>

</body></html>
```

The template listed in Example 4-5.d renders the top, left, right, and bottom sections defined in the `BORDER_REGION`.[6]

Extending Regions

Perhaps the most exciting feature of regions is that one region can extend another; for example, the JSP page shown in Figure 4-6 renders a region that extends the `SIDEBAR_REGION` used throughout this chapter.

The JSP page shown in Figure 4-6 is listed in Example 4-6.a.

Like the JSP page listed in Example 4-5.a on page 109, the JSP page listed in Example 4-6.a includes a JSP file that contains region definitions. The JSP page listed in Example 4-6.a subsequently renders the `EXTENDED_SIDEBAR_REGION`.

6. The name `BORDER_REGION` comes from the AWT `BorderLayout`, which lays out components in a similar fashion.

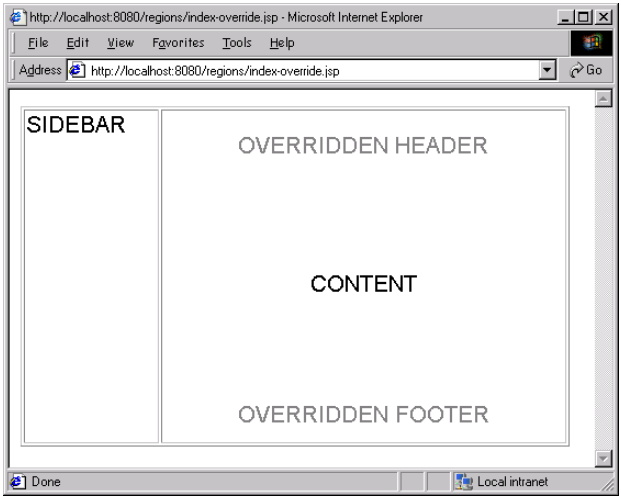

Figure 4-6 Overriding Sections

Example 4-6.a Extending an Existing Region

```
<%@ taglib uri='regions' prefix='region' %>
<%@ include file='/regionDefinitions-override.jsp' %>

<region:render region='EXTENDED_SIDEBAR_REGION'/>
```

The JSP page that defines the `SIDEBAR_REGION` and the `EXTENDED_SIDEBAR_REGION` is listed in Example 4-6.b.

Example 4-6.b /regionDefinitions-override.jsp

```
<%@ taglib uri='regions' prefix='region' %>

<region:define id='SIDEBAR_REGION' scope='application'
         template='hscf.jsp'>
   <region:put section='header'  content='/header.jsp'/>
   <region:put section='sidebar' content='/sidebar.jsp'/>
   <region:put section='content' content='/content.jsp'/>
   <region:put section='footer'  content='/footer.jsp'/>
</region:define>

<region:define id='EXTENDED_SIDEBAR_REGION' scope='application'
           region='SIDEBAR_REGION'>
   <region:put section='header' content='/overridden-header.jsp'/>
   <region:put section='footer' content='/overridden-footer.jsp'/>
</region:define>
```

Usually, regions are defined with a template, as is the case for the `SIDEBAR_REGION` defined in Example 4-6.b. But you can also define a region with another region, as is the case for the `EXTENDED_SIDEBAR_REGION`, which is defined with the `SIDEBAR_REGION`.

Defining one region in terms of another causes the newly defined region to "inherit" from the specified region; for example, `EXTENDED_SIDEBAR_REGION` is defined in terms of the `SIDEBAR_REGION`, so `EXTENDED_SIDEBAR_REGION` inherits all of the `SIDEBAR_REGION`'s content. Subsequently, `EXTENDED_SIDEBAR_REGION` selectively overrides the content for the header and footer sections defined in the `SIDEBAR_REGION` with `region:put` tags.

Combining Features

You may never need to create a web page with regions as complicated as those shown in Figure 4-7, but the regions tag library discussed in this chapter lets you combine nested and extended regions in all sorts of interesting ways.

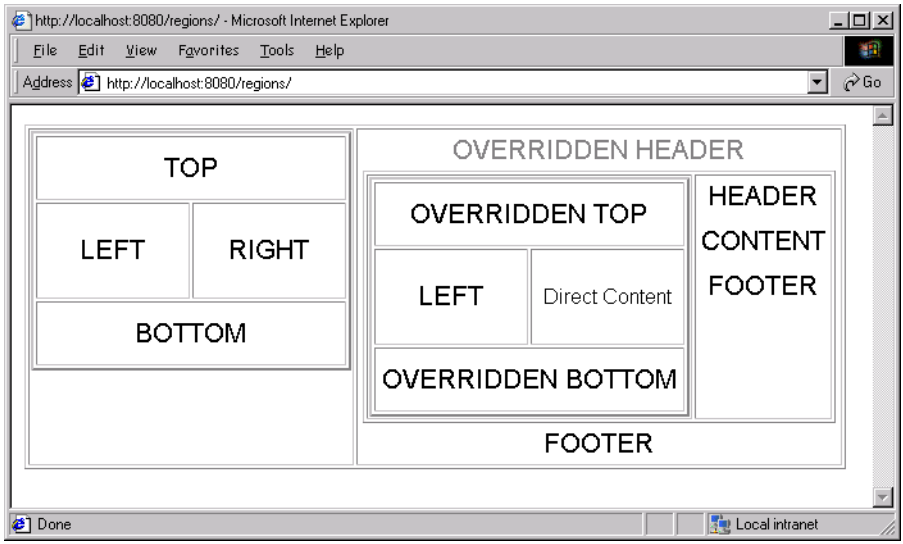

Figure 4-7 Using Extended and Nested Regions

The JSP page shown in Figure 4-7 is listed in Example 4-7.a.

Example 4-7.a /index.jsp

```
<%@ taglib uri='regions' prefix='regions' %>
<%@ include file='/regionDefinitions-override.jsp' %>

<region:render region='EXTENDED_SIDEBAR_REGION'/>
```

The JSP file included above is listed in Example 4-7.b.

Example 4-7.b /regionDefinitions-override.jsp

```
<%@ taglib uri='regions' prefix='region' %>

<region:define id='SIDEBAR_REGION' scope='request'
         template='hscf.jsp'>
  <region:put section='header'  content='/header.jsp'/>
  <region:put section='sidebar' content='EXTENDED_BORDER_REGION'/>
  <region:put section='content' content='/content.jsp'/>
  <region:put section='footer'  content='/footer.jsp'/>
</region:define>

<region:define id='BORDER_REGION' scope='request'
         template='tlbr.jsp'>
   <region:put section='top'    content='/top.jsp'/>
   <region:put section='bottom' content='/bottom.jsp'/>
   <region:put section='left'   content='/left.jsp'/>
   <region:put section='right'  content='/right.jsp'/>
</region:define>

<region:define id='EXTENDED_SIDEBAR_REGION' scope='request'
           region='SIDEBAR_REGION'>
  <region:put section='header' content='/overridden-header.jsp'/>
  <region:put section='sidebar'content='BORDER_REGION'/>
  <region:put section='content'content='SIDEBAR_REGION'/>
</region:define>

<region:define id='EXTENDED_BORDER_REGION' scope='request'
           region='BORDER_REGION'>
   <region:put section='top'    content='/overridden-top.jsp'/>
   <region:put section='bottom' content='/overridden-bottom.jsp'/>
   <region:put section='right'>
      <font size='4'>Direct Content</font>
   </region:put>
</region:define>
```

The regions defined in Example 4-7.b are so intertwined that they probably border on some kind of technology abuse; nonetheless, they demonstrate how powerful the regions custom tag library is.

The `EXTENDED_SIDEBAR_REGION` is the region that's rendered by the JSP page shown in Figure 4-7. That region extends `SIDEBAR_REGION` and overrides the `header`, `sidebar`, and `content` sections. The `sidebar` and `content` regions for the `EXTENDED_SIDEBAR_REGION` are the `BORDER_REGION` and `SIDEBAR_REGION`, respectively.

The `SIDEBAR_REGION` uses the `EXTENDED_BORDER_REGION` for its `sidebar` section. The `EXTENDED_BORDER_REGION` extends `BORDER_REGION` and overrides the `top`, `bottom`, and `right` sections. Notice that the `right` section for the `EXTENDED_BORDER_REGION` defines its content in the body of the `region:put` tag. That feature is discussed on page 130.

The regions defined above use the templates—`hscf.jsp` and `tlbr.jsp`—listed in Example 4-3.b on page 106 and Example 4-5.d on page 111, respectively, except that the table widths and heights for those templates were reduced to scale the JSP page shown in Figure 4-7 down to a reasonable size.

Region Tag Implementations

The regions custom tag library used in this chapter consists of four beans and three custom tags. The rest of this chapter explores the implementation of those beans and tags.

The Beans

The beans used by the regions custom tag library are listed in Table 4-1.

Table 4-1 Beans Used by the Regions Custom Tag Library[1]

Bean	Description
`Content`	Content that's rendered in a JSP `PageContext`
`Section`	Content that's part of a region
`Region`	A container that contains sections
`RegionStack`	A stack of regions maintained in application scope

1. All of the beans listed above are from the `beans.regions` package.

A class diagram for the beans listed in Table 4-1 is shown in Figure 4-8.

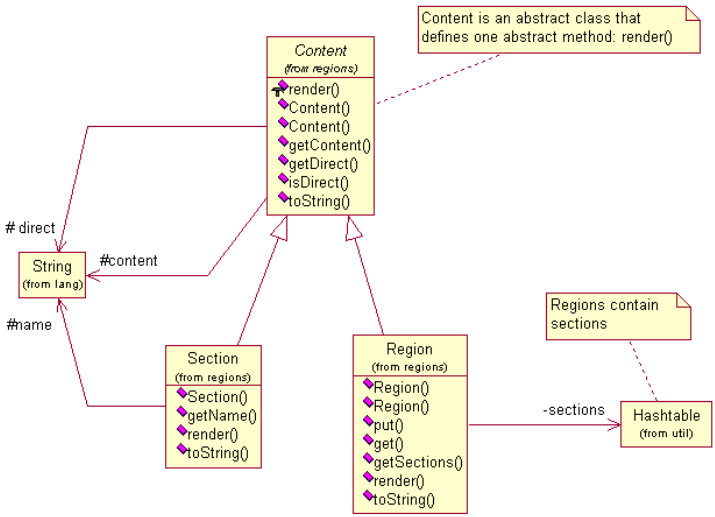

Figure 4-8 Region Beans

`Content` is an abstract class that's the superclass for `Section` and `Region`. Those three classes constitute an implementation of the Composite design pattern because the `Content` abstract class represents both primitives (sections) and their containers (regions).

The `Section` and `Region` classes both implement the `render` method, which is the only abstract method defined by the `Content` class. The `Region` class contains a hash table of its sections.

The `Content` class is listed in Example 4-8.a.

Example 4-8.a /WEB-INF/classes/beans/templates/Content.java

```
package beans.regions;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;

public abstract class Content implements java.io.Serializable {
   protected final String content, direct;

   // Render this content in a JSP page
   abstract void render(PageContext pc) throws JspException;

   public Content(String content) {
      this(content, "false");
   }
```

```
    public Content(String content, String direct) {
       this.content = content;
       this.direct  = direct;
    }
    public String getContent() {
       return content;
    }
    public String getDirect() {
       return direct;
    }
    public boolean isDirect() {
       return Boolean.valueOf(direct).booleanValue();
    }
    public String toString() {
       return "Content: " + content;
    }
}
```

The `Content` class is a simple abstract class that defines a `render` abstract method. That class also maintains two properties: `content` and `direct`. The `content` property represents the content rendered by the `render` method, and the `direct` attribute specifies how that content is rendered. If the `direct` attribute is `true`, content is rendered directly by printing it. If the `direct` attribute is `false`, content is rendered by including it.

The `Section` class is listed in Example 4-8.b.

Example 4-8.b /WEB-INF/classes/beans/templates/Section.java

```
package beans.regions;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;

// A section is content with a name that implements
// Content.render. That method renders content either by including
// it or by printing it directly, depending upon the direct
// value passed to the Section constructor.
//
// Note that a section's content can also be a region;if so,
// Region.render is called from Section.Render().

public class Section extends Content {
   protected final String name;

   public Section(String name, String content, String direct) {
      super(content, direct);
```

```
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void render(PageContext pageContext)
                                    throws JspException {
        if(content != null) {
            // see if this section's content is a region
            Region region = (Region)pageContext.
                                    findAttribute(content);
            if(region != null) {
                // render the content as a region
                RegionStack.push(pageContext, region);
                region.render(pageContext);
                RegionStack.pop(pageContext);
            }
            else {
                if(isDirect()) {
                    try {
                        pageContext.getOut().print(content.toString());
                    }
                    catch(java.io.IOException ex) {
                        throw new JspException(ex.getMessage());
                    }
                }
                else {
                     try {
                        pageContext.include(content.toString());
                    }
                    catch(Exception ex) {
                         throw new JspException(ex.getMessage());
                    }
                }
            }
        }
    }
    public String toString() {
        return "Section: " + name + ", content= " +
                                        content.toString();
    }
}
```

The `Section` class extends `Content` and implements the `render` method. If a section's `direct` attribute is `true`, that section prints its content to the implicit `out` variable. If a section's `direct` attribute is `false`, that section uses the JSP page context to include its content.

If a section's content is a region, the section pushes the region onto a stack—see Example 4-8.d for more information about the stack—and renders the region. After the region has been rendered, its section pops it off the stack.

The `Region` class is listed in Example 4-8.c.

Example 4-8.c /WEB-INF/classes/beans/templates/Region.java

```
package beans.regions;

import java.util.Enumeration;
import java.util.Hashtable;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.JspException;

// A region is content that contains a set of sections.

public class Region extends Content {
   private Hashtable sections = new Hashtable();

   public Region(String content) {
      this(content, null); // content is the name of a template
   }
   public Region(String content, Hashtable hashtable) {
      super(content);

      if(hashtable != null)
         sections = (Hashtable)hashtable.clone();
   }
   public void put(Section section) {
      sections.put(section.getName(), section);
   }
   public Section get(String name) {
      return (Section)sections.get(name);
   }
   public Hashtable getSections() {
      return sections;
   }
   public void render(PageContext pageContext)
                                    throws JspException {
      try {
         pageContext.include(content);
      }
      catch(Exception ex) { // IOException or ServletException
         throw new JspException(ex.getMessage());
      }
```

```
    }
    public String toString() {
       String s = "Region: " + content.toString() + "<br/>";
       int indent = 4;
       Enumeration e = sections.elements();

       while(e.hasMoreElements()) {
          Section section = (Section)e.nextElement();
          for(int i=0; i < indent; ++i) {
             s += " ";
          }
          s += section.toString() + "<br/>";
       }
       return s;
    }
}
```

Like the `Section` class, the `Region` class extends `Content` and implements the `render` method. Additionally, regions maintain a hash table of sections. A region's sections can be accessed with `Region.get` or with `Region.getSections`; the former returns a section given its name, and the latter returns the region's hash table of sections.

Regions are maintained on a stack in application scope. That stack is represented by the `RegionStack` class, which is listed in Example 4-8.d.

Example 4-8.d /WEB-INF/classes/beans/templates/RegionStack.java

```
package beans.regions;

import javax.servlet.jsp.PageContext;
import java.util.Stack;

public class RegionStack {
   private RegionStack() { } // no instantiations

   public static Stack getStack(PageContext pc) {
      Stack s = (Stack)pc.getAttribute("region-stack",
                                       PageContext.APPLICATION_SCOPE);
      if(s == null) {
         s = new Stack();
         pc.setAttribute("region-stack", s,
                         PageContext.APPLICATION_SCOPE);
      }
      return s;
   }
```

```
    public static Region peek(PageContext pc) {
       return (Region)getStack(pc).peek();
    }
    public static void push(PageContext pc, Region region){
       getStack(pc).push(region);
    }
    public static Region pop(PageContext pc) {
       return (Region)getStack(pc).pop();
    }
}
```

Because nested templates could potentially overwrite their enclosing template's content, regions are stored on a stack. That stack is represented by the `RegionStack` class listed above, which provides `static` methods for pushing regions on the stack, popping them off, and peeking at the top region on the stack.

The Tag Handlers

The tag handlers from the regions custom tag library are listed in Table 4-2.

Table 4-2 Tags From the Regions Custom Tag Library[1]

Bean	Description
`RegionTag`	Is a base class for `RegionDefinitionTag` and `RenderTag`
`RegionDefinitionTag`	Creates a region and stores it in a specified scope
`RenderTag`	Renders a region or a section
`PutTag`	Creates a section and stores it in a region

1. All of the tags listed above are from the `tags.regions` package.

A class diagram for the tags listed in Table 4-2 is shown in Figure 4-9.

Both the `region:render` and `region:define` tags, whose tag handlers are `RenderTag` and `RegionDefinitionTag`, respectively, create a region. That functionality is encapsulated in the `RegionTag` class, which is the base class for `RenderTag` and `RegionDefinitionTag`. All three of those classes ultimately extend `TagSupport`.

The `PutTag` class is the tag handler for the `region:put` tag. That tag handler extends `BodyTagSupport` so you can specify content in that tag's body.

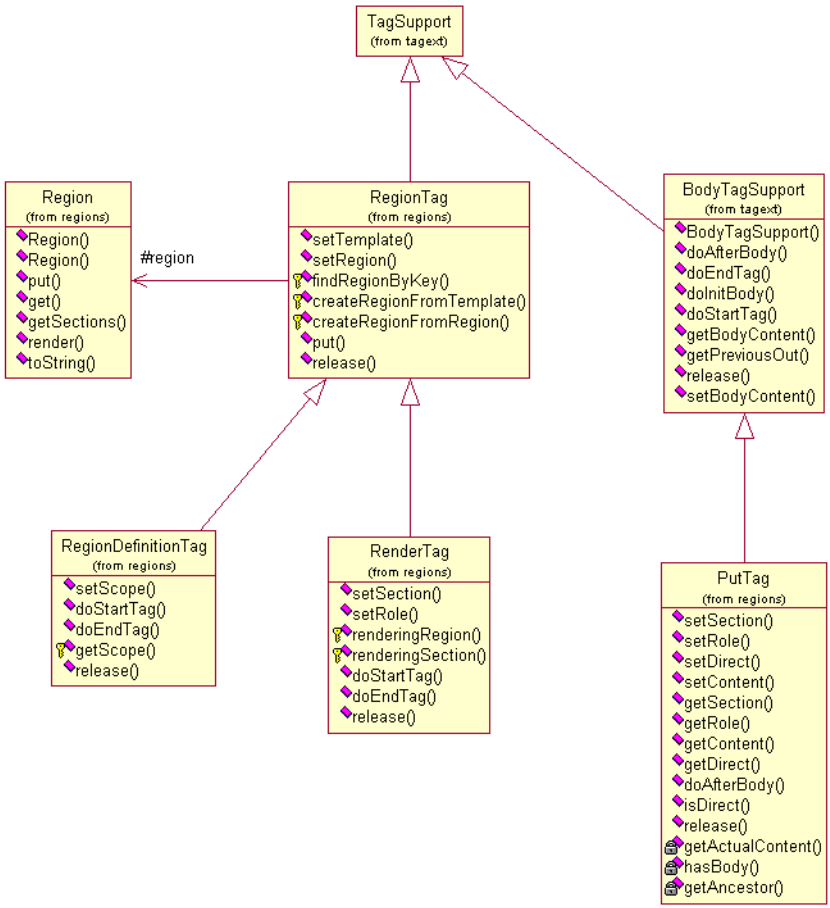

Figure 4-9 Region Tags

The `RegionTag` class is listed in Example 4-9.a.

Example 4-9.a /WEB-INF/classes/tags/regions/RegionTag.java

```
package tags.regions;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.TagSupport;

import beans.regions.Section;
import beans.regions.Region;
```

```
public class RegionTag extends TagSupport {
   protected Region region = null;
   protected String template = null;
     private String regionKey = null;

   public void setTemplate(String template) {
      this.template = template;
   }
   public void setRegion(String regionKey) {
      this.regionKey = regionKey;
   }
   protected boolean findRegionByKey() throws JspException {
      if(regionKey != null) {
         region = (Region)pageContext.findAttribute(regionKey);
         if(region == null) {
            throw new JspException("can't find page definition " +
                                   "attribute with this key: " +
                                    regionKey);
         }
      }
      return region != null;
   }
   protected void createRegionFromTemplate() throws JspException {
      if(template == null)
         throw new JspException("can't find template");

      region = new Region(template);
   }
   protected void createRegionFromRegion() throws JspException {
      findRegionByKey();

      if(region == null)
         return;

      region = new Region(region.getContent(),   // the template
                          region.getSections()); // sections
   }
   public void put(Section section) {
      region.put(section);
   }
   public void release() {
      super.release();
      region = null;
      regionKey = null;
      template = null;
   }
}
```

Regions can be created from a template or another region, both of which are specified with a `template` or `region` attribute. The `RegionTag` class provides setter methods for those attributes, implements methods for creating a region from a template or another region, and provides a method for locating an existing region given its name. All three of those methods are used by `RegionTag` subclasses.

The `RegionDefinitionTag` class is listed in Example 4-9.b.

Example 4-9.b /WEB-INF/classes/tags/regions/RegionDefinitionTag.java

```
package tags.regions;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.TagSupport;

import beans.regions.Content;
import beans.regions.Region;

public class RegionDefinitionTag extends RegionTag {
   private String scope = null;

   public void setScope(String scope) {
      this.scope = scope;
   }
   public int doStartTag() throws JspException {
      if(region != null && template != null)
         throw new JspException("regions can be created from "  +
                                "a template or another region," +
                                "but not both");
      createRegionFromRegion();

      if(region == null)
         createRegionFromTemplate();

      return EVAL_BODY_INCLUDE;
   }
   public int doEndTag() throws JspException {
      pageContext.setAttribute(id, region, getScope());
      return EVAL_PAGE;
   }
   protected int getScope() {
      int constant = PageContext.PAGE_SCOPE;

      scope = (scope == null) ? "page" : scope;
```

```
        if("page".equalsIgnoreCase(scope))
            constant = PageContext.PAGE_SCOPE;
        else if("request".equalsIgnoreCase(scope))
            constant = PageContext.REQUEST_SCOPE;
        else if("session".equalsIgnoreCase(scope))
            constant = PageContext.SESSION_SCOPE;
        else if("application".equalsIgnoreCase(scope))
            constant = PageContext.APPLICATION_SCOPE;

        return constant;
    }
    public void release() {
        super.release();
        scope = "page";
    }
}
```

The `RegionDefinitionTag` class extends `RegionTag` and creates a region, either from a template or another region. Notice that `RegionDefinitionTag.doStartTag` throws an exception if both `region` and `template` attributes are specified. That exception is thrown because a region can be created from a template or another region, but not both.

The `RenderTag` class is listed in Example 4-9.c.

Example 4-9.c /WEB-INF/classes/tags/regions/RenderTag.java

```
package tags.regions;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.TagSupport;

import beans.regions.Content;
import beans.regions.Section;
import beans.regions.Region;
import beans.regions.RegionStack;

public class RenderTag extends RegionTag {
    private String sectionName=null, role=null;

    public void setSection(String s)  { this.sectionName = s; }
    public void setRole(String s)     { this.role = s;        }

    protected boolean renderingRegion() {
        return sectionName == null;
    }
```

```
   protected boolean renderingSection() {
      return sectionName != null;
   }
   public int doStartTag() throws JspException {
      HttpServletRequest request = (HttpServletRequest)
                                    pageContext.getRequest();

      if(role != null && !request.isUserInRole(role))
         return SKIP_BODY;

      if(renderingRegion()) {
         if(!findRegionByKey()) {
            createRegionFromTemplate();
         }
         RegionStack.push(pageContext, region);
      }
      return EVAL_BODY_INCLUDE;
   }
   public int doEndTag() throws JspException {
      Region region = RegionStack.peek(pageContext);

      if(region == null)
         throw new JspException("Can't find region");

      if(renderingSection()) {
         Section section = region.get(sectionName);

         if(section == null)
            return EVAL_PAGE; // ignore missing sections

         section.render(pageContext);
      }
      else if(renderingRegion()) {
         try {
            region.render(pageContext);
            RegionStack.pop(pageContext);
         }
         catch(Exception ex) { // IOException or ServletException
              throw new JspException(ex.getMessage());
         }
      }
      return EVAL_PAGE;
   }
   public void release() {
      super.release();
      sectionName = role = null;
   }
}
```

The RenderTag class renders both sections and regions. If a section attribute is specified, that tag renders a section; otherwise, it renders a region. If a region is rendered, that region is pushed on the region stack in doStartTag and popped off the stack in doEndTag. See "The Beans" on page 116 for more information about that region stack.

The PutTag class is listed in Example 4-9.d.

Example 4-9.d /WEB-INF/classes/tags/regions/PutTag.java

```
package tags.regions;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.BodyTagSupport;
import javax.servlet.jsp.tagext.TagSupport;

import beans.regions.Content;
import beans.regions.Section;

public class PutTag extends BodyTagSupport {
   private String section, role, content, direct = null;

   public void setSection(String section){this.section = section;}
   public void setRole   (String role)   {this.role    = role;   }
   public void setDirect (String direct) {this.direct  = direct; }
   public void setContent(String cntnt)  {this.content = cntnt;  }

   public String getSection() { return section; }
   public String getRole()    { return role;    }
   public String getContent() { return content; }
   public String getDirect()  { return direct;  }

   public int doAfterBody() throws JspException {
      HttpServletRequest request =
                    (HttpServletRequest)pageContext.getRequest();

      if(role != null && !request.isUserInRole(role))
         return EVAL_PAGE;

      RegionTag regionTag = (RegionTag)getAncestor(
                                        "tags.regions.RegionTag");
      if(regionTag == null)
         throw new JspException("No RegionTag ancestor");

      regionTag.put(new Section(section, getActualContent(),
                                         isDirect()));
```

```
      return SKIP_BODY;
   }
   public String isDirect() {
      if(hasBody()) return "true";
      else          return direct == null ? "false" : "true";
   }
   public void release() {
      super.release();
      section = content = direct = role = null;
   }
   private String getActualContent() throws JspException {
      String bodyAndContentMismatchError =
         "Please specify template content in this tag's body " +
         "or with the content attribute, but not both.",
             bodyAndDirectMismatchError =
         "If content is specified in the tag body, the " +
         "direct attribute must be true.";

      boolean hasBody = hasBody();
      boolean contentSpecified = (content != null);

      if((hasBody && contentSpecified) ||
         (!hasBody && !contentSpecified))
         throw new JspException(bodyAndContentMismatchError);

      if(hasBody && direct != null &&
         direct.equalsIgnoreCase("false"))
         throw new JspException(bodyAndDirectMismatchError);

      return hasBody ? bodyContent.getString() : content;
   }
   private boolean hasBody() {
      if (bodyContent == null)
         return (false);

      return ! bodyContent.getString().equals("");
   }
   private TagSupport getAncestor(String className)
                                 throws JspException {
      Class klass = null; // can't name variable "class"
      try {
         klass = Class.forName(className);
      }
      catch(ClassNotFoundException ex) {
         throw new JspException(ex.getMessage());
      }
      return (TagSupport)findAncestorWithClass(this, klass);
   }
}
```

The `PutTag` class creates a section and stores it in the region created by its enclosing `region:define` or `region:render` tag. The `PutTag` class extends `BodyTagSupport` because it allows content that's printed directly to be specified in the tag's body; for example, the `region:put` tag can be used like this:

```
<region:render template='/WEB-INF/jsp/templates/hscf.jsp'>
   <region:put section='title'>
     The Fruitstand
   </region:put>
   ...
</region:render>
```

If a `region:put` tag has a body, as is the case for the preceding code fragment, that body must be content that's printed directly. The `PutTag` class enforces that restraint by checking to make sure that the `direct` attribute is `true` if the tag has body content.

Conclusion

Components, containers, and layout managers are found in most window toolkits because they allow applications to be built from modular components. This chapter has demonstrated how to implement those types of objects for JSP-based web applications.

By using the sections, regions, and templates discussed in this chapter, you can implement web applications that are extensible, reusable, and maintainable. See "A Case Study" on page 390 for more information about using sections, regions, and templates in a nontrivial web application.