04 469513 Ch01.qxd 1/16/04 11:04 AM Page 3

1

The Java Portlet API (JSR 168)

This chapter discusses the centerpiece of portal development, the Java Portlet API, Java Specification Request 168 (JSR 168). The chapter explains the concepts in the specification, explaining how they fit into portal architectures to enable the developer to be an effective **portal** developer.

Portlet Fundamentals

A **portal** server handles client requests. Much like a Web application server has a Web container to manage running Web components (servlets, JSPs, filters, and so on), a portal has a **portlet container** to manage running portlets. Note that most Web application servers, such as Tomcat, have additional features beyond the Web container (management console, user databases, and so on), including some specialized Web applications (an administration Web application, for example). Portals are expected to follow a similar pattern, providing higher level functionality wrapped around the portlet container that adheres to the specification, enabling portlet applications to be portable, just as Web applications are.

The Portlet API is an extension to the servlet specification, which means that a portlet container is also, by definition, a Web container. Figure 1.1 demonstrates the Portal **stack**, which indicates how the various parts build upon each other to provide a portal server.

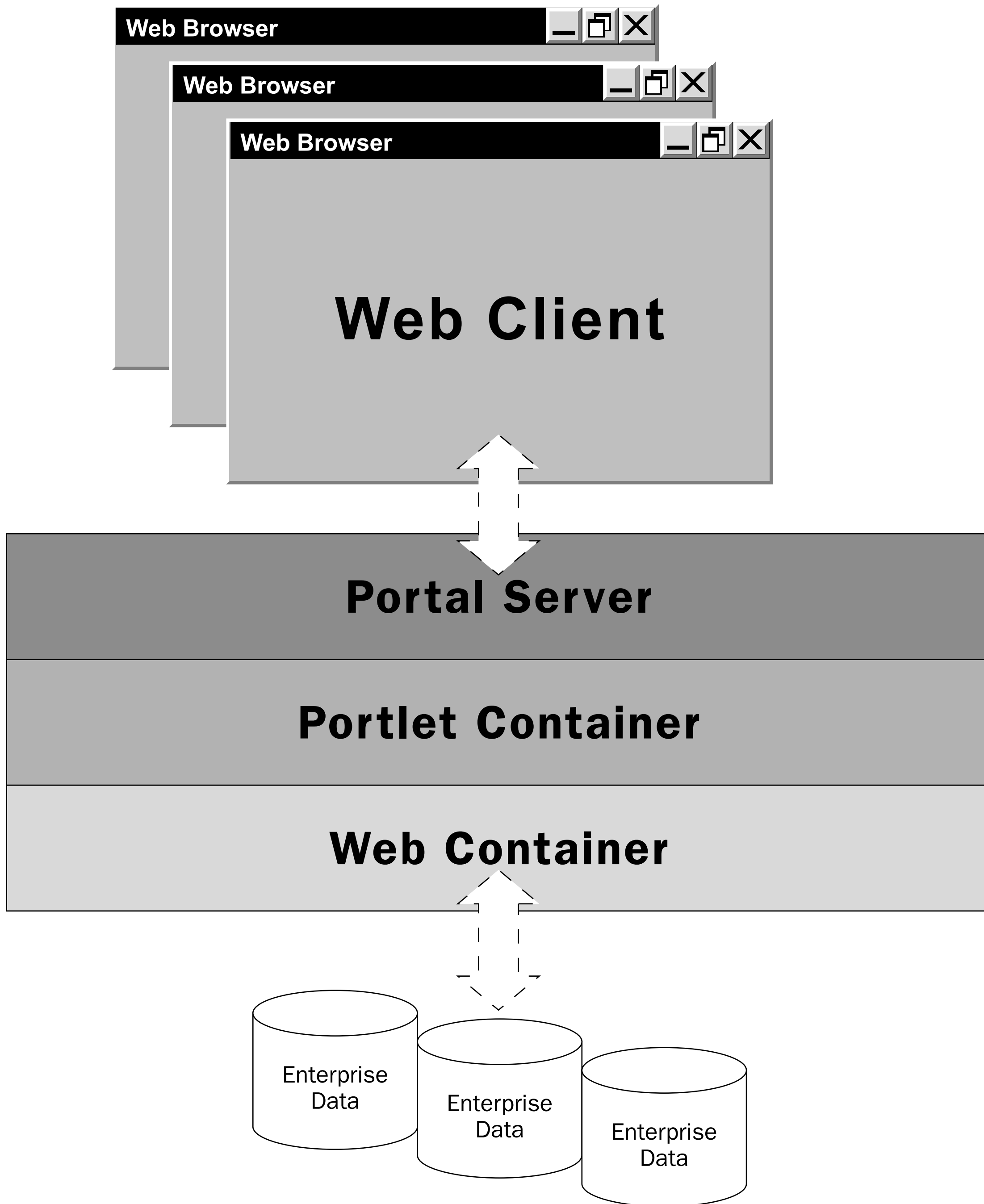


Figure 1.1

As you can see, the servlet container is the basis for any portal, upon which the portlet container extension is built. Likewise, a portal is built on top of that portlet container, which manages the portlets required to handle client requests.

Before describing the relationships between portlets and servlets, we should discuss a few of the fundamental definitions related to the Portlet API.

The following table provides a list of the definitions that are used in this chapter to explain the Portlet API. Of course, many others will be introduced throughout the chapter, but these are the fundamental ones required for an understanding of the chapter.

Term	Definition
Portal	A Web-based application server that aggregates, customizes, and personalizes content to give a presentation layer to enterprise information systems.
Portlet	A pluggable Web component managed by a portlet container; it provides dynamic content as part of an aggregated user interface.
Fragment	The result of executing a portlet, a "chunk" of markup (HTML, XML, and so on) that adheres to certain rules (see the sidebar "Rules for Fragments").
Portlet Container	The runtime environment of a portlet. It manages the life cycle of portlets, and handles requests from the portal by invoking portlets inside the container.

Although this is not an all-encompassing list, it provides the basic terms that are used repeatedly in this chapter.

Fragments are not allowed to use certain tags from their respective markup languages. These tags include html, title, head, body, base, frameset, frame, link, meta, and style. Use of these tags will invalidate the entire fragment. This is particularly important to developers (like us!) who have been abusing the forgiving nature of browsers until now.

Portlets and Servlets

As mentioned before, the Portlet API is an extension to the Servlet API. Therefore, there are both similarities and differences between the components. It is important to understand these distinctions in order to understand why there is a portlet specification (and break habits wrought from using their similar servlet sisters).

The similarities between portlets and servlets are as follows:

- Portlets and servlets are both Java 2 Enterprise Edition (J2EE) Web components.
- Both are managed by containers, which control their interactions and life cycle.
- Each generates dynamic Web content via a request/response paradigm.

04 469513 Ch01.qxd 1/16/04 11:04 AM Page 6

The differences between portlets and servlets are as follows:

- Portlets generate fragments, whereas servlets generate complete documents.
- Unlike servlets, portlets are not bound directly to a URL.
- Portlets have a more sophisticated request scheme, with two types of requests: action and render.
- Portlets adhere to a standardized set of states and modes that define their operating context and rendering rules.

Portlets are able to do some things that servlets cannot, such as the following:

- Portlets have a far more sophisticated mechanism for accessing and persisting configuration information.
- Portlets have access to user profile information, beyond the basic user and role information provided in the servlet specification.
- Portlets are able to perform portlet rewriting, so as to create links that are independent of the portal server implementation (which have various methods to track session information, and so on).
- Portlets have two different session scopes in which to store objects: application wide and portlet private.

Portlets lack some servlet features:

- Portlets cannot alter HTTP headers or set the response encoding.
- Portlets cannot access the URL that the client used to initiate the request on the portal.

Portlet applications are extended Web applications. Therefore, both types of applications are deployed in Web archive files (WAR) and both contain a Web application deployment descriptor (`web.xml`). However, a portlet application also contains a portlet application deployment descriptor (`portlet.xml`).

Because a portlet application is an extension of a Web application, it is logical that it could contain other Web application components. Portlets can use JSPs and servlets to implement their functionality. This capability is discussed later in the chapter.

Portal Interactions

It makes sense to show how a typical portal interaction occurs before diving into details about how portlets can render themselves with JSPs and servlets. Figure 1.2 demonstrates the chain of events that occur inside a portal to manage a typical client request.

Inside a portal is a Portlet API–compliant portlet container that manages the runtime state of portlets. The container evaluates those portlets into fragments, either by making requests of the portlet or by taking a fragment from cache. Then, the container hands the fragments to the portal server that manages aggregating them into portal pages.

Now that you have looked at portals, portlets, and portal containers at a high level, it is time to dig into the specifics about how to build portlets.

04 469513 Ch01.qxd 1/16/04 11:04 AM Page 7

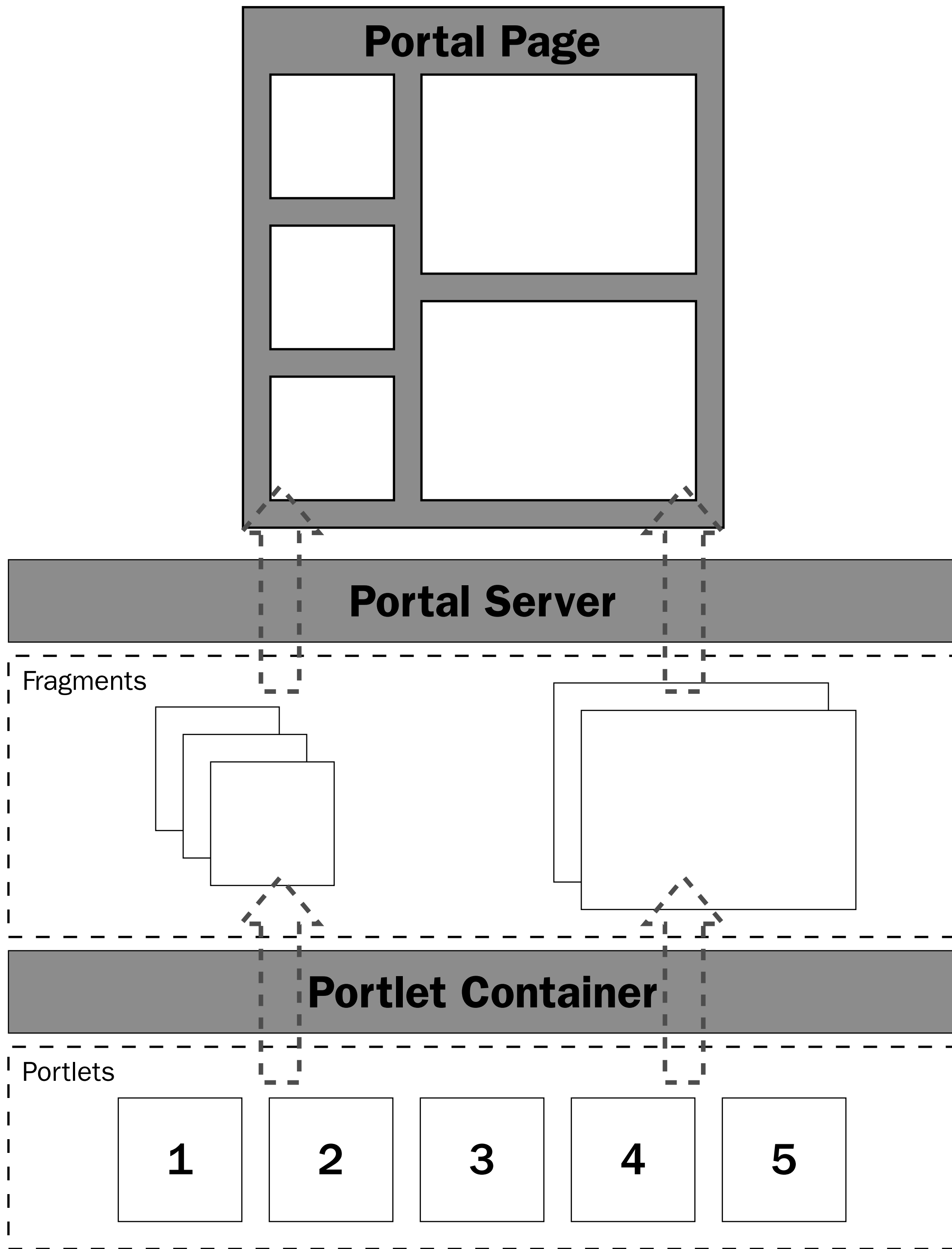


Figure 1.2

The Portlet Interface and the GenericPortlet

The `Portlet` interface defines the behaviors that all portlets must implement. Typically, you would prefer to extend the `GenericPortlet` class to build a portlet, because it provides structures for providing all of the typical portlet implementation methods, not simply the required ones.

Portlet Life Cycle

Much like servlets, a portlet's life cycle is managed by the container, and has an `init` method that is used to manage the initialization requirements (creating resources, configuring, and so on). Portlets are not guaranteed to be loaded until needed, unless you configure the container to load them on startup.

The `init` method takes an object that implements the `PortletConfig` interface, which manages initialization parameters and the portlet's `ResourceBundle`. This object can be used to get a reference to the object that implements the `PortletContext` interface.

Portlet developers don't typically spend a lot of time worrying about the intricacies of portlet container initialization exceptions, because generally they are thrown, and the developer reacts to them (debugging the circumstance that led to the exception and correcting it if appropriate). However, it is worth noting that an `UnavailableException` is able to specify a time for which the portlet will be unavailable. This could be both useful (keeping the portlet container from continuously trying to load the portlet) and aggravating (Why isn't the portlet container reloading my portlet?!) to a developer.

The `destroy` method provides the opportunity to clean up resources that were established in the `init` method. This is analogous to the `destroy` method in a servlet, and is called once when the container disposes of the portlet.

> **When an exception is thrown in the portlet `init` method, the `destroy` method is guaranteed not to be called. Therefore, if resources are created in the `init` method prior to the exception being thrown, the developer cannot expect the `destroy` method to clean them up, and must handle them in the exception's catch block.**

Portlet Runtime States

When a portlet is running, it has an associated `Preferences` object that allows for customization of the portlet. The initial values of the preferences are those specified in the deployment descriptor, but the portlet has full programmatic access to its preferences.

When a portlet is placed on a page, a `Preferences` object is related to it. The pairing of the portlet and a `Preferences` object on a page is known as a **portlet window**. A page can contain many of the same portlet windows within its display.

Before you start wondering why all of these `Preferences` objects are necessary, realize that this is providing the capability to perform a major features of a portal — customization. While the initial portlet `Preferences` object is great for specifying the configuration and runtime state of the portlet, it is

04 469513 Ch01.qxd 1/16/04 11:04 AM Page 9

necessary to tweak that state to handle customized views of the portlet. For example, say you have an employee directory portlet. Obviously, it would require certain preferences to get it running. However, when that employee directory portlet is embedded on a "Finance Department" home page, it should not only have a customized look and feel, but also have preferences related to the fact that it is on that page, such as showing only Finance Department employees.

Portlet Request Handling

Two types of requests can be issued against a portlet: action requests and render requests. Not coincidentally, these requests have their accompanying URL types: action URLs and render URLs. An action URL targets the portlet's `processAction` method, while the render URL targets its `render` method.

"There Can Be Only One"

If a client request is an action request, then it can target only one portlet, which must be executed first. No other action requests can be performed on the remaining portlets, only render requests. Figure 1.3 illustrates how a portal container would manage an action request.

As you can see, the portlet container will execute `processAction` on the targeted portlet, waiting until it finishes before it executes `render` on the rest of the portlets on the page. The calling of the `render` method on the remaining portlets can be done in any order, and can be done in parallel.

The `processAction` method is responsible for changing state on the given portlet, while the `render` method is responsible for generating the appropriate presentation content of the portlet. Therefore, it is logical that a user can change only one portlet at a time (you can only click in one box!), and that all portlets would have to call `render` to generate their content again upon the result of the action. However, this is not to say that all portlets are not able to change at a given time.

Consider the following common example: a portal for *The Simpsons*. One of the portlets allows you to select the given Simpson character whose page you would like to view. Other portlets contain character information, recent appearances, greatest quotes, and so on. When you select a new character, you would change the state of that character selector portlet through the `processAction` method. In that method, though, you would edit a given shared attribute that specifies which character's page you are on, which would cause all of the portlets to `render` themselves for that character when you invoked their `render` methods.

*Note one exception to when a portlet's **`render`** method is called, and that is when the portlet's content is cached. The Portlet API allows containers to choose to use a cached copy of the content, instead of calling **`render`**. Portlet containers are not required to provide a cache facility, but the spec provides for an expiration cache facility, which is configured in the portlet application deployment descriptor. The deployer provides an expiration-cache element into which the user specifies the number of seconds to cache (or -1 for cache that won't expire).*

*The cache is per client per portlet, and cannot be shared across client requests. Of course, a developer could implement his or her own portlet managed cache in the **`render`** method, storing some commonly requested data in the **`PortletContext`**.*

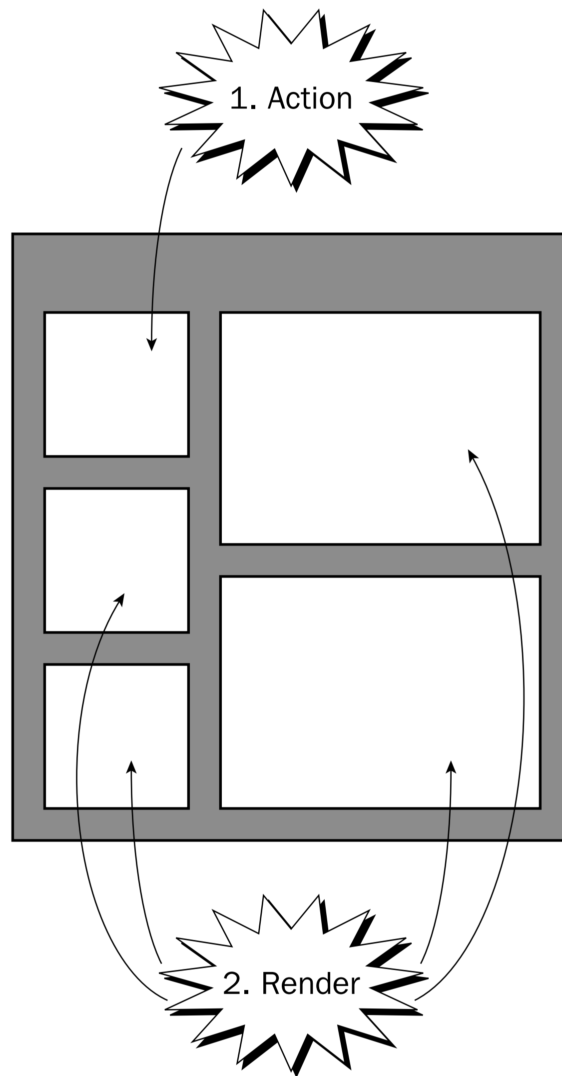


Figure 1.3

ActionRequest

As previously mentioned in the discussion of portlet request handling, action requests handle changing the state of a portlet based on the action request parameters. This is done using the `processAction` method, which takes an `ActionRequest` and `ActionResponse` object as parameters.

The `ActionRequest` object, similar to a `ServletRequest` object, provides the following:

- ❑ The action request parameters
- ❑ The portlet mode

- The portlet session
- The window state
- The portlet preferences object
- The portal context

To change the portlet mode or window state, you call the appropriate method on the `ActionResponse` object. The change becomes evident when the `render` method is called subsequent to the end of processing in the `processAction` method. You can also pass `render` parameters using the `ActionResponse` object.

RenderRequest

`RenderRequests` generate a fragment from the portlet's current state. The `RenderRequest` object provides access to the following:

- The render request parameters
- The portlet mode
- The portlet session
- The window state
- The portlet preferences object

There is also an accompanying `RenderResponse` method, which provides the vehicle needed to render content. You can call `getOutputStream` or `getWriter` as you do in a servlet, or you can dispatch the content generation to a servlet or JSP. There is more detail on this technique later in the chapter, in the section "Calling JSPs and Servlets."

> **The request and response objects *are not thread-safe*. This means that a developer should avoid sharing references to them with other threads of execution. Most developers will not run into this problem, but remember this tidbit next time you decide to try something unconventional.**

GenericPortlet

The `GenericPortlet` class is an abstract implementation of the `Portlet` interface. This is the most common way most users will write portlets — by extending this class. The `GenericPortlet` class extends the `render` method by setting the portlet title, and then calling its own `doDispatch` method, which in turn, determines the mode of the Portlet, and calls its appropriate method: `doEdit` for `EDIT`, `doView` for `VIEW`, and so on. There is more discussion on portlet modes later. The following code describes a class that extends `GenericPortlet`:

```
package org.opensourceportals.samples;
import java.io.IOException;
import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
```

```
import javax.portlet.GenericPortlet;
import javax.portlet.PortletException;
import javax.portlet.PortletMode;
import javax.portlet.PortletRequestDispatcher;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
/**
 * @author Clay Richardson
 * ExamplePortlet is a basic example of writing
 * a portlet by extending GenericPortlet
 *
 */
public class ExamplePortlet extends GenericPortlet {
  /*
   * This method overrides the doEdit of GenericPortlet
   * This is called to provide the markup to be rendered when the
   * portlet mode is PortletMode.EDIT
   * <p>
   * In this case, we will dispatch the method to a JSP
   * located in the portlet root directory called "edit.jsp"
   */
  protected void doEdit(
    RenderRequest request,
    RenderResponse response)
    throws PortletException, IOException {
    PortletRequestDispatcher prd =
      getPortletContext().getRequestDispatcher("/edit.jsp");
    prd.include(request, response);
  }
```

We declare our `ExamplePortlet`, having it extend GenericPortlet. In here, we also override the `doEdit` method, which handles rendering when the portlet is in `EDIT` mode.

```
  /*
   * This method overrides the doHelp of GenericPortlet
   * This is called to provide the markup to be rendered when the
   * portlet mode is PortletMode.HELP
   * <p>
   * In this case, we will dispatch the method to a JSP
   * located in the portlet root directory called "help.jsp"
   */
  protected void doHelp(
    RenderRequest request,
    RenderResponse response)
    throws PortletException, IOException {
    PortletRequestDispatcher prd =
      getPortletContext().getRequestDispatcher("/help.jsp");
    prd.include(request, response);
  }

  /*
   * This method overrides the doEdit of GenericPortlet
   * This is called to provide the markup to be rendered when the
   * portlet mode is PortletMode.VIEW
```

```
     * <p>
     * In this case, we will dispatch the method to a JSP
     * located in the portlet root directory called "view.jsp"
     */
    protected void doView(RenderRequest request,
      RenderResponse response)
      throws PortletException, IOException {
      PortletRequestDispatcher prd =
        getPortletContext().getRequestDispatcher("/view.jsp");
      prd.include(request, response);
    }
```

Similarly, we provide the behavior required to render the portlet when it is in its `HELP` and `VIEW` modes.

```
    /* This method was overriden to specify
     * the title programmatically
     * This may be useful if you are going to
     * have parameters in your title like:
     * "News on 9/11/2001"
     */
    protected String getTitle(RenderRequest request) {
      return "Example Portlet";
    }

    /* This method is the meat of the portlet
     * manipulations of the portlet's state are done
     * through this method.
     *
     * For simplicity sake, we will parse a param
     * that indicates the portlet mode to which the
     * portlet should be set.
     *
     */
    public void processAction(ActionRequest request,
      ActionResponse response)
      throws PortletException, IOException {

        PortletMode mode =
          new PortletMode(request.getParameter("mode"));

        response.setPortletMode(mode);
    }

  }
```

Finally, we specify overriding the `getTitle` method, allowing for more complex logic in rendering the title (such as displaying the current date) rather than displaying the static title declared in the deployment descriptor. We also handle the `processAction` method, which is responsible for the behavior in response to an `ActionRequest`.

The preceding code shows a basic implementation of a portlet by writing a class that extends `GenericPortlet`. This portlet doesn't do much beyond dispatch to other JSPs based on its mode (and set its name programmatically), but you see the crux of implementing a portlet.

Other Elements of the Java Portlet API

Now that you have examined the high-level concepts of the Portlet API, this section addresses the lower-level components within the specification, providing a portlet developer's perspective on the internals of the specification, highlighting important concepts and potential pitfalls.

PortletConfig

When a portlet is initialized, it needs access to the initialization parameters and other configuration information. The `PortletConfig` object provides these. In addition to `init` parameters, the `PortletConfig` object can also expose a `ResourceBundle` for the portlet.

The `ResourceBundle` contains certain fields required by the specification, including title, short title, and keywords. A `ResourceBundle` allows for easier localization of your portlet application.

You can specify the `ResourceBundle` inline in the portlet application deployment descriptor, as follows:

```
<portlet>
  ...
  <portlet-info>
    <title>Homer's D'oh a Day Portlet</title>
    <short-title>doh</short-title>
    <keywords>Simpsons, Homer Simpson, Entertainment</keywords>
  </portlet-info>
  ...
</portlet>
```

Alternatively, you can specify a reference to a `ResourceBundle` this way:

```
<portlet>
  ...
  <portlet-info>
    <resource-bundle>com.somedomainname.HomerPortlet</resource-bundle>
  </portlet-info>
  ...
</portlet>
```

Whichever method you use (the first is better mostly for applications with minimal localization requirements), the net effect for the developer is the same. These properties are always created in a `Resource Bundle` and made available through the `PortletConfig` object.

PortletURL

When building portlet content, it is necessary to build URLs that provide the capability to call the portal. This is the foundation of making the portal interactive. In order to allow for the proper creation of a `PortletURL`, there are two implementations: `ActionURL` and `RenderURL`. Both of these are created from the `RequestResponse` interface using the `createActionURL` and `createResponseURL` methods, respectively. The `ActionURL` provides the capability to issue action requests on the portal, to do things such as change portlet mode, change window state, submit a form, and so on. The `RenderURL` provides the capability to skip the portlet's `processAction` method and merely invoke the `render` method, passing `render` parameters to control presentation.

04 469513 Ch01.qxd 1/16/04 11:04 AM Page 15

Portlet developers should use the `PortletURL` objects (or their accompanying tag libraries) instead of directly manipulating HTTP query strings. The corollary to this is that developers should not use `GET` in HTML forms. This is because portals may encode internal state parameters in the `PortletURL`.

You will find several methods of the `PortletURL` interface interesting:

- `setSecure` — provides the capability to specify whether the URL should use HTTPS or not. If it is not used, it continues with whatever the current request specified. Therefore, you do not have to specify it repeatedly.
- `setWindowState` — enables you to change the window state of the portlet.
- `addParameter` — adds parameters to the URL.
- `toString` — provides a string representation of the URL. Note that it is not guaranteed to be a valid URL, as the portal may use tokens for URL rewriting.
- `setPortletMode` — enables you to set the portlet's mode.

Portlet Modes

A **portlet mode** represents a functional state of a portlet. This is used by the portlet to determine how to manage a `render` request. That is, depending on the mode, the portlet will render different markup. Portlets are able to change their mode as part of processing action requests. In addition, a portlet can be configured with different modes available and further restrict its availability based on role. The following table describes the standard portlet modes defined in the Portlet API.

Mode	Description
`VIEW`	Generates markup visualizing the portlet state and properties. Developers implement `doView` of `GenericPortlet` to provide this functionality.
`EDIT`	Produces markup to enable modification of portlet properties. Developers implement `doEdit` of `GenericPortlet` to provide this functionality.
`HELP`	Provides help text for the portlet. Developers implement `doHelp` of `GenericPortlet` to provide this functionality.

A portal can also provide custom portlet modes. Note that this is **portal** dependent, not **portlet** dependent. Therefore, if a portlet implements additional portlet modes, they will not be portable between various portlet containers. The portlet needs to override the `doDispatch` method to call the appropriate `render` method. For example, if you define a portlet mode called "SPECIAL", the `doDispatch` method would call its `render` method, which by convention would be `doSpecial`. Of course, because you are implementing the method, you could call the method anything you want.

Also note that you can specify which types of markup are available for a given portlet mode. Additional information on configuring portal modes is presented in a later section, "Portlet Application Deployment Descriptor."

Window States

The **window state** indicates to the portlet how much space is being allocated to it. This enables the portlet to modify its rendering to suit that window state. The following table contains the window states specified by the Portlet API.

State	Definition
NORMAL	The portlet will share the screen with other portlets. This means that the portlet should limit its markup.
MINIMIZED	The portlet should provide little or no output.
MAXIMIZED	The portlet doesn't share the screen with other portlets; thus, the portlet is not limited in its markup.

Much like portlet modes, a portal can define custom window states, which must also be configured in the portlet deployment descriptor.

Portlet Context

`PortletContext` is a wrapper object for your portlet application. There is one `PortletContext` object per portlet application. The `PortletContext` provides the following:

- Accesses initialization variables
- Gets and sets context attributes
- Logs events
- Gets application resources (such as images, XML files, and so on)
- Obtains a request dispatcher to leverage servlets and JSPs in the portlet

Portal Context

A portlet can get a reference to the portal in which it is running through the `PortalContext` object. Calling the `getPortalContext` method of the `PortletRequest` object will return the `PortalContext`.

The `PortalContext` provides the following:

- The name of the portal, through `getPortalInfo`
- Portal properties, through `getProperty` and `getPropertyNames`
- The supported portlet modes, through `getSupportedPortletModes`
- The supported window states, through `getSupportedWindowStates`

Portlet Preferences

In order to perform customization or personalization of your portlet, you need some way to vary certain parameters of the portlet. These are called **portlet preferences**. The classic portlet example is the weather

portlet, and following that example, you could have a preference called "cities," with values representing the zip codes of the cities for which you want the weather. Note that preferences are only meant for configuring the portlet, so maintaining the list of all cities available in a preference would be an inappropriate use of preferences. Thinking of them in an object-oriented programming sense, they would be attributes of the portlet itself.

Preferences are manipulated through an object that implements `PortletPreferences`. All preference values are stored as String arrays, so you would not have the capability to store complex objects as you do with request or session attributes, nor do you have the advantage of declaring the type, as you would with environment entries. Therefore, if you are storing numbers as preferences, you will need to do conversions yourself.

Specifically, the `PortletPreferences` interface provides the following:

- `getNames` — This returns an Enumeration of the names of the available preferences.
- `getValue` — You pass it the name of the preference you are looking for, along with a default value (in case it isn't found), and it returns the first element of the array of that preference's values.
- `getValues` — You pass it the name of the preference you want and a String array of default values and it returns a String array of its values.
- `setValue` — You pass it the name of the preference and the value of that preference, and it sets it to a single-element String array containing that value. This method throws an `UnmodifiableException` if the preference cannot be modified.
- `setValues` — You pass it the name of the preference and a String array representing the values for that name, and it sets the preference values. This method throws an `UnmodifiableException` if the preference cannot be modified.
- `isReadOnly` — You pass it the name of a preference and it returns a Boolean indicating whether the preference can be modified.
- `reset` — You pass it the name of the preference and it will restore the default; if there is no default, it will delete the preference.
- `store` — This stores the preferences. Because this can only be done from within `processAction`, it will throw an `IllegalStateException` if it is done from within a `render` invocation. The method will also throw a `ValidatorException` if it fails validation.
- `getMap` — This returns a Map of the preferences. The Map consists of String keys and a `String[]` for values. This map is also immutable (cannot be changed).

> **Note two important things to understand about the `store` method. First, it is an atomic transaction. Therefore, all of the changes must succeed or none of them will succeed. This is critical to understand if you have an enormous preference list for your portlet and you don't do a tremendous amount of validation of the input. Second, you could get stung by concurrent writes to the preference store. The critical message here is that you should view your preferences as one distinct entity and not a collection of independent parameters accessible through a common interface.**

The following code is an example of retrieving and setting preferences:

```
    try {
    PortletPreferences myPrefs = request.getPreferences();
    String [] cities =
      myPrefs.getValues("cities",
                        new String[] {"20112","90210"});
    for (int i=0; i < cities.length; i++) {
      System.out.println(cities[i]);
    }
    String [] newCities = new String[] {"20169","22124"};
    myPrefs.setValues("cities",newCities);
    myPrefs.store();
  } catch (ValidatorException ve) {
    System.out.println("Preferences did not validate.");
  } catch (UnmodifiableException ume) {
    System.out.println("The preference is not modifiable");
  } catch (IllegalStateException ise) {
    System.out.println("You cannot be in render!");
  }
```

Developers can define custom classes to provide validation of preferences. These classes implement the `PreferencesValidator` interface and must provide their validation capability in a thread-safe manner. Implementing a `PreferencesValidator` is very simple: There is only one method, `validate`, which takes a `PortletPreferences` object and returns nothing. Simply throw a `ValidatorException` if any of the values doesn't meet the logic you define.

Sessions

Because portals are built upon the request-response paradigm (and predominantly upon HTTP), there has to be some mechanism available to maintain state across invocations. For example, it isn't sensible to authenticate users with every request. There are several techniques to manage sessions, with cookies and URL rewriting being two of the most popular in Web applications (cookies are used under the hood by many servlet containers to implement the `HTTPSession`).

Sessions are critical to portlet development, but there are many ways to implement them, so the Portlet API provides the `PortletSession` interface. When a client makes a request, the server sends a session identifier in the response. If the client wants to join the session, the client provides that session identifier with their next request.

The `PortletSession` can be used to hold attributes. `PortletSession` operates much like `HTTPSession` in this regard, providing the capability to store key-value pairs, with arbitrary objects. There is one major difference. The `PortletSession` has two different scopes:

- ❑ `APPLICATION_SCOPE` is very similar to the `HTTPSession` scope. An object placed in this scope is available to all the portlets within the session.
- ❑ `PORTLET_SCOPE` refers to when an object is available to only that portlet.

`PORTLET_SCOPE` is unique in that it provides a namespace for the given attribute. For example, an attribute called `city.name` would be stored in `javax.portlet.p.47?city.name`. ("47" is an internally assigned

identification number). This attribute name prevents namespace collision with other portlets storing their session variables with similar names.

Despite having a special system for naming its attributes, `PORTLET_SCOPE` doesn't protect its attributes from other Web components. In addition, the namespace application is done completely under the hood. You just call `getAttribute` and `setAttribute` specifying `PORTLET_SCOPE` and the namespace conversion is done by the `PortletSession` object. To make it even more convenient, other Web components can receive this feature through the `PortletSessionUtil.decodeAttribute` method by passing the simple attribute name, such as "city.name".

Calling JSPs and Servlets

Because portlet applications are a complementary extension to Web applications, there must be a mechanism to include Java Server Pages and servlets in a portlet. Upon first examination of the `GenericPortlet` class, many Web developers cringe and think, "Oh, no! We are back to the old servlet days of embedding markup!" However, much like servlet developers found using a `RequestDispatcher` helpful in avoiding the "all your eggs in one basket" problem of placing all of your markup in your servlet, or all of your Java code in a JSP, a portlet developer can use a `PortletRequestDispatcher`.

When implementing the `render` method of the `Portlet` interface or, more likely, implementing one of the "do" methods of the `GenericPortlet` class (for example, `doView`, `doEdit`, and so on), the developer can use a `PortletRequestDispatcher` as follows:

```
String reqPath = "/calView.jsp";
PortletRequestDispatcher prd =   portContext.getRequestDispatcher(reqPath);
prd.include(req, resp);
```

In this case, we have specified our JSP, `calView.jsp`, which is located in the root of the portlet application, which we refer to with a leading slash. You must always start the path with a leading slash, and provide a path from the `PortletContext` root (usually the root directory of your portlet application). From there, you get a `PortletRequestDispatcher` (prd) from your `PortletContext` (`portContext`). Then you pass your `RenderRequest` (req) and your `RenderResponse` (resp) as parameters to the `include` method of the `PortletRequestDispatcher` (prd).

Similarly, we can call a servlet by its declared name (in the Web application deployment descriptor, also known as `web.xml`). For example, we might have specified a servlet such as the following in the `web.xml`:

```
<servlet>
  <servlet-name>chart</servlet-name>
  <servlet-class>org.opensourceportals.ChartServlet</servlet-class>
  <load-on-startup>0</load-on-startup>
</servlet>
```

Because we have named our servlet "chart," we can specify it as follows:

```
String reqName = "chart";
PortletRequestDispatcher prd = portContext.getNamedDispatcher(reqName);
prd.include(req, resp);
```

04 469513 Ch01.qxd 1/16/04 11:04 AM Page 20

This time we used the getNamedDispatcher method with the name of our servlet in order to get a PortletRequestDispatcher. This is another important point to consider if you choose to do the following:

```
String reqPath = "/calView.jsp?user=Jennifer";
PortletRequestDispatcher prd = portContext.getRequestDispatcher(reqPath);
prd.include(req, resp);
```

Because the parameter user is specified in the query string, it will take precedence over any other render parameters named user being passed to the JSP. You probably will not encounter this problem, but it is something to keep in the back of your mind in case you run into crazy behaviors: specifying a query string takes precedence over other parameters.

There are restrictions on the use of HttpServletRequest. These methods are not available to the included servlet or JSP:

- getProtocol
- getRemoteAddr
- getRemoteHost
- getRealPath
- getRequestURL
- getCharacterEncoding
- setCharacterEncoding
- getContentType
- getInputStream
- getReader
- getContentLength

These methods will all return null (getContentLength returns zero) if invoked from a servlet or JSP that has been included by a PortletRequestDispatcher. Depending on how you create your PortletRequestDispatcher, you may not have access to other methods. These additional methods are not available to servlets or JSPs accessed from a PortletRequestDispatcher created by calling getNamedDispatcher:

- getPathInfo
- getPathTranslated
- getServletPath
- getRequestURI
- getQueryString

The reason why the preceding methods would be unavailable through getNamedDispatcher is pretty simple: Because you didn't use a path for your request, there is no data with which these fields can be

populated. Likewise, `HttpServletResponse` has restrictions on what is accessible. The unavailable methods of `HttpServletResponse` are:

- `encodeRedirectURL`
- `encodeRedirectUrl`
- `setContentType`
- `setContentLength`
- `setLocale`
- `sendRedirect`
- `sendError`
- `addCookie`
- `setDateHeader`
- `addDateHeader`
- `setHeader`
- `addHeader`
- `setIntHeader`
- `addIntHeader`
- `setStatus`
- `containsHeader`

The encode methods always return null, and `containsHeader` always returns false, but the remainder will simply do nothing. This could be a source of great frustration if you are not careful as a developer, because it simply will not work and will provide no notice.

> *The Java Portlet Specification recommends against using the forward method of the* **`RequestDispatcher`** *of an included servlet or JSP. While this doesn't seem like a big deal, note that Apache's Struts Framework uses the* **`RequestDispatcher`** *forward method in its* **`ActionServlet`**. *Given the popularity of Struts as a Web application framework, this could make a significant impact on portal integration in many enterprises. This is not to say that it may not work anyway, but it is non-deterministic and should be carefully examined in your circumstance and tested with your portal implementation.*

Portlet Application Structure

Portlet applications are structured just like Web applications in that they have the following features:

- Can contain servlets, JSPs, Java classes, Java archives (JAR files) and other static files
- Are self-contained; all things in the Web application are packaged together in a common root
- Have a WEB-INF/classes directory to store standalone classes to be loaded by the application `classloader`
- Have a WEB-INF/lib directory to store Java Archives (JAR files) to be loaded by the application `classloader`

- Have a Web application deployment descriptor located at `WEB-INF/web.xml`
- Are packaged as Web archives (WAR files)

In addition to these features, the portlet application contains a portlet application deployment descriptor, located at `WEB-INF/portlet.xml`. This file is described in detail later in this chapter, in the section "Portlet Application Deployment Descriptor."

Security

Because security is a bigger matter than simply the requirements of the Portlet API, we defer the discussion on security in the Portlet API to Chapter 6.

CSS Style Definitions

In order to achieve a common and pluggable look and feel for portlets, the Java Portlet API defines a set of Cascading Stylesheets (CSS) styles that portlets should use in rendering their markup. By using a standard set of styles, portals can support *skins,* customized colors and fonts. These styles are meant to coincide with the OASIS Web Services for Remote Portlets standard.

In order to be complete, these style definitions are presented in the following table, as specified in Appendix C of the JSR 168 (Java Portlet API).

Attribute Name	Description
`portlet-font`	This is for normal, unaccented text used in a portlet. Size can be overridden using the `style` attribute with something such as "font-size:large".
`portlet-font-dim`	This is for suppressed text, essentially text that has been grayed out.
`portlet-msg-status`	This is used to represent text that is providing the current state of an operation in project, such as "*Please wait while data loads...*"
`portlet-msg-info`	Use this for informational text such as "Reminder: your username is your e-mail address."
`portlet-msg-error`	This styles messages such as "An unexpected error occurred, please contact the administrator."
`portlet-msg-alert`	This is for warnings, such as "Could not get open database connection, please try again in a couple of minutes."
`portlet-msg-success`	This relays messages when the submission was successful, such as "Your request was submitted."
`portlet-section-header`	Use this to render the table or section header.
`portlet-section-body`	This is to style the internals of a table cell.
`portlet-section-alternate`	When using a technique called *banding,* in which you provide alternate styles in between alternate rows, this style provides that capability.

04 469513 Ch01.qxd 1/16/04 11:04 AM Page 23

Attribute Name	Description
portlet-section-selected	This style is used for highlighting a particular set of cells for the user.
portlet-section-subheader	If you have subheadings in your table that you need to distinguish from your table header, you use this style.
portlet-section-footer	If you include a footer to your table, this style would be used for those cells.
portlet-section-text	This style is used for things that do not fit in the other style definitions.
portlet-form-label	This is to style the label for the whole form, such as “User Registration Form.”
portlet-form-input-field	This is for the text that a user enters into an input field.
portlet-form-button	This is for the text that appears on the face of a button.
portlet-icon-label	This styles text next to an application-specific icon, such as “Export.”
portlet-dlg-icon-label	This styles text next to a standard icon, such as “Cancel.”
portlet-form-field-label	This styles text that separates form fields (such as radio buttons).
portlet-form-field	This is used for labels of checkboxes, but not for input fields.
portlet-menu	This styles the menu itself (e.g., color).
portlet-menu-item	Use this to style an ordinary menu item that is not selected.
portlet-menu-item-selected	This is used to style an item that has been selected.
portlet-menu-item-hover	Use this to style an ordinary menu item when the mouse hovers over it.
portlet-menu-item-hover-selected	This is used to style a selected item when the mouse hovers over it.
portlet-menu-cascade-item	Use this to style an unselected menu item that has menu items nested underneath it.
portlet-menu-cascade-item-selected	Use this to style a selected menu item that has menu items nested underneath it.
portlet-menu-description	This styles text that is used to describe the menu.
portlet-menu-caption	This is used for the menu caption.

By using these styles, portlet developers ensure that their portlets can be reused in many portlet applications, and be consistent with WSRP. In addition, it enables developers to apply “skins” from other portlet applications to their portlet.

User Information Attributes

User attributes are used to create a profile for a user. They are meant to be standardized on the World Wide Web Consortium's (W3C) Platform for Privacy Preferences (P3P) 1.0 (`www.w3.org/TR/P3P/`). These attributes are also consistent with the efforts of the OASIS Web Services for Remote Portals standard. The following table lists the user attributes and their descriptions.

Attribute Name	Description
`user.bdate`	The user's birth date, expressed in milliseconds from January 1, 1970 at 00:00:00 Greenwich Mean Time
`user.gender`	The sex of the user
`user.employer`	The name of the user's employer
`user.department`	The department in which the user works
`user.jobtitle`	The user's job title
`user.name.prefix`	The prefix of the user's name (Mr., Mrs., Dr., etc.)
`user.name.given`	The user's given name (Jennifer, Alicia, etc.)
`user.name.family`	The user's last name (Richardson, Smith, Avondolio, etc.)
`user.name.middle`	The user's middle name (Anne, Clay, Trent, etc.)
`user.name.suffix`	The suffix following a user's name (Sr., Jr., III, etc.)
`user.name.nickname`	A user's nickname (Mojo, PAB, BP, etc.)
`user.home-info.postal.name`	The name that should appear at the top of a home address (for example, The Richardsons or William C. Richardson)
`user.home-info.postal.street`	The street address of the user's home (1600 Pennsylvania Avenue or 742 Evergreen Terrace)
`user.home-info.postal.city`	The postal city of the user's home (Haymarket, Manassas, etc.)
`user.home-info.postal.stateprov`	The state or province used in the user's home address (Virginia, British Columbia, etc.)
`user.home-info.postal.postalcode`	The user's home zip code (90210, etc.)
`user.home-info.postal.country`	The user's home country (United States of America, Canada, etc.)
`user.home-info.postal.organization`	A subheading in the address block, like "Finance Department," which doesn't make a lot of sense for a home address, but is included for completeness
`user.home-info.telecom.telephone.intcode`	The international access code for the user's home telephone (for example, 44 for the United Kingdom and 1 for the United States).
`user.home-info.telecom.telephone.loccode`	The user's home telephone area code (for example, 703, 818, etc.)

04 469513 Ch01.qxd 1/16/04 11:04 AM Page 25

Attribute Name	Description
user.home-info.telecom.telephone.number	The user's home telephone local number (for example, 555-1111, etc.)
user.home-info.telecom.telephone.ext	The user's home telephone extension, if they have one (for example, 745, 2918, etc.)
user.home-info.telecom.telephone.comment	Comments about the user's home telephone (optional)
user.home-info.telecom.fax.intcode	The international access code for the user's home fax (for example, 44 for the United Kingdom and 1 for the United States)
user.home-info.telecom.fax.loccode	The user's home fax area code 703, 818, etc.)
user.home-info.telecom.fax.number	The user's home fax local number (555-1111, etc.)
user.home-info.telecom.fax.ext	The user's home fax extension, if they have one (745, 2918, etc.)
user.home-info.telecom.fax.comment	Comments about the user's home fax (optional)
user.home-info.telecom.mobile.intcode	The international access code for the user's home mobile telephone (for example, 44 for the United Kingdom and 1 for the United States)
user.home-info.telecom.mobile.loccode	The user's home mobile telephone area code (for example, 703, 818, etc.)
user.home-info.telecom.mobile.number	The user's home mobile telephone local number (555-1111, etc.)
user.home-info.telecom.mobile.ext	The user's home mobile telephone extension, if they have one (for example, 745, 2918, etc.)
user.home-info.telecom.mobile.comment	Comments about the user's home mobile telephone (optional)
user.home-info.telecom.pager.intcode	The international access code for the user's home pager (for example, 44 for the United Kingdom and 1 for the United States)
user.home-info.telecom.pager.loccode	The user's home pager area code (for example, 703, 818, etc.)
user.home-info.telecom.pager.number	The user's home pager local number (for example, 555-1111, etc.)
user.home-info.telecom.pager.ext	The user's home pager extension, if they have one (for example, 745, 2918, etc.)
user.home-info.telecom.pager.comment	Comments about the user's home pager (optional)

Table continued on following page

Attribute Name	Description
`user.home-info.online.email`	The user's home e-mail address
`user.home-info.online.uri`	The user's home Web page
`user.business-info.postal.name`	The name that should appear at the top of a work address (for example, Sun Microsystems or XYZ, Inc., etc.)
`user.business-info.postal.street`	The street address of the user's work (for example, 1600 Pennsylvania Avenue or 742 Evergreen Terrace)
`user.business-info.postal.city`	The postal city of the user's work (for example, Haymarket, Manassas, etc.)
`user.business-info.postal.stateprov`	The state or province used in the user's work address (for example, Virginia, British Columbia, etc.)
`user.business-info.postal.postalcode`	The user's work zip code (for example, 90210)
`user.business-info.postal.country`	The user's work country (for example, United States of America, Canada, etc.)
`user.business-info.postal.organization`	A subheading in the address block, like "Finance Department"
`user.business-info.telecom.telephone.intcode`	The international access code for the user's work telephone (for example, 44 for the United Kingdom and 1 for the United States)
`user.business-info.telecom.telephone.loccode`	The user's work telephone area code (for example, 703, 818, etc.)
`user.business-info.telecom.telephone.number`	The user's work telephone local number (555-1111, etc.)
`user.business-info.telecom.telephone.ext`	The user's work telephone extension, if they have one (for example, 745, 2918, etc.)
`user.business-info.telecom.telephone.comment`	Comments about the user's work telephone (optional)
`user.business-info.telecom.fax.intcode`	The international access code for the user's work fax (for example, 44 for the United Kingdom and 1 for the United States)
`user.business-info.telecom.fax.loccode`	The user's work fax area code (for example, 703, 818, etc.)
`user.business-info.telecom.fax.number`	The user's work fax local number (for example, 555-1111, etc.)
`user.business-info.telecom.fax.ext`	The user's work fax extension, if they have one (for example, 745, 2918, etc.)
`user.business-info.telecom.fax.comment`	Comments about the user's work fax (optional)

Attribute Name	Description
`user.business-info.telecom.mobile.intcode`	The international access code for the user's work mobile telephone (for example, 44 for the United Kingdom and 1 for the United States)
`user.business-info.telecom.mobile.loccode`	The user's work mobile telephone area code (for example, 703, 818, etc.)
`user.business-info.telecom.mobile.number`	The user's work mobile telephone local number (555-1111, etc.)
`user.business-info.telecom.mobile.ext`	The user's work mobile telephone extension, if they have one (for example, 745, 2918, etc.)
`user.business-info.telecom.mobile.comment`	Comments about the user's work mobile telephone (optional)
`user.business-info.telecom.pager.intcode`	The international access code for the user's work pager (for example, 44 for the United Kingdom and 1 for the United States)
`user.business-info.telecom.pager.loccode`	The user's work pager area code (for example, 703, 818, etc.)
`user.business-info.telecom.pager.number`	The user's work pager local number (for example, 555-1111, etc.)
`user.business-info.telecom.pager.ext`	The user's work pager extension, if they have one (for example, 745, 2918, etc.)
`user.business-info.telecom.pager.comment`	Comments about the user's work pager (optional)
`user.business-info.online.email`	The user's work e-mail address
`user.business-info.online.uri`	The user's work Web page

As you can see, the attributes are a bit repetitive, but offer a breadth of options to you as a developer in terms of which user attributes you need to use for your portlet application.

However, it is not sufficient just to use these in your application. Your deployment descriptor must declare which of these are used by a portlet application, and the deployer needs to map them against the related ones available in the targeted portal server. This is where using the standard attributes comes in handy, as it will greatly reduce, if not eliminate, the amount of mapping necessary to deploy your application.

Presuming you have done all of the appropriate mappings (and the section "Portlet Application Deployment Descriptor" discusses how to do this in your `portlet.xml`), you can gain access to user attributes such as the following:

```
Map userdata = (Map) request.getAttribute(PortletRequest.USER_INFO);
String workEmail =
(String) request.getAttribute("user.business-info.online.email");
```

04 469513 Ch01.qxd 1/16/04 11:04 AM Page 28

You grab a Map of the deployed user attributes by getting that attribute from the `PortletRequest`. Then, you simply look up the appropriate value; in this case, the user's work e-mail (stored under "user.business-info.online.email").

User attributes are very important in deploying personalized portlet solutions. Be sure to familiarize yourself with these attributes.

Portlet Tag Library

The Portlet JSP Tag Library gives developers the capability to reference Portal components from within a JSP page. The following table explains the tags:

Tag Name	Purpose
`defineObjects`	This tag declares three objects within the JSP page: `RenderRequest renderRequest`, `RenderResponse renderResponse`, and `PortletConfig portletConfig`.
`actionURL`	This tag builds action URLs that point to the current portlet. This tag is nested with `param` tags in order to pass the appropriate parameters in the action URL.
`renderURL`	This tag builds render URLs. It also is nested with `param` tags.
`namespace`	This tag provides a unique name based on the current portlet in order to avoid conflicts in variable and function names when all of the portlet fragments are consolidated into a portal page.
`param`	This tag gives the name and value of a parameter. It is nested inside either the `actionURL` or `renderURL` tags.

Portlet Deployment

This section describes the portlet application deployment descriptor (`portlet.xml`), by dissecting a sample and explaining the various sections of the descriptor piece by piece.

Portlet Application Deployment Descriptor

In order to understand the portlet application deployment descriptor well, you should examine each part.

If you have used XML very much, you will find this first section unremarkable. The only thing worth noting here is that the `version` attribute is required to determine which version of the specification is in effect.

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app  xmlns="http://java.sun.com/xml/ns/portlet"
  version="1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://java.sun.com/xml/ns/portlet-app_1_0.xsd">
```

Portlet Declarations

This is the first part of the declaration of one portlet. In this, we give it a description, a local name, a display name, and a class name. The description and display name are meant to make it more human-friendly, while the local name and the class actually provide the nuts and bolts required to programmatically load and reference the portlet. The local name must, of course, be unique within the portlet application.

```
  <portlet>
    <description>Example of creating a portlet</description>
    <portlet-name>ExamplePortlet</portlet-name>
    <display-name>Example Portlet</display-name>
    <portlet-class>org.opensourceportals.samples.ExamplePortlet</portlet-class>
```

The `expiration-cache` tag represents the number of seconds a portlet is kept in cache. The expiration cache value of `-1` means that the portlet will always be kept in cache. If the value were zero, it would be just the opposite — never cached.

```
    <expiration-cache>-1</expiration-cache>
```

This next section declares the supported portlet modes and mime types. For each mime type, the supported portlet modes are listed. Wild cards (*) are acceptable in describing a mime type. For example, all text-based mime types could be specified as `text/*`, and all mime types could be expressed as `*/*`. In this case, the portlet supports traditional HTML with three different modes available: `VIEW`, `EDIT`, and `HELP`. However, it supports the `VIEW` mode only when the content type is the Wireless Markup Language (WML). Each of these definitions must be unique — that is, you cannot define multiple supports blocks for the same MIME type.

```
    <supports>
      <mime-type>text/html</mime-type>
      <portlet-mode>view</portlet-mode>
      <portlet-mode>edit</portlet-mode>
      <portlet-mode>help</portlet-mode>
    </supports>
    <supports>
      <mime-type>text/wml</mime-type>
      <portlet-mode>view</portlet-mode>
    </supports>
```

This portlet provides only one `Locale`, English, for its internationalization support. It could support many locales, and it would list them all here. For more information, examine Java's internationalization support.

```
    <supported-locale>EN</supported-locale>
```

This element (`<portlet-info>`) provides the metadata about the portlet. The title element represents the title that will be displayed on the portlet. The short title provides an abbreviated way of referencing the portlet, such as in a management console. Similarly, the keywords are meant to provide context about the subject of the portlet. This portlet information could have also been referenced in a `ResourceBundle`, with only the name of the `ResourceBundle` specified here.

```
        <portlet-info>
          <title>Pre-configured title</title>
          <short-title>Example</short-title>
          <keywords>JSR 168, Portlet API, Example, Simple</keywords>
        </portlet-info>
```

This section shows the portlet preferences. Of course, the preferences you define must be unique. The `index-location` preference is read-only, so it cannot be changed programmatically; instead, it must be changed here. The second preference, `sites-to-crawl`, shows how multiple values can be specified for a preference. The last preference, `crawl-depth` represents a number that must be converted through `Integer.parseInt`, because preferences are all retrieved as either a String or a String array. The `preferences-validator` element specifies a class that is used to validate these `portlet-preferences`. In this case, the validator would confirm that the `sites-to-crawl` are valid and that the `crawl-depth` is less than five (to keep the crawling time down).

```
        <portlet-preferences>
          <preference>
            <name>index-location</name>
            <value>/opt/lucene/index</value>
            <read-only>true</read-only>
          </preference>
          <preference>
            <name>sites-to-crawl</name>
            <value>http://jakarta.apache.org</value>
            <value>http://java.sun.com</value>
            <value>http://onjava.com</value>
          </preference>
          <preference>
            <name>crawl-depth</name>
            <value>2</value>
          </preference>
          <preferences-validator>
            com.opensourceportals.validator.CrawlValidator
          </preferences-validator>
        </portlet-preferences>
```

Here we have the relevant security references for the portlet. The `role-name` element specifies the parameter that should be passed to the `request.isUserInRole(String roleName)` method. The `role-link` is the role, defined in the portlet application's `web.xml`, into which the user should be mapped.

```
        <security-role-ref>
          <role-name>ADMIN</role-name>
          <role-link>administrator</role-link>
        </security-role-ref>
```

This closes the definition of the portlet. This can be repeated as many times as necessary to define all of the portlets within your portlet application. Of course, each of your portlets must have a unique name within this portlet application.

```
      </portlet>
```

Portlet Customization Declarations

This defines a custom portlet mode called MY_MODE. Of course, whenever an application defines a custom portlet mode, it must not only be available through the targeted portal server, but also needs to have portlets that actually use the mode (while programming defensively enough to avoid breaking in unsupported portal servers).

```
  <custom-portlet-mode>
      <description xml:lang="EN">Custom portlet mode MY_MODE</description>
      <portlet-mode>MY_MODE</portlet-mode>
    </custom-portlet-mode>
```

LEFT-SIDE is defined here as a custom window state. Just as with custom portlet modes, custom window states can cause problems with your application's portability. In addition, like custom portlet modes, you can define multiples in the same application, but they must have unique names.

```
    <custom-window-state>
      <description xml:lang="EN">Docked into the left side</description>
      <window-state>LEFT-SIDE</window-state>
    </custom-window-state>
```

User Attributes and Security Constraints

The following code defines a user attribute called user.business-info.online.email that refers to a user's business e-mail address. Typically, these attributes are mapping from the portal server's personalization database, which is why using standard names can ease the integration of user attributes. Developers should use user attributes from the W3C P3P standard, as specified in the Portlet API. This should ensure that they are unique (as required in the XML Schema for the portlet.xml) and reusable (standards-compliant).

```
    <user-attribute>
      <description>P3P attribute for work e-mail address</description>
      <name>user.business-info.online.email</name>
    </user-attribute>
```

This is the security constraint for this portlet application. It lists the name of the portlet, ExamplePortlet, and also specifies a user data constraint of INTEGRAL. User data constraints define the guarantees specifying how the portlet communicates with the user. Three values are allowed here: NONE, INTEGRAL, and CONFIDENTIAL. NONE means that there are no guarantees about the transmission of data between the portlet application and the user. INTEGRAL specifies that the data must be checked to ensure that it has not been manipulated (no added or removed information from the message). CONFIDENTIAL requires that the data cannot be read by anyone other than the user. Typically, Secure Sockets Layer (SSL) is used to provide both INTEGRAL and CONFIDENTIAL constraints.

```
    <security-constraint>
      <portlet-collection>
        <portlet-name>ExamplePortlet</portlet-name>
      </portlet-collection>
      <user-data-constraint>
        <transport-guarantee>INTEGRAL</transport-guarantee>
      </user-data-constraint>
    </security-constraint>
```

This defines the end of the portlet application.

```
</portlet-app>
```

Building a Portlet

Now, let's work through a complete example of a portlet. This portlet is a little more complex than our previous example, but it is not as involved as the portlets found in Chapter 9, which covers building portlet applications.

The first thing to do in building the portlet is to build the portlet class. Most portlet classes simply extend `GenericPortlet`, as shown in the following code:

```
/*
 * This class demonstrates a basic search portlet
 * using the Jakarta Lucene search API.
 *
 */
package org.opensourceportals.samples;
import java.io.IOException;
import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.GenericPortlet;
import javax.portlet.PortletException;
import javax.portlet.PortletMode;
import javax.portlet.PortletPreferences;
import javax.portlet.PortletRequestDispatcher;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
import javax.portlet.ValidatorException;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.Hits;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.Searcher;
/**
 * @author Clay Richardson
 *
 */
public class LucenePortlet extends GenericPortlet {
  /*
   * This method overrides the doEdit of GenericPortlet
   * This is called to provide the markup to be rendered when the
   * portlet mode is PortletMode.EDIT
   * <p>
   * This mode should always show a form to change the indexPath
   * preference, indicating where the index is stored.
   */
  protected void doEdit(
```

```
    RenderRequest request,
    RenderResponse response)
    throws PortletException, IOException {
    PortletRequestDispatcher prd =
      getPortletContext().getRequestDispatcher("/searchEdit.jsp");
    prd.include(request, response);
  }
```

We declare our `LucenePortlet`, which extends `GenericPortlet`. When we are in `EDIT` mode, we display the `searchEdit.jsp`.

```
  /*
   * This method overrides the doHelp of GenericPortlet
   * This is called to provide the markup to be rendered when the
   * portlet mode is PortletMode.HELP
   * <p>
   * This method provides help information by dispatching
   * the request to "help.jsp"
   */
  protected void doHelp(
    RenderRequest request,
    RenderResponse response)
    throws PortletException, IOException {
    PortletRequestDispatcher prd =
      getPortletContext().getRequestDispatcher("/help.jsp");
    prd.include(request, response);
  }
```

And when we are in `HELP` mode, we display `help.jsp`.

```
  /*
   * This method overrides the doEdit of GenericPortlet
   * This is called to provide the markup to be rendered when the
   * portlet mode is PortletMode.VIEW
   * <p>
   * In this case, we will dispatch the method to a JSP
   * located in the portlet root directory called "view.jsp"
   */
  protected void doView(
    RenderRequest request,
    RenderResponse response)
    throws PortletException, IOException {
    String queryMode = request.getParameter("queryMode");
    String forwardString = "/searchView.jsp";
    if (queryMode != null) {
      forwardString = "/searchResults.jsp";
    }
    PortletRequestDispatcher prd =
      getPortletContext().getRequestDispatcher(forwardString);
    prd.include(request, response);
  }
```

Rendering the `VIEW` mode depends on whether we passed in a query mode `RenderRequest` parameter. If there is a `queryMode` parameter, we display the results; if not, we display the search box.

```
/* This method is overriden to specify
 * the title programmatically
 */
protected String getTitle(RenderRequest request) {
  return "Lucene Portlet";
}
/* This method is the meat of the portlet
 * manipulations of the portlet's state are done
 * through this method.
 * <p>
 * There are really two types of actions for this portlet
 * depending on which mode we are in.  In the VIEW mode,
 * we have an action that searches a Lucene Index and then
 * places the hits in a request attribute.  In the EDIT mode,
 * we allow users to change the location of the Lucene
 * index.
 *
 */
public void processAction(ActionRequest aReq, ActionResponse aRes)
  throws PortletException, IOException {
  PortletMode mode = new PortletMode(aReq.getParameter("mode"));
  aRes.setPortletMode(mode);
  if (mode.equals(PortletMode.VIEW)) {
    PortletPreferences prefs = aReq.getPreferences();
    String indexPath = prefs.getValue("indexPath", null);
    Searcher searcher = new IndexSearcher(indexPath);
    Analyzer analyzer = new StandardAnalyzer();
    StringBuffer queryBuffer = new StringBuffer();
    String searchString = aReq.getParameter("searchString");
    if (searchString != null) {
      queryBuffer.append(searchString);
      String line = queryBuffer.toString();
      try {
        Query query = QueryParser.parse(line, "contents", analyzer);
        Hits hits = searcher.search(query);
        aReq.setAttribute("hits", hits);
      } catch (ParseException pe) {
        pe.printStackTrace();
      }
    } else {
      aRes.setRenderParameter("queryMode", "begin");
    }
```

Our `processAction` behavior depends on the portlet's mode. If we have a `searchString` object, then we use it to query Lucene. If not, we set a render parameter.

```
  } else if (mode.equals(PortletMode.EDIT)) {
    /**
     * If the Submit button is passed as a parameter, then we
```

```
         * know it came from a form.
         */
        if (aReq.getParameter("Submit") != null) {
          PortletPreferences prefs = aReq.getPreferences();
          String indexPath = aReq.getParameter("indexPath");
          prefs.setValue("indexPath", indexPath);

          try {
            prefs.store();
            aRes.setRenderParameter(
              "success",
              "The update was successful.");
          } catch (ValidatorException ve) {
            System.out.println("Preferences did not validate.");
            aRes.setRenderParameter(
              "success",
              "The preferences did not validate.");
          }

          aRes.setRenderParameter(
            "index",
            prefs.getValue("indexPath", ""));
        /**
         * Otherwise, we want to pull the preference, and pass it
         * as a render parameter to our portlet.
         */
        } else {
          PortletPreferences prefs = aReq.getPreferences();
          aRes.setRenderParameter(
            "index",
            prefs.getValue("indexPath", ""));
        }
      }
    }
  }
```

In `EDIT` mode, we set the `"index"` portlet preference based on the parameters submitted from the `searchEdit.jsp`. If there is no `"Submit"` parameter, we know that it is the initial view of the `EDIT` page, and therefore, we want to retrieve the `"index"` preference.

The `LucenePortlet` uses a common method of building portlets in that it delegates its user interface rendering to JSPs through a `PortletRequestDispatcher`. Two major actions are processed by the portlet. One, in `VIEW` mode, handles searching the Lucene index; and the other, in `EDIT` mode, allows modification of the Lucene index location, through a portlet preference called `indexPath`.

We want to determine whether the `indexPath` preference is actually pointed at a Lucene index, so we will implement our own `PreferencesValidator`, called `LuceneValidator`, which is shown in the following code:

```
/*
 * This class demonstrates a basic validator.
 *
 */
```

```
package org.opensourceportals.validator;
import java.io.File;
import java.util.ArrayList;
import java.util.List;
import javax.portlet.PortletPreferences;
import javax.portlet.PreferencesValidator;
import javax.portlet.ValidatorException;
/**
 * @author Clay Richardson
 */
public class LuceneValidator implements PreferencesValidator {
  /*
   * In order to create a validator, we implement the
   * javax.portlet.PreferencesValidator interface.
   *
   * If it fails validation, we throw a ValidatorException.
   *
   */
```

We declare our `LuceneValidator` by extending `PreferencesValidator`.

```
  public void validate(PortletPreferences preferences)
    throws ValidatorException {
    List problems = new ArrayList();
    String indexPath = preferences.getValue("indexPath", null);

    //Does the preference even exist?
    if (indexPath == null) {
      problems.add("indexPath");
      throw new ValidatorException(
        "indexPath preference doesn't exist",
        problems);
    } else {

      //Let's check to see if the index is actually there
      //The segments file is a good file to key off of...
      File index = new File(indexPath, "segments");
      if (!index.exists()) {
        problems.add("indexPath");
        throw new ValidatorException(
          "The index doesn't exist",
          problems);
      }
    }
  }
}
```

The `LuceneValidator` checks first to see if the `indexPath` preference is actually set to some value. If it is set, then it determines whether the specified directory contains a “segments” file, which would indicate the presence of a Lucene search engine index. This is a perfect example of how you would

want to use a validator as a mechanism to check a condition that is more complicated than simply "present" or "not present." If either of these conditions is false, then the `LuceneValidator` will throw a `ValidatorException`.

Now that we have managed the major logical parts of the `LucenePortlet`, we will review the presentation aspects of the portlet. The following code shows `search_view.jsp`, which presents a user interface for searching:

```
<%@ page import="javax.portlet.PortletURL" %>
<%@ taglib uri="http://java.sun.com/portlet" prefix="portlet" %>
<portlet:defineObjects/>
<table width="100%" border="0" cellspacing="0" cellpadding="0">
  <tr>
    <th>SEARCH</th>
  </tr>
  <tr>
    <td>
      <FORM action="<portlet:actionURL />" method="POST">
        <input type="text" name="searchString"/>
        <INPUT type="submit" name="Submit" value="Go"/>
      </FORM>
    </td>
  </tr>
</table>
```

In this JSP, we see the first use of the Portlet Tag Library. These tags provide access to the portlet from within a JSP. The most interesting use here is the `actionURL` tag, which is the proper way of creating links within a portlet application. Creating them as was traditionally done in Web applications, by appending parameters to an HTTP query string, is not correct, as it may omit portal-specific references that should be included in the URL.

Logically, we now need a page to view the results of our search. The following code provides a search results page:

```
<%@ page import="java.util.*,org.apache.lucene.search.*,java.text.*" %>
<%@ page import="org.apache.lucene.document.*" %>
<jsp:useBean id="hits" scope="request" type="org.apache.lucene.search.Hits"/>
<%
String shaded = "silver";
DecimalFormat form = new DecimalFormat("#");
%>
<p>
<i><b><%=hits.length()%></b></i> Documents match your request.
<p>
<center>
<table width="100%">
<%
   for (int i = 0; i < 25; i++) {
             Document doc = hits.doc(i);
             if (i % 2 == 0) {
                 shaded = "silver";
```

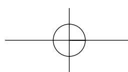
```
                } else {
                    shaded = "white";
                }
%>
     <tr bgcolor="<%=shaded%>">
         <td>
         <%=(i+1+startRow)%>.  (<font color="red">
         <%=form.format((hits.score(i)*100))%>%</font>)
                 <a href="<%=doc.get("url")%>"><%=doc.get("title")%></a><br>
         <%=doc.get("summary")%>
         </td>
  </tr>
<%        }  %>
</table>
</center>
```

As you can see, this is a simplistic search result page, providing only twenty-five search results, and no paging capability.

Finally, the following code provides the portlet application deployment descriptor, also known as the `portlet.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app>
  <portlet id="LucenePortlet">
    <portlet-name>LucenePortlet</portlet-name>
    <display-name>Example Search Portlet with Lucene</display-name>
    <portlet-class>org.opensourceportals.samples.LucenePortlet</portlet-class>
    <expiration-cache>-1</expiration-cache>
    <supports>
      <mime-type>text/html</mime-type>
      <portlet-mode>EDIT</portlet-mode>
      <portlet-mode>VIEW</portlet-mode>
      <portlet-mode>HELP</portlet-mode>
    </supports>
    <portlet-info>
      <title>Example Search Portlet with Lucene</title>
      <short-title>Lucene Search</short-title>
      <keywords>Search, Lucene</keywords>
    </portlet-info>
    <portlet-preferences>
      <preference>
        <name>indexPath</name>
        <value>C:\lucene\index</value>
      </preference>
      <preferences-
validator>org.opensourceportals.validator.LuceneValidator</preferences-validator>
    </portlet-preferences>
  </portlet>
</portlet-app>
```

The `portlet.xml` file is very basic, just like the portlet we just wrote. It provides three modes, and specifies a validator for its preferences.



Summary

This chapter focused on explaining the Java Portlet API. It provided a basic end-to-end example of a portlet to demonstrate the API in a practical use. However, for a more comprehensive chapter on building portlet applications, see Chapter 9.

04 469513 Ch01.qxd 1/16/04 11:04 AM Page 40