Building Portals with the Java Portlet API

JEFF LINWOOD, DAVE MINTER

Apress®

Building Portals with the Java Portlet API

ISBN (pbk): 1-59059-284-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin

Technical Reviewer: Carsten Ziegeler

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, John Franklin, Jason Gilmore, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Project Manager: Kylie Johnston

Copy Edit Manager: Nicole LeClerc

Copy Editor: Liz Welch

Production Manager: Kari Brooks

Production Editor: Ellie Fountain

Compositor: Kinetic Publishing Services, LLC

Proofreader: Nancy Sixsmith

Indexer: James Minkin

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, e-mail orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 3

The Portlet Life Cycle

As we've seen in earlier chapters, portlets are conceptually very similar to servlets. Like servlets, they can only operate within a container. Both have obligations that their design must satisfy to allow them to interact with their container, and both demand clearly specified behavior from their containers.

The portlet's obligations are, broadly speaking, to provide implementations of specific methods, to respond appropriately when these are invoked by the container, and to handle error conditions gracefully.

This chapter describes the container's interactions with a portlet, starting with its creation and concluding with its destruction. It also considers the constraints that are incumbent upon both the portlet and the container at each step in this life cycle.

The Portlet Interface

To demonstrate the basic steps in the life cycle, let's first look at a simple portlet that implements the `Portlet` interface directly. Portlets need to implement this interface, either directly, or indirectly by extending a class that has already implemented the interface.

Here is a devastatingly simple portlet example:

```
package com.portalbook.crawler;

import java.io.*;
import javax.portlet.*;

public class SimplePortlet
   implements Portlet {

   public SimplePortlet() {
   }

   public void destroy() {
      portletCounter--;
   }
```

```
public void init(PortletConfig config)
   throws PortletException
{
   portletCounter++;
}

public void processAction(
       ActionRequest request,
       ActionResponse response)
   throws PortletException, IOException
{
   actionCounter++;
}

public void render(
       RenderRequest request,
       RenderResponse response)
   throws PortletException, IOException
{
   renderCounter++;

   response.setTitle("Simple Portlet");
   response.setContentType("text/html");
   PrintWriter out = response.getWriter();
   out.write("The server has instantiated " +
          portletCounter +
          " copies of the portlet<br>");

   out.write("This portlet has been rendered " +
          renderCounter +
          " times (including this one)<br>");

   out.write("This portlet has received " +
          actionCounter +
          " action requests<br>");

   PortletURL action = response.createActionURL();
   out.write("Click <a href=\"");
   out.write(action.toString());
   out.write("\">here</a> to trigger an action.<br>");
}
```

```
    private static int portletCounter = 0;
    private int renderCounter = 0;
    private int actionCounter = 0;
}
```

The portlet.xml file for the simple portlet follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app
    xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
    version="1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd
    http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd">
    <portlet>
        <description>PortletBook Simple Portlet</description>
        <portlet-name>simple</portlet-name>
        <display-name>Simple Portlet</display-name>
        <portlet-class>com.portalbook.crawler.SimplePortlet</portlet-class>
        <expiration-cache>-1</expiration-cache>

        <supports>
            <mime-type>text/html</mime-type>
            <portlet-mode>VIEW</portlet-mode>
        </supports>

        <supported-locale>en</supported-locale>
        <portlet-info>
            <title>Simple Portlet</title>
            <short-title>Simple</short-title>
            <keywords>Simple, Example</keywords>
        </portlet-info>
    </portlet>
</portlet-app>
```

This portlet tracks the number of instances of the class that are being maintained by the portlet container at any given time, it counts the number of times that the specific instance has been rendered, and it counts the number of action requests that have been handled by the specific instance.

In the last section of this chapter, we will build a more realistic portlet application that demonstrates some of the issues involved in a threading application, and that builds upon the `GenericPortlet` class to make more complex portlet reactions possible.

Overview

It shouldn't come as much of a surprise to find that the life cycle of a portlet is broadly the same as that of a servlet. Servlets are generally responsible for rendering complete pages, and portlets are generally responsible for rendering fragments of pages, so there's an obvious correlation.

The life cycle of a portlet therefore breaks down into the following stages:

1. Creation of the portlet
2. Processing of a number of user requests (or possibly none)
3. Removal and garbage collection of the portlet

Creation of the Portlet

The creation of the portlet is probably the most complex "phase" in the life cycle since it involves three quite distinct steps. However, two of them—loading and instantiation—are very familiar Java concepts.

Loading the Classes

The container is able to load the classes required by the portlet at any point prior to invocation of the constructor.

A portlet application often consists of many classes and libraries, for which the actual portlet class is a relatively minor part of the whole. However, the portlet represents the user's way of interacting with the application. As such, it must have access to the rest of the application. The specification therefore demands that the portlet be loaded by the same classloader as the rest of the portlet application.

This guarantees that the servlets and other resources of the application may be accessed by the portlet that integrates it into the portal.

As with any class, at load time the class attributes will be initialized to their default values, so our variable `portletCounter` will be set to zero.

Invoking the Constructor

A portlet is required to provide a public default constructor—that is to say, a constructor taking no parameters.

Loading and instantiation (invoking the constructor) can take place either when the container starts the portlet application or when the container determines that the portlet is needed to service a request.

The option for delayed loading presents a benefit when a portlet will be used infrequently and consumes substantial resources, since they will not be acquired until they are actually needed. The trade-off is against performance, since the time taken by the portlet to service a request will be increased by the time taken to initialize its resources—but this will affect only the first user of the portlet. Where a portlet is initialized with the portlet application, the hit is taken "up front" when the application starts.

Our (minimal) constructor looks like this:

```
public SimplePortlet() {
}
```

It's hard to say anything interesting about our sample constructor. The normal instance initialization will take place, so that before the invocation of the constructor the attributes `renderCounter` and `actionCounter` will be set to zero, but then it does nothing, and, in fact, this is completely normal for a portlet implementation.

Initializing the Portlet

The container is required to initialize the portlet once it has been loaded and instantiated.

Although there's nothing to prevent you from doing useful initialization in the constructor, the configuration information isn't available to you until the `init()` method is called. As well as simplifying the implementation of the container, this allows the complete API of the portlet to be defined by the `Portlet` interface (interfaces don't allow the signature of the constructor to be specified):

```
public void init(PortletConfig config) throws PortletException
```

The `init()` method is passed an object implementing the `PortletConfig` interface. This object will be unique to the portlet definition and provides access to the initialization parameters and the `ResourceBundle` configured for the portlet in the portlet definition.

The `init()` method on a portlet instance is called only once by a portlet container.

Until the `init()` method has been invoked successfully, the portlet will not be considered active, so static initialization of the class should not trigger any methods that make this assumption. For example, the static (class rather than object scope) initializers of your class should not invoke connections to a database.

In our example, the `init()` method increases the number of portlet instances noted in the `portletCounter` attribute:

```
public void init(PortletConfig config)
              throws PortletException
{
              portletCounter++;
}
```

In a larger application, this method would be populated with code to extract configuration information in order to establish resources such as database connections and background threads. Our crawler example in the final section will demonstrate the initialization of a background thread in its `init()` method.

Exceptions During Initialization

The initialization process is error-prone. You are likely to be making connections to resources that may be unavailable and over which you have no control. For example, your database server might be unavailable. Without a mechanism for handling such errors, your portlet could end up in an invalid state. As you would expect, the usual exception-handling mechanism comes into play here.

The `init()` method is permitted to throw a `PortletException`. If it does so, the container is allowed to reattempt to load the portlet at any later time.

When constructing an `UnavailableException`, the portlet can provide a message describing the problem. In this case, the portlet must not be restarted. Use this exception if a configuration setting of the portlet will have to be changed to get the portlet work to properly. For instance, the portlet may require version 9 of a database to connect to, but the database it tried connecting to was version 7.

```
public UnavailableException(String text)
```

For example:

```
throw new UnavailableException("The database has been decommissioned");
```

Alternatively, you can specify a minimum period of time (in seconds) during which no attempt must be made to restart the portlet:

```
public UnavailableException(String text, int seconds)
```

For example:

```
throw new UnavailableException("The database is not currently available",5);
```

If the duration of the resource unavailability cannot be determined but is still considered to be a temporary problem, the portlet should return a zero or negative time.

For example:

```
throw new UnavailableException("A website resource could not be reached",0);
```

If any other `PortletException` is thrown, the container is allowed to attempt to restart the portlet at any time after the error. The container may either reuse the original instance, or discard the original and re-create it.

If a portlet needs to throw an exception from its initialization method, it must free any resources that it successfully acquired up to that point before doing so—this is because the `destroy()` method will not subsequently be called, as the portlet is considered to be uninitialized.

Request Handling

Once the portlet has been initialized, it is waiting for interactions with the users of the portal.

The container translates requests from the users of the portal into invocations of the `render()` and `processAction()` methods, thus elegantly breaking down the user requests into actions that command the portlet to change the state of its underlying application and render requests that display the application in its current state at any given point.

Users trigger actions by clicking on action URLs or submitting HTML forms that post data to an action URL. Upon receiving the action request from the user, the portal must invoke (via the container) the appropriate portlet's `processAction()` method. Once this method has completed, it must call the `render()` method for all of the portlets on the page. It is not required to invoke the `render()` methods in any particular order.

Users trigger render requests by either triggering action URLs as described previously, or by triggering a render URL. Again, upon receiving a request for a render URL from a user, the portal must invoke the `render()` method on all of the portlets in the page but is not obliged to follow any particular order.

The only exception to the invocations of the `render()` method as described is (optionally) for portlets that are cached by the portal and for which the state has not changed.

Since a single portlet is generally handling requests from multiple users, it must be able to handle simultaneous requests on different threads in each of these methods. In addition, it must not rely on any particular ordering of the calls to these methods.

Because the portlet cannot maintain all of the state information for a session, it is the container's responsibility to manage this and provide it when these methods are called.

Both action requests and render requests are similar, but each takes different classes for arguments. The action request cannot write any content to the portlet's response, because its `ActionResponse` does not have access to the output. The

signatures of the two methods are very similar. First, here is the `processAction()` method signature:

```
public void processAction(
   ActionRequest request,
   ActionResponse response)
throws PortletException, IOException
```

And here is the `render()` method signature:

```
public void render(
   RenderRequest request,
   RenderResponse response)
throws PortletException, IOException
```

Each method receives a request object and a response object tailored to its function. In each case, the request represents the state of the session for the user, and the response object allows the method to interact with the portlet's response.

The `RenderRequest` object will not generally need to change the state of the underlying portlet application, so it provides the portlet with the information necessary to produce a view of it in its current state.

Specifically, these include

- The state of the portlet window (minimized, maximized, etc.)
- The mode of the portlet (e.g., VIEW mode)
- The context of the portlet
- The session associated with the portlet (including authorization information)
- The preferences information associated with the portlet
- Any render parameters that have been set on a render URL from a posted Form, or that have been set during the `processAction()` method

The `ActionRequest` object represents an opportunity to change the state of the portlet based on its current state, so this provides everything offered by the `PortletRequest` along with direct access to the content of the HTTP request made by the user of the portal.

Note that `ActionRequest` and `RenderRequest` are both interfaces, so it is the responsibility of the container to provide concrete implementation classes giving access to the appropriate information

To respond to `processAction()` the portlet should update its `ActionResponse` object. This provides methods to

- Redirect the client to a new page
- Change the mode of the portlet
- Add or modify rendering parameters for the user's session

You change the state of the portlet's container window in the portal. To respond to `render()`, the portlet should update its `RenderResponse` object. This provides methods to

- Render content into the container window displayed in the user's view of the portal
- Render URLs into that content, which will invoke actions on the portlet

Again, `ActionResponse` and `RenderResponse` are interfaces, and the container must provide a suitable implementation to be used by the portlet.

Here is our sample `processAction()` method:

```
public void processAction(
          ActionRequest request,
          ActionResponse response)
       throws PortletException, IOException
   {
       actionCounter++;
   }
```

The example `processAction()` makes only a trivial change to the state of the portlet; it increments the counter of actions handled, so it does not have to inform the container of any changes via the response.

Our sample `render()` method looks like this:

```
public void render(
          RenderRequest request,
          RenderResponse response)
       throws PortletException, IOException
   {
       renderCounter++;

       response.setTitle("Simple Portlet");
       response.setContentType("text/html");
       PrintWriter out = response.getWriter();
       out.write("The server has instantiated " +
              portletCounter +
              " copies of the portlet<br>");
```

```
        out.write("This portlet has been rendered " +
               renderCounter +
               " times (including this one)<br>");

        out.write("This portlet has received " +
               actionCounter +
               " action requests<br>");

        PortletURL action = response.createActionURL ();
        out.write("Click <a href=\"");
        out.write(action.toString());
        out.write("\">here</a> to trigger an action.<br>");
    }
```

Because our sample portlet does not need to tailor its view to the different users of the system, it is able to ignore the request parameter, which contains the user-specific (session) state information. It does, however, need to render its current state to the browser, and must specify the type of content that it will produce.

Our sample portlet demonstrates an important relationship between the portlet and the portal: one portlet can be rendered multiple times on a single page. If an instance of the portlet is placed in a portal page in two distinct places, the portlet will be loaded once, rendered multiple times (twice each time the portal page as a whole is rendered), and destroyed once.

The last few lines of this example method are of particular interest, since they demonstrate how to provide a mechanism by which actions (and thus calls by the container to `processAction()`) can be rendered:

```
PortletURL action = response.createActionURL ();
```

This retrieves an object from the response object provided in the parameters of the `render()` method, which can render a URL representing a specific action. Our example has only one type of action, but methods such as `addParameter()` can be called on the `PortletURL` object to differentiate between calls to the various actions that you want to implement.

Again, `PortletURL` is an interface, and it is the responsibility of the container to provide a suitable implementation.

It is not appropriate to hardcode a URL into your portlet since the precise details of the mappings between URLs and portlets are configurable by the administrator of the portal. In addition, portlet URLs distinguish between different instances of a portlet running inside of a portlet. The portlet URLs are usually prefixed with a namespace or another unique ID. These details will be specific to the portal in which your portlet is running, so if you want to make your portlet compatible between portals (and even between versions of the same portal) you should always rely on the `createActionURL()` method.

Destroying the Portlet

The `destroy()` method will not be invoked until all other initialization or processing threads on the instance have completed. It will be invoked when the container determines that the portlet is no longer required—the container is not required to keep the portlet in service for any specific period of time.

The container invokes the `destroy()` method to release any resources that have been retained by the portlet. The portlet will then be de-referenced by the container and the garbage collector will be free to remove the portlet object from memory.

The `destroy()` method is guaranteed to be called (unless initialization failed with an exception), so this is also an appropriate place in which to notify other parts of the application that the portlet is becoming unavailable.

Finalizers should not be used since their invocation is not guaranteed.

Destroying our example portlet looks like this:

```
public void destroy() {
   portletCounter--;
}
```

Our example portlet uses the `destroy()` method to reduce the count of its running instances in the class attribute `portletCounter`.

Threading Issues

In this final section, we demonstrate the life cycle of a portlet that uses background threads of execution in a web crawler application.

Handling Concurrent Requests

Because the portlet container will handle concurrent requests from clients by invoking the methods on the portlet on separate threads of execution, your portlet must be able to handle any combination and number of simultaneous calls to `render()` and/or `processAction()`.

You must therefore implement your portlet to handle these concurrent requests safely. In practice this is not usually too tricky—all the information you need to process a request is provided in a thread-safe manner in the parameter list, so if your portlets don't use instance variables and they don't access other resources external to the portlet, your application will automatically be thread-safe.

It is guaranteed that your `init()` method will be called only once at the beginning of the life cycle and that no other methods will be invoked by the container until `init()` completes successfully, so your `init()` method does *not* have to be thread safe.

Your `render()` and `processAction()` methods will be invoked with request and response objects. These are guaranteed to be unique to that invocation of the method *during the lifetime of the method*. Containers are likely to recycle these objects once the method in question has completed, so retaining a reference to them outside the scope of the method to which they were passed may result in unexpected behavior.

Our Thread-Safe Crawler

Our crawler class is implemented to be thread safe. It implements `Runnable` so it can be created and started within a background thread.

Once the crawler is running, it can be queried at any time. The `get` methods return unmodifiable sets so it is not possible for the client to externally alter their state.

The crawler can be stopped by an external thread by calling the `stopCrawler()` method. This is essential so that our portlet can be unloaded safely.

Our crawler implementation follows, with a running commentary. Although this illustrates the functionality that's needed in a web crawler, you should note that it is a demonstration application only. We make a lot of assumptions and take shortcuts that would not be acceptable in a commercial product.

```
package com.portalbook.crawler;

import java.io.*;
import java.net.*;
import java.util.*;

public class Crawler
   implements Runnable
{
```

Our simplest constructor creates an instance to crawl a given path. This will search only within the host of the path specified:

```
   public Crawler(String path)
      throws MalformedURLException
   {
      this(path,DEFAULT_LINK_DEPTH);
   }
```

This more complex constructor creates an instance to crawl a given path. It will search within the host of the path specified, and up to the specified number of sites (`depth`) away. If `depth` is 2, the crawler will look within the host of the path specified and within the hosts of sites referenced directly from this site, but no further:

```
public Crawler(String path, int depth)
   throws MalformedURLException
{
   this(new HashSet(
          Arrays.asList(
              new Object[]
                 { new URL(path) })),
         new HashSet(),
         new HashSet(),
         new HashSet(),
         new HashSet(),
         depth);
}
```

This internal constructor is used directly or indirectly by the public ones to create the instance to crawl a given set of paths. It will search within the hosts specified, and up to the specified number of sites away from those sites. It will not search forbidden hosts, failed hosts, or already visited hosts:

```
protected Crawler( Set links,
                   Set visited,
                   Set visitedHosts,
                   Set forbidden,
                   Set failed,
                   int depth )
{
   this.links        = links;
   this.visited      = visited;
   this.visitedHosts = visitedHosts;
   this.forbidden    = forbidden;
   this.failed       = failed;
   this.currentHost  =
      (URL)links.iterator().next();
   this.depth        = depth;
}
```

Our crawler is designed to run as a thread, so it implements the `Runnable` interface. It can therefore be passed in as a parameter to a new `Thread` object.

When the `start()` method of the containing `Thread` object is called, the following `run()` method will be started on the background thread of execution:

```
public void run() {
```

This code flags that work is in progress:

```
   setStopped(false);
   try {
```

until we instruct the thread to stop, or it runs out of links to search, or it reaches the edge of the network of links that we're allowing it to search:

```
      while( !isStopped() &&
             (links.size() > 0) &&
             (depth > 0) ) {
```

This code gets the first link from the queue of links to search:

```
         URL link =
            (URL)links.iterator().next();
```

If the link is on a different host

```
         if( !isCurrentHost(link) ) {
```

this code goes to that host and searches the link (if we're allowed):

```
            crawlNewHost(link);
         } else {
```

This code crawls the link:

```
            crawl(link);
         }
      }
```

Normal or abnormal termination should flag that work has completed regardless:

```
   } finally {
      setStopped(true);
   }
}
```

TIP *Most of the work of the crawler is carried out from within private methods. As with all good object-oriented designs, these hide the implementation details from the user of the class—this is particularly important with a multithreaded design, since you will need to carefully isolate any code that might cause problems if two different threads were to execute it simultaneously.*

This private method processes a buffer containing the contents of a robots.txt file and adds forbidden URLs to the appropriate list. The link for the robots.txt file is required to convert the relative disallowed paths into absolute URLs:

```
private void processRobotBuffer(
      StringBuffer buffer,
      URL link)
{
```

Now we prepare to gather up potential URL strings:

```
   List disallows = new ArrayList();
```

and look for `DISALLOW` tokens. Any token immediately following a `DISALLOW` is presumed to be (potentially) a path:

```
   StringTokenizer tokenizer =
      new StringTokenizer(buffer.toString());
   while(tokenizer.hasMoreElements()) {
      String element = tokenizer.nextToken();

      if( element.equalsIgnoreCase(DISALLOW) &&
          tokenizer.hasMoreElements() ) {

         String path = tokenizer.nextToken();
         disallows.add(path);
      }
   }
```

The following code iterates over the disallow tokens gathered and converts them into absolute paths and then URLs to forbid access:

```
   Iterator i = disallows.iterator();
   while(i.hasNext()) {
      String path = (String)i.next();
      try {
         URL disallowedURL = new URL(link,path);
         forbidden.add(disallowedURL);
```

```
            } catch( MalformedURLException e ) {
               // Couldn't form a URL from this.
               // No point disallowing access to a
               // link we can't access anyway.
            }
         }
      }
```

This private method determines the robots.txt file that governs access to the host on which the link resides, parses, and then disallows access to links as required by the robots specification; however, we're polite and assume that *any* link that's disallowed to *anybody* must be forbidden to us. This makes the logic slightly simpler but would not normally be done in a production system. The robots.txt information is cached with the host as the key, so we look it up only when we encounter the first link from a site:

```
   private void processRobots(URL link) {
```

If the host has been visited before, we're not obliged to reread the robots.txt file. If this is the first visit, however, we need to determine which paths to discard:

```
      HttpURLConnection connection = null;
      try {
         if( !visitedHosts.contains(link.getHost()) ) {
```

This code creates an HTTP connection to the link:

```
            URL robotLink = new URL(link,"/robots.txt");

            connection =
               (HttpURLConnection)robotLink.openConnection();
```

This code gets the page:

```
            connection.setRequestMethod(GET);
            connection.setRequestProperty(
                  USER_AGENT,AGENT_IDENTIFIER);
```

And this reads the page into a buffer:

```
            InputStream input =
               (InputStream)connection.getContent();
```

```
            BufferedReader reader =
               new BufferedReader(
                  new InputStreamReader(input));

            int i = 0;
            StringBuffer buffer = new StringBuffer();
            String line = null;
            while((line = reader.readLine()) != null) {
               i++;
               buffer.append(line);
               buffer.append('\r');
            }
```

We now close the connection to the page and discard held resources:

```
            reader.close();
            connection.disconnect();
            connection = null;
```

Next we process the buffer to determine what links are permitted:

```
            processRobotBuffer(buffer,link);
         }
      } catch( IOException e ) {
```

Good. There is no robots.txt file, and we're allowed to view the site.
To ensure that the connection is always closed correctly, we use the following:

```
      } finally {
         if( connection != null ) {
            connection.disconnect();
         }
      }
   }
```

This private method crawls a specified URL and adds all valid HREF entries to the queue of links to be crawled (unless they are denied by the appropriate robots.txt file or have already been crawled):

```
   private void crawl(URL link) {
      HttpURLConnection connection = null;
      try {
```

Now let's check to see what is or isn't permitted on this host:

```
         processRobots(link);
```

This code creates an HTTP connection to the link:

```
connection =
   (HttpURLConnection)link.openConnection();
```

Now we get the page:

```
connection.setRequestMethod(GET);

String contentType = connection.getContentType();
if( (contentType != null) &&
    contentType.startsWith(TEXT_HTML) ) {

   InputStream input =
      (InputStream)connection.getContent();
```

And read the page into a buffer:

```
   BufferedReader reader =
   new BufferedReader(new InputStreamReader(input));

   int i = 0;
   StringBuffer buffer = new StringBuffer();
   String line = null;
   while((line = reader.readLine()) != null) {
      i++;
      buffer.append(line);
   }
```

To close the connection to the page and discard held resources, we use this code:

```
   reader.close();
   connection.disconnect();
   connection = null;
```

The next step is to process the page. This code assumes that the page is small enough to manage in memory:

```
   processBuffer(buffer,link);
} else {
   connection.disconnect();
   connection = null;
}
```

We've handled the item, so let's remove it from the queue of links:

```
            visited(link);
        } catch( IOException e ) {
```

The item caused a problem, so let's remove it from the queue:

```
            links.remove(link);
            failed.add(link);
        } finally {
```

To ensure that the connection is closed correctly, we use the following:

```
            if( connection != null ) {
                connection.disconnect();
            }
        }
    }
```

This private method carries out actions once a link has been crawled. It removes the link from the queue of links to visit, adds it to the set of visited links, and adds its host to the set of visited hosts:

```
    private void visited(URL link) {
        links.remove(link);
        visited.add(link);
        visitedHosts.add(link.getHost());
    }
```

This private method processes a buffer containing an HTML page. The context is provided so that relative URLs can be resolved to their absolute URLs. HREFs within the HTML are extracted, converted to absolute URLs, and added to the queue of links to be crawled (if they have not already been visited and have not been forbidden by a robots.txt file):

```
    private void processBuffer(StringBuffer buffer, URL url) {
```

Let's now prepare to gather up potential URL strings:

```
        List foundHREFs = new ArrayList();
```

And look for HREF tokens. Any token immediately following an HREF is presumed to be (potentially) a URL:

```
        StringTokenizer tokenizer =
            new StringTokenizer(buffer.toString(),DELIM);
```

```
while(tokenizer.hasMoreElements()) {
   String element = tokenizer.nextToken();

   if( element.equalsIgnoreCase(HREF) &&
       tokenizer.hasMoreElements() ) {

      String path = tokenizer.nextToken();
      foundHREFs.add(path);
   }
}
```

Now let's boil them down to absolute URLs:

```
Set absolute = new HashSet();
Iterator i = foundHREFs.iterator();
while(i.hasNext()) {
   String path = (String)i.next();

   try {
      URL toAdd = new URL(url,path);
      if( !toAdd.getProtocol().equalsIgnoreCase("http") ) {
```

If it's not a link beginning with "http" then it's not a protocol we're interested in:

```
      } else {
         absolute.add(toAdd);
      }
   } catch( MalformedURLException e ) {
```

If we encounter a `MalformedURLException`, then the crawler must have tried to load an invalid path. We should ignore the URL as it's probably a typo:

```
   }
}
```

Let's now remove all the URLs that we're not allowed to visit for one reason or another:

```
absolute.removeAll(forbidden);
```

And remove all the URLs that we've already visited anyway:

```
absolute.removeAll(visited);
```

We'll add the remainder to the visit queue:

```
      links.addAll(absolute);
   }
```

The following private method crawls a link that is on a host other than the one currently being processed. The simplest way to effect this is to instantiate a new crawler specifying the appropriate URL, but to pass it the information on visited and forbidden hosts so that already crawled links can be ignored. Note that we reduce the permitted depth by 1 for this next crawler so that it can't creep too far away from the original link. Eventually we'll reach 0, and there's no point in instantiating the crawler because it would simply complete immediately.

```
   private void crawlNewHost(URL url) {
      Set links = new HashSet();
      links.add(url);
      if( depth > 1 ) {
```

Now let's create a new crawler to crawl the external link:

```
         Crawler crawler =
            new Crawler(links,
                        visited,
                        visitedHosts,
                        forbidden,
                        failed,
                        (depth - 1));
```

The new crawler should be stopped whenever this crawler is stopped:

```
         addListener(crawler);
```

This code flags that the URL has been visited:

```
         crawler.run();
      }

      this.links.remove(url);
   }
```

This private method adds a crawler to a set of crawlers that are children of this crawler so that they can all by stopped when this one is stopped:

```
   private void addListener(Crawler crawler) {
      stopListeners.add(crawler);
   }
```

This private method stops all the crawlers that are children of this crawler:

```
private void notifyStopCrawlers() {
   Iterator i = stopListeners.iterator();
   while(i.hasNext() && depth > 0) {
      Crawler crawler = (Crawler)i.next();
      crawler.setStopped(true);
   }
}
```

This method allows the owner of the crawler to test to see if the crawler is currently running:

```
public boolean isStopped() {
   return stopped;
}
```

This method allows the owner to stop the crawler:

```
public void stopCrawler() {
   setStopped(true);
}
```

The following private method sets the flag that causes the crawler to stop, and if it's set to `true`, it instructs any child crawler threads to stop as well:

```
private void setStopped(boolean stopped) {
   if(stopped) notifyStopCrawlers();
   this.stopped = stopped;
}
```

The next method allows the owner of the crawler to retrieve the set of links (as URL objects) that the crawler has encountered so far and that were valid. Access to the set of visited URLs is synchronized, and a copy is returned, so that it is not possible for two threads to try to access the set simultaneously. By pursuing this policy for all of the publicly accessible stores of data provided by the crawler, we make it thread safe with very little use of synchronization. Judicious use of the `synchronized` keyword is important because it carries a significant performance penalty. While this isn't a big deal for our crawler (which is not limited by processing speed so much as by bandwidth), getting good responsiveness is often the primary goal when using threads:

```
public Set getVisitedURLs() {
   synchronized(this.visited) {
      return new HashSet(this.visited);
   }
}
```

Similarly, this code allows the owner to retrieves the set of hosts (as `Strings`) that the crawler has encountered so far:

```
public Set getVisitedHosts() {
   synchronized(this.visitedHosts) {
      return new HashSet(this.visitedHosts);
   }
}
```

The following code retrieves the set of links as URLs that the crawler was forbidden to search by their associated robots.txt files:

```
public Set getForbiddenURLs() {
   synchronized(this.forbidden) {
      return new HashSet(this.forbidden);
   }
}
```

Next we retrieve the set of invalid links as URLs that the crawler could not search (usually because of a network problem of some sort, such as "404 Host Not Found"):

```
public Set getFailedURLs() {
   synchronized(this.failed) {
      return Collections.unmodifiableSet(this.failed);
   }
}
```

This private method determines if a given link falls within the set of links being crawled by this crawler's instance:

```
private boolean isCurrentHost(URL url) {
   return currentHost.getHost().equalsIgnoreCase(url.getHost());
}
```

This field identifies the host that this crawler is searching:

```
private URL currentHost;
```

These fields are used to stop this thread and its child threads:

```
private Set stopListeners = new HashSet();
private boolean stopped = true;
```

These fields maintain information on where we've been so far:

```
   private Set links;
   private Set visited;
   private Set visitedHosts;
   private Set forbidden;
   private Set failed;
   private int depth;
```

The following are `String` constants required by the application mostly during tokenization or protocol negotiation:

```
   private static final String DELIM      =  "\t\r\n\" <>=";
   private static final String TEXT_HTML  =  "text/html";
   private static final String GET        =  "GET";
   private static final String HREF       =  "HREF";
   private static final String HTTP       =  "HTTP:";
   private static final String DISALLOW   =  "DISALLOW:";
   private static final String USER_AGENT =  "User-Agent";
```

This is the "official" name of this agent for identification in the logs of the servers we're searching:

```
   private static final String AGENT_IDENTIFIER="WoolGatherer";
```

Finally, we specify the default search depth to be used if the single-argument constructor is used:

```
   private static final int DEFAULT_LINK_DEPTH = 1;
}
```

Here is the portlet.xml file for the crawler portlet:

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

      <portlet-name>crawler</portlet-name>
      <display-name>Link Crawler</display-name>
```

```
        <portlet-class>
            com.portalbook.crawler.CrawlerPortlet
        </portlet-class>

        <init-param>
            <name>crawlPath</name>
            <value>http://localhost:8080/</value>
        </init-param>

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

        <supports>
            <mime-type>text/html</mime-type>
            <portlet-mode>VIEW</portlet-mode>
        </supports>

        <supported-locale>en</supported-locale>

        <portlet-info>
            <title>Link Crawler Portlet</title>
            <short-title>Crawler</short-title>
            <keywords>Crawler, Link, Checker</keywords>
        </portlet-info>

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

The GenericPortlet Abstract Class

Portlets are allowed to be displayed in the three standard modes—VIEW, EDIT, and HELP. They do not have to support any of them except VIEW. Many do not require any action handling. This results in a lot of boilerplate code to handle the requests for each of these modes and render the response.

The `GenericPortlet` abstract class therefore provides an initial framework around which you can build your custom portlet. At its simplest, a portlet derived from `GenericPortlet` must implement a constructor, a `getTitle()` method, and a `doView()` method. `GenericPortlet` then provides your application with a variety of helper methods that you will make frequent use of. It is generally recommended that you override `GenericPortlet` rather than directly creating your own implementation of the `Portlet` interface.

`GenericPortlet`'s `render()` method calls the `doDispatch()` method. This method determines the current mode of the portlet render request, and calls `doView()`, `doEdit()`, and `doHelp()` as appropriate (the modes that are permissible for a portlet are configured in its deployment descriptor). The `processAction()` method throws

an exception if invoked, so your class must override this if you wish to handle any actions.

Since our crawler portlet provides only a view method and does not accept any actions, `GenericPortlet` provides the perfect starting point.

Our Threaded Crawler Portlet

Our crawler overrides `GenericPortlet`; this will simplify the implementation of our very conventional design.

The constructor is empty:

```
public CrawlerPortlet() {
   super();
}
```

The `init()` method retrieves the path from which the crawler will start from the portlet configuration:

```
String path = (String)config.getInitParameter("crawlPath");
```

It then creates a new crawler object, and invokes it on a background thread:

```
crawler = new Crawler(path);
Thread background = new Thread(crawler);
background.start();
```

Our code does not need to retain a reference to the thread object, since this will not be manipulated directly, but it does retain a reference to the crawler object, since it will need to access its methods in order to retrieve information and ultimately stop the crawler.

When the portlet is eventually unloaded by the container, the `destroy()` method will be invoked:

```
public void destroy() {
   crawler.stopCrawler();
   crawler = null;
}
```

This method cleans up the portlet's resources by stopping the crawler. The crawler's thread will exit, and no further action will be necessary (the container will manage the removal of the portlet from memory).

When running, the portlet may be required to render itself into its container and then onto the portal itself. This process would typically be triggered by the user browsing to a page on which the portlet is configured to be displayed.

The `render()` method of the `Portlet` interface will be invoked. This is caught by the `GenericPortlet` implementation, and it instead sets the title (which it determines by invoking `getTitle()`) and then invokes `doDispatch()`. The default implementation gets the title from the `ResourceBundle` of the `PortletConfig` of the portlet, but we will override this to return a specific string without requiring configuration:

```
protected String getTitle(RenderRequest request) {
   return "Link Crawler";
}
```

`doDispatch()` is provided as a default implementation by `GenericPortlet`. It determines the mode of the portlet (VIEW, EDIT, or HELP) and then calls the `doView()`, `doEdit()`, or `doHelp()` method as appropriate in order to render the portlet in its appropriate mode.

Our sample provides only a VIEW mode, so all render requests must result in a call to `doView()`:

```
protected void doView( RenderRequest request, RenderResponse response)
       throws   PortletException,
                 IOException
{
       response.setContentType("text/html");
       PrintWriter out = response.getWriter();

       out.write("<table><tr><td>");
       out.write("<h2>Crawler</h2>");

       out.write("<table cellspacing=\"0\"")
       out.write("border=\"1\">");
       out.write("<tr><td align=\"right\">")
       out.write("<i>Status</i></td><td><b>");
       out.write(
          crawler.isStopped() ? "stopped" : "running");
       out.write("</b></td></tr>");
       out.write("</table>");

       renderCollection(out, "Hosts Crawled",
          crawler.getVisitedHosts());

       renderCollection(out, "Links Visited",
          crawler.getVisitedURLs());

       renderCollection(out, "Failed Links",
          crawler.getFailedURLs());
```

```
        renderCollection(out, "Forbidden Links",
            crawler.getForbiddenURLs());

        out.write("</td></tr></table>");
}
```

Whenever `doView()` is invoked, we render a simple set of tables demonstrating the current state of the crawler running in the background. Appropriate headers for the response are rendered, and then the output print writer is retrieved from the response object and all output is generated using this.

Both of our examples have used inline HTML to render output to the portlet window. Although this is straightforward, you can see with this example that it rapidly makes the structure of the page difficult to follow. In Chapter 5, we discuss the option of delegating the rendering of the page to a servlet or JavaServer Pages (JSP) page instead.

The full implementation of our threaded portlet follows. Because our crawler implementation is thread safe, we have only to ensure that we start the crawler thread in the `init()` method, and stop it in the `destroy()` method, and that we don't start any additional threads during the lifetime of our portlet.

Here is the threaded crawler portlet in full:

```
package com.portalbook.crawler;

import java.io.*;
import java.net.*;
import java.util.*;
import javax.portlet.*;

public class CrawlerPortlet
    extends GenericPortlet
{
    public CrawlerPortlet() {
        super();
    }

    public void init(PortletConfig config)
    throws PortletException
    {
        String path =
            (String)config.getInitParameter("crawlPath");

        try {
            crawler = new Crawler(path);

            // Here we create and kick off the background
            // thread of execution.
```

```
        Thread background = new Thread(crawler);
        background.start();

    } catch( MalformedURLException e ) {
        throw new PortletException(
            "Portlet could not be initialised",e);
    }
}

public void destroy() {
    // Here we ensure that the background
    // thread is terminated safely.
    crawler.stopCrawler();
    crawler = null;
}

private void renderCollection(
        PrintWriter out,
        String title,
        Collection collection )
    throws IOException
{
    out.write("<table cellspacing=\"0\"");
    out.write("border=\"0\">");
    out.write("<tr><td><b>");
    out.write(title);
    out.write("</b></td></tr>");

    Iterator i = collection.iterator();
    while(i.hasNext()) {
        out.write("<tr><td>");
        out.write(i.next().toString());
        out.write("</td></tr>");
    }

    out.write("</table><br>");
}

protected void doView(
        RenderRequest request,
        RenderResponse response)
    throws   PortletException,
             IOException
{
```

```
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.write("<table><tr><td>");
        out.write("<h2>Crawler</h2>");

        out.write("<table cellspacing=\"0\"");
        out.write("border=\"1\">");
        out.write("<tr><td align=\"right\">");
        out.write("<i>Status</i></td><td><b>");
        out.write(
           crawler.isStopped() ? "stopped" : "running");
        out.write("</b></td></tr>");
        out.write("</table>");

        // Note that while we can access an object running
        // in a background thread from the doView method,
        // we must not create a new thread of execution here.
        renderCollection(out, "Hosts Crawled",
           crawler.getVisitedHosts());

        renderCollection(out, "Links Visited",
           crawler.getVisitedURLs());

        renderCollection(out, "Failed Links",
           crawler.getFailedURLs());

        renderCollection(out, "Forbidden Links",
           crawler.getForbiddenURLs());

        out.write("</td></tr></table>");
     }

     protected String getTitle(RenderRequest request) {
        return "Link Crawler";
     }

     private Crawler crawler;
}
```

Summary

In this chapter, we have discussed the life cycle of a portlet. The portlet container calls the `init()` method on the portlet to initialize the portlet, the portlet handles any requests from the user, and the portlet is destroyed. The user can send either action or render requests, and we discussed the order in which these requests are handled. We also discussed error handling from the portlet's initialization phase.

Our web crawler example demonstrates how a portlet can create and manage a background thread. We discussed which methods on the portlet need to be made thread safe, and which objects are guaranteed to be unique during processing.

In the next chapter, we discuss the concepts that underlie the design of the portlet API and that account for the specifics of the portlet life cycle that we have just described.