CHAPTER

6

Crawling the Web with Java

By James Holmes

Have you ever wondered how Internet search engines like Google and Yahoo! can search the Internet on virtually any topic and return a list of results so quickly? Obviously it would be impossible to scour the Internet each time a search request was initiated. Instead search engines query highly optimized databases of Web pages that have been aggregated and indexed ahead of time. Compiling these databases ahead of time allows search engines to scan billions of Web pages for something as esoteric as "astrophysics" or as common as "weather" and return the results almost instantly.

The real mystery of search engines does not lie in their databases of Web pages, but rather in how the databases are created. Search engines use software known as *Web crawlers* to traverse the Internet and to save each of the individual pages passed by along the way. Search engines then use additional software to index each of the saved pages, creating a database containing all the words in the pages.

Web crawlers are an essential component to search engines; however, their use is not limited to just creating databases of Web pages. In fact, Web crawlers have many practical uses. For example, you might use a crawler to look for broken links in a commercial Web site. You might also use a crawler to find changes to a Web site. To do so, first, crawl the site, creating a record of the links contained in the site. At a later date, crawl the site again and then compare the two sets of links, looking for changes. A crawler could also be used to archive the contents of a site. Frankly, crawler technology is useful in many types of Web-related applications.

Although Web crawlers are conceptually easy in that you just follow the links from one site to another, they are a bit challenging to create. One complication is that a list of links to be crawled must be maintained, and this list grows and shrinks as sites are searched. Another complication is the complexity of handling absolute versus relative links. Fortunately, Java contains features that help make it easier to implement a Web crawler. First, Java's support for networking makes downloading Web pages simple. Second, Java's support for regular expression processing simplifies the finding of links. Third, Java's Collection Framework supplies the mechanisms needed to store a list of links.

The Web crawler developed in this chapter is called Search Crawler. It crawls the Web, looking for sites that contain strings matching those specified by the user. It displays the URLs of the sites in which matches are found. Although Search Crawler is a useful utility as is, its greatest benefit is found when it is used as a starting point for your own crawler-based applications.

Fundamentals of a Web Crawler

Despite the numerous applications for Web crawlers, at the core they are all fundamentally the same. Following is the process by which Web crawlers work:

1. Download the Web page.
2. Parse through the downloaded page and retrieve all the links.
3. For each link retrieved, repeat the process.

Now let's look at each step of the process in more detail.

In the first step, a Web crawler takes a URL and downloads the page from the Internet at the given URL. Oftentimes the downloaded page is saved to a file on disk or put in a database. Saving the page allows the crawler or other software to go back later and manipulate the page, be it for indexing words (as in the case with a search engine) or for archiving the page for use by an automated archiver.

In the second step, a Web crawler parses through the downloaded page and retrieves the links to other pages. Each link in the page is defined with an HTML anchor tag similar to the one shown here:

```
<A HREF="http://www.host.com/directory/file.html">Link</A>
```

After the crawler has retrieved the links from the page, each link is added to a list of links to be crawled.

The third step of Web crawling repeats the process. All crawlers work in a recursive or loop fashion, but there are two different ways to handle it. Links can be crawled in a depth-first or breadth-first manner. *Depth-first crawling* follows each possible path to its conclusion before another path is tried. It works by finding the first link on the first page. It then crawls the page associated with that link, finding the first link on the new page, and so on, until the end of the path has been reached. The process continues until all the branches of all the links have been exhausted.

Breadth-first crawling checks each link on a page before proceeding to the next page. Thus, it crawls each link on the first page and then crawls each link on the first page's first link, and so on, until each level of links has been exhausted. Choosing whether to use depth- or breadth-first crawling often depends on the crawling application and its needs. Search Crawler uses breadth-first crawling, but you can change this behavior if you like.

Although Web crawling seems quite simple at first glance, there's actually a lot that goes into creating a full-fledged Web crawling application. For example, Web crawlers need to adhere to the "Robot protocol," as explained in the following section. Web crawlers also have to handle many "exception" scenarios such as Web server errors, redirects, and so on.

Adhering to the Robot Protocol

As you can imagine, crawling a Web site can put an enormous strain on a Web server's resources as a myriad of requests are made back to back. Typically, a few pages are downloaded at a time from a Web site, not hundreds or thousands in succession. Web sites also often have restricted areas that crawlers should not crawl. To address these concerns, many Web sites adopted the *Robot protocol,* which establishes guidelines that crawlers should follow. Over time, the protocol has become the unwritten law of the Internet for Web crawlers.

The Robot protocol specifies that Web sites wishing to restrict certain areas or pages from crawling have a file called **robots.txt** placed at the root of the Web site. Ethical crawlers will

reference the robot file and determine which parts of the site are disallowed for crawling. The disallowed areas will then be skipped by the ethical crawlers. Following is an example **robots.txt** file and an explanation of its format:

```
# robots.txt for http://somehost.com/

User-agent: *
Disallow: /cgi-bin/
Disallow: /registration  # Disallow robots on registration page
Disallow: /login
```

The first line of the sample file has a comment on it, as denoted by the use of a hash (#) character. Comments can be on lines unto themselves or on statement lines, as shown on the fifth line of the preceding sample file. Crawlers reading **robots.txt** files should ignore any comments.

The third line of the sample file specifies the *User-agent* to which the *Disallow* rules following it apply. User-agent is a term used for the programs that access a Web site. For example, when accessing a Web site with Microsoft's Internet Explorer, the User-agent is "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)" or something similar to it. Each browser has a unique User-agent value that it sends along with each request to a Web server. Web crawlers also typically send a User-agent value along with each request to a Web server. The use of User-agents in the **robots.txt** file allows Web sites to set rules on a User-agent–by–User-agent basis. However, typically Web sites want to disallow all robots (or User-agents) access to certain areas, so they use a value of asterisk (*) for the User-agent. This specifies that all User-agents are disallowed for the rules that follow it. You might be thinking that the use of an asterisk to disallow all User-agents from accessing a site would prevent standard browser software from working with certain sections of Web sites. This is not a problem, though, because browsers do not observe the Robot protocol and are not expected to.

The lines following the User-agent line are called *disallow statements*. The disallow statements define the Web site paths that crawlers are not allowed to access. For example, the first disallow statement in the sample file tells crawlers not to crawl any links that begin with "/cgi-bin/". Thus, the URLs

http://somehost.com/cgi-bin/
http://somehost.com/cgi-bin/register

are both off limits to crawlers according to that line. Disallow statements are for paths and not specific files; thus any link being requested that contains a path on the disallow list is off limits.

An Overview of the Search Crawler

Search Crawler is a basic Web crawler for searching the Web, and it illustrates the fundamental structure of crawler-based applications. With Search Crawler, you can enter search criteria and then search the Web in real time, URL by URL, looking for matches to the criteria.

Search Crawler's interface, as shown in Figure 6-1, has three prominent sections, which we will refer to as *Search, Stats,* and *Matches*. The Search section at the top of the window has controls for entering search criteria, including the start URL for the search, the maximum number of URLs to crawl, and the search string. The search criteria can be additionally tweaked by choosing to limit the search to the site of the beginning URL and by selecting the Case Sensitive check box for the search string.

The Stats section, located in the middle of the window, has controls showing the current status of crawling when searching is underway. This section also has a progress bar to indicate the progress toward completing the search.

The Matches section at the bottom of the window has a table listing all the matches found by a search. These are the URLs of the Web pages that contain the search string.

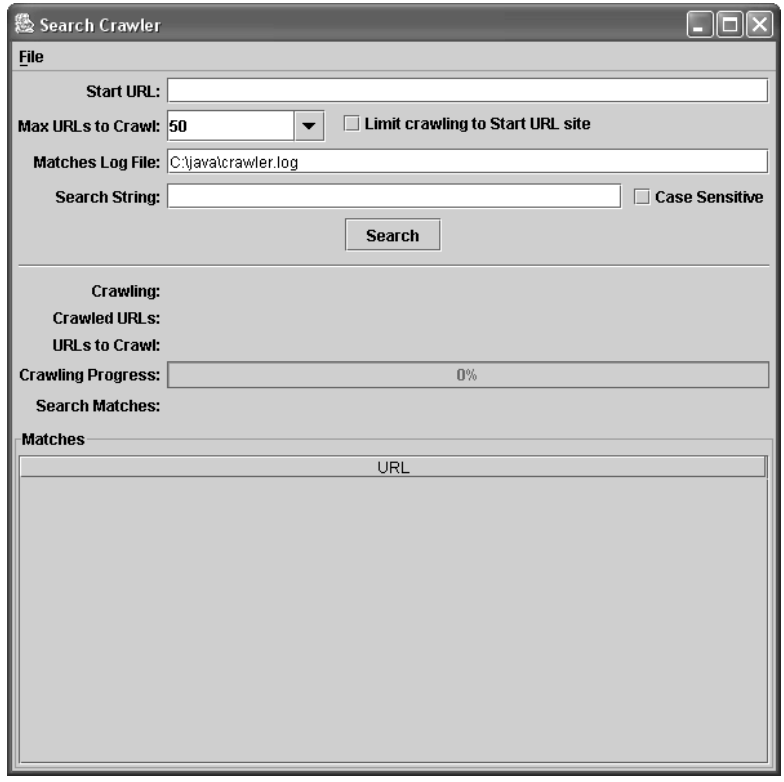

Figure 6-1 *The Search Crawler GUI interface*

The SearchCrawler Class

SearchCrawler has a **main( )** method, so on execution it will be invoked first. The **main( )** method instantiates a new **SearchCrawler** object and then calls its **show( )** method, which causes it to be displayed.

The **SearchCrawler** class is shown here and is examined in detail in the following sections. Notice that it extends **JFrame**:

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.regex.*;
import javax.swing.*;
import javax.swing.table.*;

// The Search Web Crawler
public class SearchCrawler extends JFrame
{
  // Max URLs drop-down values.
  private static final String[] MAX_URLS =
    {"50", "100", "500", "1000"};

  // Cache of robot disallow lists.
  private HashMap disallowListCache = new HashMap();

  // Search GUI controls.
  private JTextField startTextField;
  private JComboBox maxComboBox;
  private JCheckBox limitCheckBox;
  private JTextField logTextField;
  private JTextField searchTextField;
  private JCheckBox caseCheckBox;
  private JButton searchButton;

  // Search stats GUI controls.
  private JLabel crawlingLabel2;
  private JLabel crawledLabel2;
  private JLabel toCrawlLabel2;
  private JProgressBar progressBar;
  private JLabel matchesLabel2;

  // Table listing search matches.
  private JTable table;
```

```
// Flag for whether or not crawling is underway.
private boolean crawling;

// Matches log file print writer.
private PrintWriter logFileWriter;

// Constructor for Search Web Crawler.
public SearchCrawler()
{
  // Set application title.
  setTitle("Search Crawler");

  // Set window size.
  setSize(600, 600);

   // Handle window closing events.
  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      actionExit();
    }
  });

  // Set up File menu.
  JMenuBar menuBar = new JMenuBar();
  JMenu fileMenu = new JMenu("File");
  fileMenu.setMnemonic(KeyEvent.VK_F);
  JMenuItem fileExitMenuItem = new JMenuItem("Exit",
    KeyEvent.VK_X);
  fileExitMenuItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionExit();
    }
  });
  fileMenu.add(fileExitMenuItem);
  menuBar.add(fileMenu);
  setJMenuBar(menuBar);

  // Set up search panel.
  JPanel searchPanel = new JPanel();
  GridBagConstraints constraints;
  GridBagLayout layout = new GridBagLayout();
  searchPanel.setLayout(layout);

  JLabel startLabel = new JLabel("Start URL:");
  constraints = new GridBagConstraints();
```

```
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(startLabel, constraints);
searchPanel.add(startLabel);

startTextField = new JTextField();
constraints = new GridBagConstraints();
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 0, 5);
layout.setConstraints(startTextField, constraints);
searchPanel.add(startTextField);

JLabel maxLabel = new JLabel("Max URLs to Crawl:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(maxLabel, constraints);
searchPanel.add(maxLabel);

maxComboBox = new JComboBox(MAX_URLS);
maxComboBox.setEditable(true);
constraints = new GridBagConstraints();
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(maxComboBox, constraints);
searchPanel.add(maxComboBox);

limitCheckBox =
  new JCheckBox("Limit crawling to Start URL site");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.WEST;
constraints.insets = new Insets(0, 10, 0, 0);
layout.setConstraints(limitCheckBox, constraints);
searchPanel.add(limitCheckBox);

JLabel blankLabel = new JLabel();
constraints = new GridBagConstraints();
constraints.gridwidth = GridBagConstraints.REMAINDER;
layout.setConstraints(blankLabel, constraints);
searchPanel.add(blankLabel);

JLabel logLabel = new JLabel("Matches Log File:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 0, 0);
```

```
layout.setConstraints(logLabel, constraints);
searchPanel.add(logLabel);

String file =
  System.getProperty("user.dir") +
  System.getProperty("file.separator") +
  "crawler.log";
logTextField = new JTextField(file);
constraints = new GridBagConstraints();
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 0, 5);
layout.setConstraints(logTextField, constraints);
searchPanel.add(logTextField);

JLabel searchLabel = new JLabel("Search String:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(searchLabel, constraints);
searchPanel.add(searchLabel);

searchTextField = new JTextField();
constraints = new GridBagConstraints();
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.insets = new Insets(5, 5, 0, 0);
constraints.gridwidth= 2;
constraints.weightx = 1.0d;
layout.setConstraints(searchTextField, constraints);
searchPanel.add(searchTextField);

caseCheckBox = new JCheckBox("Case Sensitive");
constraints = new GridBagConstraints();
constraints.insets = new Insets(5, 5, 0, 5);
constraints.gridwidth = GridBagConstraints.REMAINDER;
layout.setConstraints(caseCheckBox, constraints);
searchPanel.add(caseCheckBox);

searchButton = new JButton("Search");
searchButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    actionSearch();
  }
});
constraints = new GridBagConstraints();
```

```
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 5, 5);
layout.setConstraints(searchButton, constraints);
searchPanel.add(searchButton);

JSeparator separator = new JSeparator();
constraints = new GridBagConstraints();
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 5, 5);
layout.setConstraints(separator, constraints);
searchPanel.add(separator);

JLabel crawlingLabel1 = new JLabel("Crawling:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(crawlingLabel1, constraints);
searchPanel.add(crawlingLabel1);

crawlingLabel2 = new JLabel();
crawlingLabel2.setFont(
  crawlingLabel2.getFont().deriveFont(Font.PLAIN));
constraints = new GridBagConstraints();
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 0, 5);
layout.setConstraints(crawlingLabel2, constraints);
searchPanel.add(crawlingLabel2);

JLabel crawledLabel1 = new JLabel("Crawled URLs:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(crawledLabel1, constraints);
searchPanel.add(crawledLabel1);

crawledLabel2 = new JLabel();
crawledLabel2.setFont(
  crawledLabel2.getFont().deriveFont(Font.PLAIN));
constraints = new GridBagConstraints();
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 0, 5);
```

```
layout.setConstraints(crawledLabel2, constraints);
searchPanel.add(crawledLabel2);

JLabel toCrawlLabel1 = new JLabel("URLs to Crawl:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(toCrawlLabel1, constraints);
searchPanel.add(toCrawlLabel1);

toCrawlLabel2 = new JLabel();
toCrawlLabel2.setFont(
  toCrawlLabel2.getFont().deriveFont(Font.PLAIN));
constraints = new GridBagConstraints();
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 0, 5);
layout.setConstraints(toCrawlLabel2, constraints);
searchPanel.add(toCrawlLabel2);

JLabel progressLabel = new JLabel("Crawling Progress:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 0, 0);
layout.setConstraints(progressLabel, constraints);
searchPanel.add(progressLabel);

progressBar = new JProgressBar();
progressBar.setMinimum(0);
progressBar.setStringPainted(true);
constraints = new GridBagConstraints();
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.insets = new Insets(5, 5, 0, 5);
layout.setConstraints(progressBar, constraints);
searchPanel.add(progressBar);

JLabel matchesLabel1 = new JLabel("Search Matches:");
constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.EAST;
constraints.insets = new Insets(5, 5, 10, 0);
layout.setConstraints(matchesLabel1, constraints);
searchPanel.add(matchesLabel1);
```

```
    matchesLabel2 = new JLabel();
    matchesLabel2.setFont(
      matchesLabel2.getFont().deriveFont(Font.PLAIN));
    constraints = new GridBagConstraints();
    constraints.fill = GridBagConstraints.HORIZONTAL;
    constraints.gridwidth = GridBagConstraints.REMAINDER;
    constraints.insets = new Insets(5, 5, 10, 5);
    layout.setConstraints(matchesLabel2, constraints);
    searchPanel.add(matchesLabel2);

    // Set up matches table.
    table =
      new JTable(new DefaultTableModel(new Object[][]{},
        new String[]{"URL"}) {
      public boolean isCellEditable(int row, int column)
      {
        return false;
      }
    });

    // Set up Matches panel.
    JPanel matchesPanel = new JPanel();
    matchesPanel.setBorder(
      BorderFactory.createTitledBorder("Matches"));
    matchesPanel.setLayout(new BorderLayout());
    matchesPanel.add(new JScrollPane(table),
      BorderLayout.CENTER);

    // Add panels to display.
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(searchPanel, BorderLayout.NORTH);
    getContentPane().add(matchesPanel, BorderLayout.CENTER);
  }

  // Exit this program.
  private void actionExit() {
    System.exit(0);
  }

  // Handle Search/Stop button being clicked.
  private void actionSearch() {
    // If stop button clicked, turn crawling flag off.
    if (crawling) {
      crawling = false;
      return;
```

```
}

ArrayList errorList = new ArrayList();

// Validate that start URL has been entered.
String startUrl = startTextField.getText().trim();
if (startUrl.length() < 1) {
  errorList.add("Missing Start URL.");
}
// Verify start URL.
else if (verifyUrl(startUrl) == null) {
  errorList.add("Invalid Start URL.");
}

// Validate that Max URLs is either empty or is a number.
int maxUrls = 0;
String max = ((String) maxComboBox.getSelectedItem()).trim();
if (max.length() > 0) {
  try {
    maxUrls = Integer.parseInt(max);
  } catch (NumberFormatException e) {
  }
  if (maxUrls < 1) {
    errorList.add("Invalid Max URLs value.");
  }
}

// Validate that matches log file has been entered.
String logFile = logTextField.getText().trim();
if (logFile.length() < 1) {
  errorList.add("Missing Matches Log File.");
}

// Validate that search string has been entered.
String searchString = searchTextField.getText().trim();
if (searchString.length() < 1) {
  errorList.add("Missing Search String.");
}

// Show errors, if any, and return.
if (errorList.size() > 0) {
  StringBuffer message = new StringBuffer();

  // Concatenate errors into single message.
  for (int i = 0; i < errorList.size(); i++) {
```

```
        message.append(errorList.get(i));
        if (i + 1 < errorList.size()) {
          message.append("\n");
        }
      }

      showError(message.toString());
      return;
    }

    // Remove "www" from start URL if present.
    startUrl = removeWwwFromUrl(startUrl);

    // Start the Search Crawler.
    search(logFile, startUrl, maxUrls, searchString);
  }

  private void search(final String logFile, final String startUrl,
    final int maxUrls, final String searchString)
  {
    // Start the search in a new thread.
    Thread thread = new Thread(new Runnable() {
      public void run() {
        // Show hour glass cursor while crawling is under way.
        setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));

        // Disable search controls.
        startTextField.setEnabled(false);
        maxComboBox.setEnabled(false);
        limitCheckBox.setEnabled(false);
        logTextField.setEnabled(false);
        searchTextField.setEnabled(false);
        caseCheckBox.setEnabled(false);

        // Switch Search button to "Stop."
        searchButton.setText("Stop");

        // Reset stats.
        table.setModel(new DefaultTableModel(new Object[][]{},
          new String[]{"URL"}) {
          public boolean isCellEditable(int row, int column)
          {
            return false;
          }
        });
```

```
updateStats(startUrl, 0, 0, maxUrls);

// Open matches log file.
try {
  logFileWriter = new PrintWriter(new FileWriter(logFile));
} catch (Exception e) {
  showError("Unable to open matches log file.");
  return;
}

// Turn crawling flag on.
crawling = true;

// Perform the actual crawling.
crawl(startUrl, maxUrls, limitCheckBox.isSelected(),
  searchString, caseCheckBox.isSelected());

// Turn crawling flag off.
crawling = false;

// Close matches log file.
try {
  logFileWriter.close();
} catch (Exception e) {
  showError("Unable to close matches log file.");
}

// Mark search as done.
crawlingLabel2.setText("Done");

// Enable search controls.
startTextField.setEnabled(true);
maxComboBox.setEnabled(true);
limitCheckBox.setEnabled(true);
logTextField.setEnabled(true);
searchTextField.setEnabled(true);
caseCheckBox.setEnabled(true);

// Switch search button back to "Search."
searchButton.setText("Search");

// Return to default cursor.
setCursor(Cursor.getDefaultCursor());

// Show message if search string not found.
```

```
      if (table.getRowCount() == 0) {
        JOptionPane.showMessageDialog(SearchCrawler.this,
          "Your Search String was not found. Please try another.",
          "Search String Not Found",
          JOptionPane.WARNING_MESSAGE);
      }
    }
  });
  thread.start();
}

// Show dialog box with error message.
private void showError(String message) {
  JOptionPane.showMessageDialog(this, message, "Error",
    JOptionPane.ERROR_MESSAGE);
}

// Update crawling stats.
private void updateStats(
  String crawling, int crawled, int toCrawl, int maxUrls)
{
  crawlingLabel2.setText(crawling);
  crawledLabel2.setText("" + crawled);
  toCrawlLabel2.setText("" + toCrawl);

  // Update progress bar.
  if (maxUrls == -1) {
    progressBar.setMaximum(crawled + toCrawl);
  } else {
    progressBar.setMaximum(maxUrls);
  }
  progressBar.setValue(crawled);

  matchesLabel2.setText("" + table.getRowCount());
}

// Add match to matches table and log file.
private void addMatch(String url) {
  // Add URL to matches table.
  DefaultTableModel model =
    (DefaultTableModel) table.getModel();
  model.addRow(new Object[]{url});

  // Add URL to matches log file.
  try {
```

```
    logFileWriter.println(url);
  } catch (Exception e) {
    showError("Unable to log match.");
  }
}

// Verify URL format.
private URL verifyUrl(String url) {
  // Only allow HTTP URLs.
  if (!url.toLowerCase().startsWith("http://"))
    return null;

  // Verify format of URL.
  URL verifiedUrl = null;
  try {
    verifiedUrl = new URL(url);
  } catch (Exception e) {
    return null;
  }

  return verifiedUrl;
}

// Check if robot is allowed to access the given URL.
private boolean isRobotAllowed(URL urlToCheck) {
  String host = urlToCheck.getHost().toLowerCase();

  // Retrieve host's disallow list from cache.
  ArrayList disallowList =
    (ArrayList) disallowListCache.get(host);

  // If list is not in the cache, download and cache it.
  if (disallowList == null) {
    disallowList = new ArrayList();

    try {
      URL robotsFileUrl =
        new URL("http://" + host + "/robots.txt");

      // Open connection to robot file URL for reading.
      BufferedReader reader =
        new BufferedReader(new InputStreamReader(
          robotsFileUrl.openStream()));

      // Read robot file, creating list of disallowed paths.
```

```
        String line;
        while ((line = reader.readLine()) != null) {
          if (line.indexOf("Disallow:") == 0) {
            String disallowPath =
              line.substring("Disallow:".length());

            // Check disallow path for comments and remove if present.
            int commentIndex = disallowPath.indexOf("#");
            if (commentIndex != - 1) {
              disallowPath =
                disallowPath.substring(0, commentIndex);
            }

            // Remove leading or trailing spaces from disallow path.
            disallowPath = disallowPath.trim();

            // Add disallow path to list.
            disallowList.add(disallowPath);
          }
        }

        // Add new disallow list to cache.
        disallowListCache.put(host, disallowList);
      }
      catch (Exception e) {
        /* Assume robot is allowed since an exception
           is thrown if the robot file doesn't exist. */
        return true;
      }
    }

    /* Loop through disallow list to see if
       crawling is allowed for the given URL. */
    String file = urlToCheck.getFile();
    for (int i = 0; i < disallowList.size(); i++) {
      String disallow = (String) disallowList.get(i);
      if (file.startsWith(disallow)) {
        return false;
      }
    }

    return true;
  }

  // Download page at given URL.
```

```
private String downloadPage(URL pageUrl) {
   try {
      // Open connection to URL for reading.
      BufferedReader reader =
        new BufferedReader(new InputStreamReader(
          pageUrl.openStream()));

      // Read page into buffer.
      String line;
      StringBuffer pageBuffer = new StringBuffer();
      while ((line = reader.readLine()) != null) {
        pageBuffer.append(line);
      }

      return pageBuffer.toString();
   } catch (Exception e) {
   }

   return null;
}

// Remove leading "www" from a URL's host if present.
private String removeWwwFromUrl(String url) {
  int index = url.indexOf("://www.");
  if (index != -1) {
    return url.substring(0, index + 3) +
      url.substring(index + 7);
  }

  return (url);
}

// Parse through page contents and retrieve links.
private ArrayList retrieveLinks(
  URL pageUrl, String pageContents, HashSet crawledList,
  boolean limitHost)
{
  // Compile link matching pattern.
  Pattern p =
    Pattern.compile("<a\\s+href\\s*=\\s*\"?(.*?)[\"|>]",
      Pattern.CASE_INSENSITIVE);
  Matcher m = p.matcher(pageContents);

  // Create list of link matches.
  ArrayList linkList = new ArrayList();
```

```
while (m.find()) {
  String link = m.group(1).trim();

  // Skip empty links.
  if (link.length() < 1) {
    continue;
  }

  // Skip links that are just page anchors.
  if (link.charAt(0) == '#') {
    continue;
  }

  // Skip mailto links.
  if (link.indexOf("mailto:") != -1) {
    continue;
  }

  // Skip JavaScript links.
  if (link.toLowerCase().indexOf("javascript") != -1) {
    continue;
  }

  // Prefix absolute and relative URLs if necessary.
  if (link.indexOf("://") == -1) {
    // Handle absolute URLs.
    if (link.charAt(0) == '/') {
      link = "http://" + pageUrl.getHost() + link;
    // Handle relative URLs.
    } else {
      String file = pageUrl.getFile();
      if (file.indexOf('/') == -1) {
        link = "http://" + pageUrl.getHost() + "/" + link;
      } else {
        String path =
          file.substring(0, file.lastIndexOf('/') + 1);
        link = "http://" + pageUrl.getHost() + path + link;
      }
    }
  }

  // Remove anchors from link.
  int index = link.indexOf('#');
  if (index != -1) {
    link = link.substring(0, index);
```

```
    }

    // Remove leading "www" from URL's host if present.
    link = removeWwwFromUrl(link);

    // Verify link and skip if invalid.
    URL verifiedLink = verifyUrl(link);
    if (verifiedLink == null) {
      continue;
    }

    /* If specified, limit links to those
       having the same host as the start URL. */
    if (limitHost &&
        !pageUrl.getHost().toLowerCase().equals(
          verifiedLink.getHost().toLowerCase()))
    {
      continue;
    }

    // Skip link if it has already been crawled.
    if (crawledList.contains(link)) {
      continue;
    }

    // Add link to list.
    linkList.add(link);
  }

  return (linkList);
}

/* Determine whether or not search string is
   matched in the given page contents. */
private boolean searchStringMatches(
  String pageContents, String searchString,
  boolean caseSensitive)
{
  String searchContents = pageContents;

  /* If case-sensitive search, lowercase
     page contents for comparison. */
  if (!caseSensitive) {
    searchContents = pageContents.toLowerCase();
  }
```

```
    // Split search string into individual terms.
    Pattern p = Pattern.compile("[\\s]+");
    String[] terms = p.split(searchString);

    // Check to see if each term matches.
    for (int i = 0; i < terms.length; i++) {
      if (caseSensitive) {
        if (searchContents.indexOf(terms[i]) == -1) {
          return false;
        }
      } else {
        if (searchContents.indexOf(terms[i].toLowerCase()) == -1) {
          return false;
        }
      }
    }

    return true;
  }

  // Perform the actual crawling, searching for the search string.
  public void crawl(
    String startUrl, int maxUrls, boolean limitHost,
    String searchString, boolean caseSensitive)
  {
    // Set up crawl lists.
    HashSet crawledList = new HashSet();
    LinkedHashSet toCrawlList = new LinkedHashSet();

    // Add start URL to the to crawl list.
    toCrawlList.add(startUrl);

    /* Perform actual crawling by looping
       through the To Crawl list. */
    while (crawling && toCrawlList.size() > 0)
    {
      /* Check to see if the max URL count has
          been reached, if it was specified.*/
      if (maxUrls != -1) {
        if (crawledList.size() == maxUrls) {
          break;
        }
      }

      // Get URL at bottom of the list.
```

```
String url = (String) toCrawlList.iterator().next();

// Remove URL from the To Crawl list.
toCrawlList.remove(url);

// Convert string url to URL object.
URL verifiedUrl = verifyUrl(url);

// Skip URL if robots are not allowed to access it.
if (!isRobotAllowed(verifiedUrl)) {
  continue;
}

// Update crawling stats.
updateStats(url, crawledList.size(), toCrawlList.size(),
  maxUrls);

// Add page to the crawled list.
crawledList.add(url);

// Download the page at the given URL.
String pageContents = downloadPage(verifiedUrl);

/* If the page was downloaded successfully, retrieve all its
   links and then see if it contains the search string. */
if (pageContents != null && pageContents.length() > 0)
{
  // Retrieve list of valid links from page.
  ArrayList links =
    retrieveLinks(verifiedUrl, pageContents, crawledList,
      limitHost);

  // Add links to the To Crawl list.
  toCrawlList.addAll(links);

  /* Check if search string is present in
     page, and if so, record a match. */
  if (searchStringMatches(pageContents, searchString,
       caseSensitive))
  {
    addMatch(url);
  }
}

// Update crawling stats.
```

```
        updateStats(url, crawledList.size(), toCrawlList.size(),
          maxUrls);
      }
    }

    // Run the Search Crawler.
    public static void main(String[] args) {
      SearchCrawler crawler = new SearchCrawler();
      crawler.show();
    }
}
```

The SearchCrawler Variables

SearchCrawler starts off by declaring several instance variables, most of which hold references to the interface controls. First, the **MAX_URLS String** array declares the list of values to be displayed in the Max URLs to Crawl combo box. Second, **disallowListCache** is defined for caching robot disallow lists so that they don't have to be retrieved for each URL being crawled. Next, each of the interface controls is declared for the Search, Stats, and Matches sections of the interface. After the interface controls have been declared, the **crawling** flag is defined for tracking whether or not crawling is underway. Finally, the **logFileWriter** instance variable, which is used for printing search matches to a log file, is declared.

The SearchCrawler Constructor

When the **SearchCrawler** is instantiated, all the interface controls are initialized inside its constructor. The constructor contains a lot of code, but most of it is straightforward. The following discussion gives an overview.

First, the application's window title is set with a call to **setTitle( )**. Next, the **setSize( )** call establishes the window's width and height in pixels. After that, a window listener is added by calling **addWindowListener( )**, which passes a **WindowAdapter** object that overrides the **windowClosing( )** event handler. This handler calls the **actionExit( )** method when the application's window is closed. Next, a menu bar with a File menu is added to the application's window.

The next several lines of the constructor initiate and lay out the interface controls. Similar to other applications in this book, the layout is arranged using the **GridBagLayout** class and its associated **GridBagConstraints** class. First, the Search section of the interface is laid out, followed by the Stats section. The Search section includes all the controls for entering the search criteria and constraints. The Stats section holds all the controls for displaying the current crawling status, such as how many URLs have been crawled and how many URLs are left to crawl.

It's important to point out three things in the Search and Stats sections. First, the Matches Log File text field control is initialized with a string containing a filename. This string is set

to a file called **crawler.log** in the directory the application is run from, as specified by the Java environment variable **user.dir**. Second, an **ActionListener** is added to the Search button so that the **actionSearch( )** method is called each time the button is clicked. Third, the font for each label that is used to display results is updated with a call to **setFont( )**. The **setFont( )** call is used to turn off the bolding of the label fonts so that they are distinguished in the interface.

Following the Search and Stats sections of the interface is the Matches section that consists of the matches table, which contains the URLs containing the search string. The matches table is instantiated with a new **DefaultTableModel** subclass passed to its constructor. Typically a fully qualified subclass of **DefaultTableModel** is used to customize the data model used by a **JTable**; however, in this case only the **isCellEditable( )** method needs to be implemented. The **isCellEditable( )** method instructs the table that no cells should be editable by returning **false**, regardless of the row and column specified.

Once the matches table is initialized, it is added to the Matches panel. Finally, the Search panel and Matches panel are added to the interface.

The actionSearch( ) Method

The **actionSearch( )** method is invoked each time the Search (or Stop) button is clicked. The **actionSearch( )** method starts with these lines of code:

```
// If stop button clicked, turn crawling flag off.
if (crawling) {
  crawling = false;
  return;
}
```

Since the Search button in the interface doubles as both the Search button and the Stop button, it's necessary to know which of the two buttons was clicked. When crawling is underway, the **crawling** flag is set to **true**. Thus if the **crawling** flag is **true** when the **actionsearch( )** method is invoked, the Stop button was clicked. In this scenario, the **crawling** flag is set to **false** and **actionSearch( )** returns so that the rest of the method is not executed.

Next, an **ArrayList** variable, **errorList**, is initialized:

```
ArrayList errorList = new ArrayList();
```

The **errorList** is used to hold any error messages generated by the next several lines of code that validate all required search fields have been entered.

It goes without saying that the Search Crawler will not function without a URL that specifies the location at which to start crawling. The following code verifies that a starting URL has been entered and that the URL is valid:

```
// Validate that the start URL has been entered.
String startUrl = startTextField.getText().trim();
if (startUrl.length() < 1) {
  errorList.add("Missing Start URL.");
```

```
}
// Verify start URL.
else if (verifyUrl(startUrl) == null) {
  errorList.add("Invalid Start URL.");
}
```

If either of these checks fails, an error message is added to the error list.

Next, the Max URLs to Crawl combo box value is validated:

```
// Validate that Max URLs is either empty or is a number.
int maxUrls = -1;
String max = ((String) maxComboBox.getSelectedItem()).trim();
if (max.length() > 0) {
  try {
    maxUrls = Integer.parseInt(max);
  } catch (NumberFormatException e) {
  }
  if (maxUrls < 1) {
    errorList.add("Invalid Max URLs value.");
  }
}
```

Validating the maximum number of URLs to crawl is a bit more involved than the other validations in this method. This is because the Max URLs to Crawl field can either contain a positive number that indicates the maximum number of URLs to crawl or can be left blank to indicate that no maximum should be used. Initially, **maxUrls** is defaulted to –1 to indicate no maximum. If the user enters something into the Max URLs to Crawl field, it is validated as being a valid numeric value with a call to **Integer.parseInt( )**. **Integer.parseInt( )** converts a **String** representation of an integer into an **int** value. If the **String** representation cannot be converted to an integer, a **NumberFormatException** is thrown and the **maxUrls** value is not set. Next, **maxUrls** is checked to see if it is less than 1. If so, an error is added to the error list.

Next, the Matches Log File and Search String fields are validated:

```
// Validate that the matches log file has been entered.
String logFile = logTextField.getText().trim();
if (logFile.length() < 1) {
  errorList.add("Missing Matches Log File.");
}

// Validate that the search string has been entered.
String searchString = searchTextField.getText().trim();
if (searchString.length() < 1) {
  errorList.add("Missing Search String.");
}
```

If either of these fields has not been entered, an error message is added to the error list.

The following code checks to see if any errors have been recorded during validation. If so, all the errors are concatenated into a single message and displayed with a call to **showError( )**.

```
// Show errors, if any, and return.
if (errorList.size() > 0) {
  StringBuffer message = new StringBuffer();

  // Concatenate errors into single message.
  for (int i = 0; i < errorList.size(); i++) {
    message.append(errorList.get(i));
    if (i + 1 < errorList.size()) {
      message.append("\n");
    }
  }

  showError(message.toString());
  return;
}
```

For efficiency's sake, a **StringBuffer** object (referred to by **message**) is used to hold the concatenated message. The error list is iterated over with a **for** loop, adding each message to **message**. Notice that each time a message is added, a check is performed to see if the message is the last in the list or not. If the message is not the last message in the list, a newline (**\n**) character is added so that each message will be displayed on its own line in the error dialog box shown with the **showError( )** method.

Finally, after all the field validations are successful, **actionSearch( )** concludes by removing "www" from the starting URL and then calling the **search( )** method:

```
// Remove "www" from start URL if present.
startUrl = removeWwwFromUrl(startUrl);

// Start the Search Crawler.
search(logFile, startUrl, maxUrls, searchString);
```

The search( ) Method

The **search( )** method is used to begin the Web crawling process. Since this process can take a considerable amount of time to complete, a new thread is created so that the search code can run independently. This frees up Swing's event thread, allowing changes in the interface to take place while crawling is underway.

The **search( )** method starts with these lines of code:

```
// Start the search in a new thread.
Thread thread = new Thread(new Runnable() {
  public void run() {
```

To run the search code in a separate thread, a new **Thread** object is instantiated with a **Runnable** instance passed to its constructor. Instead of creating a separate class that implements the **Runnable** interface, the code is in-lined.

Before the search starts, the interface controls are updated to indicate that crawling is underway, as shown here:

```
// Show hour glass cursor while crawling is under way.
setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));

// Disable search controls.
startTextField.setEnabled(false);
maxComboBox.setEnabled(false);
limitCheckBox.setEnabled(false);
logTextField.setEnabled(false);
searchTextField.setEnabled(false);
caseCheckBox.setEnabled(false);

// Switch search button to "Stop."
searchButton.setText("Stop");
```

First, the application's cursor is set to the **WAIT_CURSOR** to signify that the application is busy. On most operating systems, the **WAIT_CURSOR** is an hourglass. After the cursor has been set, each of the search interface controls is disabled by calling the **setEnabled( )** method with a **false** flag on the control. Next, the Search button is changed to read "Stop." The Search button is changed, because when searching is underway the button doubles as a control for stopping the current search.

After disabling the search controls, the Stats section of the interface is reset, as shown here:

```
// Reset stats.
table.setModel(new DefaultTableModel(new Object[][]{},
  new String[]{"URL"}) {
  public boolean isCellEditable(int row, int column)
  {
    return false;
  }
});
updateStats(startUrl, 0, 0, maxUrls);
```

First, the matches table's data model is reset by passing the **setModel( )** method an all new, empty **DefaultTableModel** instance. Second, the **updateStats( )** method is called to refresh the progress bar and the status labels.

Next, the log file is opened and the **crawling** flag is turned on:

```
// Open matches log file.
try {
  logFileWriter = new PrintWriter(new FileWriter(logFile));
} catch (Exception e) {
```

```
  showError("Unable to open matches log file.");
  return;
}

// Turn crawling flag on.
crawling = true;
```

The log file is opened by way of creating a new **PrintWriter** instance for writing to the file. If the file cannot be opened, an error dialog box is displayed by calling **showError( )**. The **crawling** flag is set to **true** to indicate to the **actionSearch( )** method that crawling is underway.

The following code kicks off the actual search crawling by invoking the **crawl( )** method:

```
// Perform the actual crawling.
crawl(startUrl, maxUrls, limitCheckBox.isSelected(),
  searchString, caseCheckBox.isSelected());
```

After crawling has completed, the **crawling** flag is turned off and the matches log file is closed, as shown here:

```
// Turn crawling flag off.
crawling = false;

// Close matches log file.
try {
  logFileWriter.close();
} catch (Exception e) {
  showError("Unable to close matches log file.");
}
```

The **crawling** flag is set to **false** to indicate that crawling is no longer underway. Next, the matches log file is closed since crawling is finished. Similar to opening the file, if an exception is thrown while trying to close the file, an error dialog box will be shown with a call to **showError( )**.

Because crawling is finished, the search controls are reactivated by the following code:

```
// Mark search as done.
crawlingLabel2.setText("Done");

// Enable search controls.
startTextField.setEnabled(true);
maxComboBox.setEnabled(true);
limitCheckBox.setEnabled(true);
logTextField.setEnabled(true);
searchTextField.setEnabled(true);
caseCheckBox.setEnabled(true);
```

```
// Switch search button back to "Search."
searchButton.setText("Search");

// Return to default cursor.
setCursor(Cursor.getDefaultCursor());
```

First, the Crawling field is updated to display "Done." Second, each of the search controls is reenabled. Third, the Stop button is reverted back to displaying "Search." Finally, the cursor is reverted back to the default application cursor.

If the search did not yield any matches, the following code displays a dialog box to indicate this fact:

```
// Show message if search string not found.
if (table.getRowCount() == 0) {
  JOptionPane.showMessageDialog(SearchCrawler.this,
    "Your Search String was not found. Please try another.",
    "Search String Not Found",
    JOptionPane.WARNING_MESSAGE);
}
```

The **search( )** method wraps up with the following lines of code:

```
  }
});
thread.start();
```

After the **Runnable** implementation's **run( )** method has been defined, the search thread is started with a call to **thread.start( )**. Upon the thread's execution, the **Runnable** instance's **run( )** method will be invoked.

The showError( ) Method

The **showError( )** method, shown here, displays an error dialog box on the screen with the given message. This method is invoked if any required search options are missing or if there are any problems opening, writing to, or closing the log file.

```
// Show dialog box with error message.
private void showError(String message) {
  JOptionPane.showMessageDialog(this, message, "Error",
    JOptionPane.ERROR_MESSAGE);
}
```

The updateStats( ) Method

The **updateStats( )** method, shown here, updates the values displayed in the Stats section of the interface:

```
// Update crawling stats.
private void updateStats(
  String crawling, int crawled, int toCrawl, int maxUrls)
{
  crawlingLabel2.setText(crawling);
  crawledLabel2.setText("" + crawled);
  toCrawlLabel2.setText("" + toCrawl);

  // Update progress bar.
  if (maxUrls == -1) {
    progressBar.setMaximum(crawled + toCrawl);
  } else {
    progressBar.setMaximum(maxUrls);
  }
  progressBar.setValue(crawled);

  matchesLabel2.setText("" + table.getRowCount());
}
```

First, the crawling results are updated to reflect the current URL being crawled, the number of URLs crawled thus far, and the number of URLs that are left to crawl. Take note that the URLs to Crawl field may be misleading. It displays the number of links that have been aggregated and put in the To Crawl queue, not the difference between the specified maximum URLs and the number of URLs that have been crawled thus far. Notice also that when **setText( )** is called with **crawled** and **toCrawl**, it is passed an empty string (" ") plus an **int** value. This is so that Java will convert the **int** values into **String** objects, which the **setText( )** method requires.

Next, the progress bar is updated to reflect the current progress made toward finishing crawling. If the Max URLs to Crawl text field was left blank, which specifies that crawling should not be capped, the **maxUrls** variable will have the value –1. In this case, the progress bar's maximum is set to the number of URLs that have been crawled plus the number of URLs left to crawl. If, on the other hand, a Max URLs to Crawl value was specified, it will be used as the progress bar's maximum. After establishing the progress bar's maximum value, its current value is set. The **JProgressBar** class uses the maximum and current values to calculate the percentage shown in text on the progress bar.

Finally, the Search Matches label is updated to reflect the current number of URLs that contain the specified search string.

The addMatch( ) Method

The **addMatch( )** method is called by the **crawl( )** method each time a match with the search string is found. The **addMatch( )** method, shown here, adds a URL to both the matches table and the log file:

```
// Add match to matches table and log file.
private void addMatch(String url) {
```

```
  // Add URL to matches table.
  DefaultTableModel model =
    (DefaultTableModel) table.getModel();
  model.addRow(new Object[]{url});

  // Add URL to matches log file.
  try {
    logFileWriter.println(url);
  } catch (Exception e) {
    showError("Unable to log match.");
  }
}
```

This method first adds the URL to the matches table by retrieving the table's data model and calling its **addRow( )** method. Notice that the **addRow( )** method takes an **Object** array as input. In order to satisfy that requirement, the **url String** object is wrapped in an **Object** array. After adding the URL to the matches table, the URL is written to the log file with a call to **logFileWriter.println( )**. This call is wrapped in a **try-catch** block; and if an exception is thrown, the **showError( )** method is called to alert the user that an error has occurred while trying to write to the log file.

The verifyUrl( ) Method

The **verifyUrl( )** method, shown here, is used throughout **SearchCrawler** to verify the format of a URL. Additionally, this method serves to convert a string representation of a URL into a **URL** object:

```
// Verify URL format.
private URL verifyUrl(String url) {
  // Only allow HTTP URLs.
  if (!url.toLowerCase().startsWith("http://"))
    return null;

  // Verify format of URL.
  URL verifiedUrl = null;
  try {
    verifiedUrl = new URL(url);
  } catch (Exception e) {
    return null;
  }

  return verifiedUrl;
}
```

This method first verifies that the given URL is an HTTP URL since only HTTP URLs are supported by Search Crawler. Next, the URL being verified is used to construct a new **URL**

object. If the URL is malformed, the **URL** class constructor will throw an exception resulting in **null** being returned from this method. A **null** return value is used to denote that the string passed to **url** is not valid or verified.

The isRobotAllowed( ) Method

The **isRobotAllowed( )** method fulfills the robot protocol. In order to fully explain this method, we'll review it line by line.

The **isRobotAllowed( )** method starts with these lines of code:

```
String host = urlToCheck.getHost().toLowerCase();

// Retrieve host's disallow list from cache.
ArrayList disallowList =
  (ArrayList) disallowListCache.get(host);

// If list is not in the cache, download and cache it.
if (disallowList == null) {
  disallowList = new ArrayList();
```

In order to efficiently check whether or not robots are allowed to access a URL, Search Crawler caches each host's disallow list after it has been retrieved. This significantly improves the performance of Search Crawler because it avoids downloading the disallow list for each URL being verified. Instead, it just retrieves the list from cache.

The disallow list cache is keyed on the host portion of a URL, so **isRobotAllowed( )** starts out by retrieving the **urlToCheck**'s host by calling its **getHost( )** method. Notice that **toLowerCase( )** is tacked on to the end of the **getHost( )** call. Lowercasing the host ensures that duplicate host entries are not placed in the cache. Take note that the host portion of URLs is case insensitive on the Internet; however, the cache keys are case-sensitive strings. After retrieving the **urlToCheck**'s host, an attempt to retrieve a disallow list from the cache for the host is made. If there is not a list in cache already, **null** is returned, signaling that the disallow list must be downloaded from the host. The process of retrieving the disallow list from a host starts by creating a new **ArrayList** object.

Next, the contents of the disallow list are populated, beginning with the following lines of code:

```
try {
  URL robotsFileUrl =
    new URL("http://" + host + "/robots.txt");

  // Open connection to robot file URL for reading.
  BufferedReader reader =
    new BufferedReader(new InputStreamReader(
      robotsFileUrl.openStream()));
```

As mentioned earlier, Web site owners wishing to prevent Web crawlers from crawling their site, or portions of their site, must have a file called **robots.txt** at the root of their Web site hierarchy. The host portion of the **urlToCheck** is used to construct a URL, **robotsFileUrl**, pointing to the **robots.txt** file. Then a **BufferedReader** object is created for reading the contents of the **robots.txt** file. The **BufferedReader**'s constructor is passed an instance of **InputStreamReader**, whose constructor is passed the **InputStream** object returned from calling **robotsFileUrl.openStream( )**.

The following sequence sets up a **while** loop for reading the contents of the **robots.txt** file:

```
// Read robot file, creating list of disallowed paths.
String line;
while ((line = reader.readLine()) != null) {
  if (line.indexOf("Disallow:") == 0) {
    String disallowPath =
      line.substring("Disallow:".length());
```

The loop reads the contents of the file, line by line, until the **reader.readLine( )** method returns **null**, signaling that all lines have been read. Each line that is read is checked to see if it has a Disallow statement by using the **indexOf( )** method defined by **String**. If the line does in fact have a Disallow statement, the disallow path is culled from the line by taking a substring of the line from the point where the string "Disallow:" ends.

As discussed, comments can be interspersed in the **robots.txt** file by using a hash (#) character followed by a comment. Since comments will interfere with the Disallow statement comparisons, they are removed in the following lines of code:

```
    // Check disallow path for comments and remove if present.
    int commentIndex = disallowPath.indexOf("#");
    if (commentIndex != - 1) {
      disallowPath =
        disallowPath.substring(0, commentIndex);
    }

    // Remove leading or trailing spaces from disallow path.
    disallowPath = disallowPath.trim();

    // Add disallow path to list.
    disallowList.add(disallowPath);
  }
}
```

First, the disallow path is searched to see if it contains a hash character. If it does, the disallow path is substringed, removing the comment from the end of the string. After checking and potentially removing a comment from the disallow path, **disallowPath.trim( )** is called to

remove any leading or trailing space characters. Similar to comments, extraneous space characters will trip up comparisons, so they are removed. Finally, the disallow path is added to the list of disallow paths.

After the disallow path list has been created, it is added to the disallow list cache, as shown here:

```
    // Add new disallow list to cache.
    disallowListCache.put(host, disallowList);
  }
  catch (Exception e) {
    /* Assume robot is allowed since an exception
       is thrown if the robot file doesn't exist. */
    return true;
  }
}
```

The disallow path is added to the disallow list cache so that subsequent requests for the list can be quickly retrieved from cache instead of having to be downloaded again.

If an error occurs while opening the input stream to the robot file URL or while reading the contents of the file, an exception will be thrown. Since an exception will be thrown if the **robots.txt** file does not exist, we'll assume robots are allowed if an exception is thrown. Normally, the error checking in this scenario should be more robust; however, for simplicity and brevity's sake, we'll make the blanket decision that robots are allowed.

Next, the following code iterates over the disallow list to see if the **urlToCheck** is allowed or not:

```
/* Loop through disallow list to see if the
   crawling is allowed for the given URL. */
String file = urlToCheck.getFile();
for (int i = 0; i < disallowList.size(); i++) {
  String disallow = (String) disallowList.get(i);
  if (file.startsWith(disallow)) {
    return false;
  }
}

return true;
```

Each iteration of the **for** loop checks to see if the file portion of the **urlToCheck** is found in the disallow list. If the **urlToCheck**'s file does in fact match one of the statements in the disallow list, then **false** is returned, indicating that crawlers are not allowed to crawl the given URL. However, if the list is iterated over and no match is made, **true** is returned, indicating that crawling is allowed.

The downloadPage( ) Method

The **downloadPage( )** method, shown here, simply does as its name implies: it downloads the Web page at the given URL and returns the contents of the page as a large string:

```
// Download page at given URL.
private String downloadPage(URL pageUrl) {
   try {
      // Open connection to URL for reading.
      BufferedReader reader =
        new BufferedReader(new InputStreamReader(
          pageUrl.openStream()));

      // Read page into buffer.
      String line;
      StringBuffer pageBuffer = new StringBuffer();
      while ((line = reader.readLine()) != null) {
        pageBuffer.append(line);
      }

      return pageBuffer.toString();
   } catch (Exception e) {
   }

   return null;
}
```

Downloading Web pages from the Internet in Java is quite simple, as evidenced by this method. First, a **BufferedReader** object is created for reading the contents of the page at the given URL. The **BufferedReader**'s constructor is passed an instance of **InputStreamReader**, whose constructor is passed the **InputStream** object returned from calling **pageUrl.openStream( )**. Next, a **while** loop is used to read the contents of the page, line by line, until the **reader.readLine( )** method returns **null**, signaling that all lines have been read. Each line that is read with the **while** loop is added to the **pageBuffer StringBuffer** instance. After the page has been downloaded, its contents are returned as a **String** by calling **pageBuffer.toString( )**.

If an error occurs when opening the input stream to the page URL or while reading the contents of the Web page, an exception will be thrown. This exception will be caught by the empty catch block. The **catch** block has purposefully been left blank so that execution will continue to the remaining **return null** line. A return value of **null** from this method indicates to callers that an error occurred.

The removeWwwFromUrl( ) Method

The **removeWwwFromUrl( )** method is a simple utility method used to remove the "www" portion of a URL's host. For example, take the URL:

http://www.osborne.com

This method removes the "www." piece of the URL, yielding:

http://osborne.com

Because many Web sites intermingle URLs that do and don't start with "www", the Search Crawler uses this technique to find the "lowest common denominator" URL. Effectively, both URLs are the same on most Web sites, and having the lowest common denominator allows the Search Crawler to skip over duplicate URLs that would otherwise be redundantly crawled.

The **removeWwwFromUrl( )** method is shown here:

```
// Remove leading "www" from a URL's host if present.
private String removeWwwFromUrl(String url) {
  int index = url.indexOf("://www.");
  if (index != -1) {
    return url.substring(0, index + 3) +
      url.substring(index + 7);
  }

  return (url);
}
```

The **removeWwwFromUrl( )** method starts out by finding the index of "://www." inside the string passed to **url**. The "://" at the beginning of the string passed to the **indexOf( )** method indicates that "www" should be found at the beginning of a URL where the protocol is defined (for example, http://www.osborne.com). This way, URLs that simply contain the string "www" are not tampered with. If **url** contains "://www.", the characters before and after "www." are concatenated and returned. Otherwise, the string passed to **url** is returned.

The retrieveLinks( ) Method

The **retrieveLinks( )** method parses through the contents of a Web page and retrieves all the relevant links. The Web page for which links are being retrieved is stored in a large **String** object. To say the least, parsing through this string, looking for specific character sequences, would be quite cumbersome using the methods defined by the **String** class. Fortunately,

beginning with Java 2, v1.4, Java comes standard with a regular expression API library that makes easy work of parsing through strings.

The regular expression API is contained in **java.util.regex**. The topic of regular expressions is fairly large, and a complete discussion is beyond the scope of this book. However, because parsing regular expressions is key to Search Crawler, a brief overview is presented here.

An Overview of Regular Expression Processing

As the term is used here, a *regular expression* is a sequence of characters that describes a character sequence. This general description, called a *pattern,* can then be used to find matches in other character sequences. Regular expressions can specify wildcard characters, sets of characters, and various quantifiers. Thus, you can specify a regular expression that represents a general form that can match several different specific character sequences. There are two classes that support regular expression processing: **Pattern** and **Matcher**. You use **Pattern** to define a regular expression. To match the pattern against another sequence, use **Matcher**.

The **Pattern** class defines no constructors. Instead, a pattern is created by calling the **compile( )** factory method. The form used here is

```
static Pattern compile(String pattern, int options)
```

Here, *pattern* is the regular expression that you want to use, and *options* specifies one or more options that affect matching. The option used by Search Crawler is **Pattern.CASE_INSENSITIVE**, which causes the case of the strings to be ignored. The **compile( )** method transforms the string in *pattern* into a pattern that can be used for pattern matching by the **Matcher** class. It returns a **Pattern** object that contains the pattern.

Once you have created a **Pattern** object, you will use it to create a **Matcher**. This is done by calling the **matcher( )** factory method defined by **Pattern**. It is shown here:

```
Matcher matcher(CharSequence str)
```

Here, *str* is the character sequence that the pattern will be matched against. This is called the *input sequence*. **CharSequence** is an interface that was added by Java 2, v1.4 and defines a read-only set of characters. It is implemented by the **String** class, among others. Thus, you can pass a string to **matcher( )**.

You will use methods defined by **Matcher** to perform various pattern-matching operations. The ones used by **retrieveLinks( )** are **find( )** and **group( )**. The **find( )** method determines if a subsequence of the input sequence matches the pattern. The version used by Search Crawler is shown here:

```
boolean find( )
```

It returns **true** if there is a matching subsequence and **false** otherwise. This method can be called repeatedly, allowing it to find all matching subsequences. Each call to **find( )** begins where the previous one left off.

You can obtain a string containing a matching sequence by calling **group( )**. The form used by Search Crawler is shown here:

String group(int *which*)

Here, *which* specifies the sequence (group of characters), with the first group being 1. The matching string is returned.

Regular Expression Syntax

The syntax and rules that define a regular expression are similar to those used by Perl 5. Although no single rule is complicated, there are a large number of them, and a complete discussion is beyond the scope of this book. However, a few of the more commonly used constructs are described here.

In general, a regular expression is comprised of normal characters, character classes (sets of characters), wildcard characters, and quantifiers. A normal character is matched as is. Thus, if a pattern consists of "xy", the only input sequence that will match it is "xy". Characters such as newlines and tabs are specified using the standard escape sequences, which begin with a backslash (\). For example, a newline is specified by **\n**. In the language of regular expressions, a normal character is also called a *literal*.

A character class is a set of characters. A character class is specified by putting the characters in the class between brackets. For example, the class **[wxyz]** matches w, x, y, or z. To specify an inverted set, precede the characters with a circumflex (^). For example, **[^wxyz]** matches any character except w, x, y, or z. You can specify a range of characters using a hyphen. For example, to specify a character class that will match the digits 1 through 9, use **[1–9]**.

The wildcard character is the dot (**.**), and it matches any character. Thus, a pattern that consists of "." will match these (and other) input sequences: "A", "a", "x", and so on.

A quantifier determines how many times an expression is matched. The quantifiers are shown here:

+	Match one or more.
*	Match zero or more.
?	Match zero or one.

For example, the pattern **"x+"** will match **"x"**, **"xx"**, and **"xxx"**, among others.

A Close Look at retrieveLinks( )

The **retrieveLinks( )** method uses the regular expression API to obtain the links from a page. It begins with these lines of code:

```
// Compile link matching pattern.
Pattern p =
  Pattern.compile("<a\\s+href\\s*=\\s*\"?(.*?)[\"|>]",
    Pattern.CASE_INSENSITIVE);
Matcher m = p.matcher(pageContents);
```

The regular expression used to obtain links can be broken down as a series of steps, as shown in the following table:

Character Sequence	Explanation
<a	Look for the characters "<a".
\\s+	Look for one or more space characters.
href	Look for the characters “href”.
\\s*	Look for zero or more space characters.
=	Look for the character "=".
\\s*	Look for zero or more space characters.
\"?	Look for zero or one quote character.
(.*?)	Look for zero or more of any character until the next part of the pattern is matched, and place the results in a group.
[\”|>]	Look for quote character or greater than (">") character.

Notice that **Pattern.CASE_INSENSITIVE** is passed to the pattern compiler. As mentioned, this indicates that the pattern should ignore case when searching for matches.

Next, a list to hold the links is created, and the search for the links begins, as shown here:

```
// Create list of link matches.
ArrayList linkList = new ArrayList();
while (m.find()) {
  String link = m.group(1).trim();
```

Each link is found by cycling through **m** with a **while** loop. The **find( )** method of **Matcher** returns **true** until no more matches are found. Each match (link) found is retrieved by calling the **group( )** method defined by **Matcher**. Notice that **group( )** takes 1 as an argument. This specifies that the first group from the matching sequences be returned. Notice also that **trim( )** is called on the return value from the **group( )** method. This removes any unnecessary leading or trailing space from the value.

Many of the links found in Web pages are not suited for crawling. The following code filters out several links that the Search Crawler is uninterested in:

```
// Skip empty links.
if (link.length() < 1) {
  continue;
}

// Skip links that are just page anchors.
if (link.charAt(0) == '#') {
  continue;
}

// Skip mailto links.
```

```
if (link.indexOf("mailto:") != -1) {
  continue;
}

// Skip JavaScript links.
if (link.toLowerCase().indexOf("javascript") != -1) {
  continue;
}
```

First, empty links are skipped so as not to waste any more time on them. Second, links that are simply anchors into a page are skipped by checking to see if the first character of the link is a hash (#).

Page anchors allow for links to be made to a certain section of a page. Take, for example, this URL:

http://osborne.com/#contact

This URL has an anchor to the "contact" section of the page located at http://osborne.com. Links inside the page at http://osborne.com can reference the section relatively as just "#contact". Since anchors are not links to "new" pages, they are skipped over.

Next, "mailto" links are skipped. Mailto links are used for specifying an e-mail link in a Web page. For example, the link

mailto:books@osborne.com

is a mailto link. Since mailto links don't point to Web pages and cannot be crawled, they are skipped over. Finally, JavaScript links are skipped. JavaScript is a scripting language that can be embedded in Web pages for adding interactive functionality to the page. Additionally, JavaScript functionality can be accessed from links. Similar to mailto links, JavaScript links cannot be crawled; thus they are overlooked.

As you've just seen, the links in Web pages can take many formats, such as mailto and JavaScript formats. Additionally, traditional links inside Web pages can take a few different formats as well. Following are the three formats that traditional links can take:

- http://osborne.com/books/ArtofJava
- /books/ArtofJava
- books/ArtofJava

The first of the three links shown here is considered to be a fully qualified URL. The second example is a shortened version of the first URL, omitting the "host" portion of the URL. Notice the slash (/) at the beginning of the URL. The slash indicates that the URL is what's called "absolute." *Absolute URLs* are URLs that start at the root of a Web site. The third example is again a shortened version of the first URL, omitting the "host" portion of the URL. Notice that this third example does not have the leading slash. Since the leading

slash is absent, the URL is considered to be "relative." *Relative,* in the realm of URLs, means that the URL address is relative to the URL on which the link is found.

The lines of code in the next section handle converting absolute and relative links into fully qualified URLs:

```
// Prefix absolute and relative URLs if necessary.
if (link.indexOf("://") == -1) {
  // Handle absolute URLs.
  if (link.charAt(0) == '/') {
    link = "http://" + pageUrl.getHost() + link;
  // Handle relative URLs.
  } else {
    String file = pageUrl.getFile();
    if (file.indexOf('/') == -1) {
      link = "http://" + pageUrl.getHost() + "/" + link;
    } else {
      String path =
        file.substring(0, file.lastIndexOf('/') + 1);
      link = "http://" + pageUrl.getHost() + path + link;
    }
  }
}
```

First, the link is checked to see whether or not it is fully qualified by looking for the presence of "://" in the link. If these characters exist, the URL is assumed to be fully qualified. However, if they are not present, the link is converted to a fully qualified URL. As discussed, links beginning with a slash (/) are absolute, so this code adds "http://" and the current page's URL host to the link to fully qualify it. Relative links are converted here in a similar fashion.

For relative links, the current page URL's filename is taken and checked to see if it contains a slash (/). A slash in the filename indicates that the file is in a directory hierarchy. For example, a file may look like this:

dir1/dir2/file.html

or simply like this:

file.html

In the latter case, "http://", the current page's URL host, and "/" are added to the link since the current page is at the root of the Web site. In the former case, the "path" (or directory) portion of the filename is retrieved to create the fully qualified URL. This case concatenates "http://", the current page's URL host, the path, and the link together to create a fully qualified URL.

Next, page anchors and "www" are removed from the fully qualified link:

```
// Remove anchors from link.
int index = link.indexOf('#');
if (index != -1) {
  link = link.substring(0, index);
}

// Remove leading "www" from URL's host if present.
link = removeWwwFromUrl(link);
```

For the same reason that anchor-only links are skipped over, links with anchors tacked on to the end are skipped over. The leading "www" is also removed from links so that duplicate links are skipped over later in this method.

Next, the link is verified to make sure it is a valid URL:

```
// Verify link and skip if invalid.
URL verifiedLink = verifyUrl(link);
if (verifiedLink == null) {
  continue;
}
```

After validating that the link is a URL, the following code checks to see if the link's host is the same as the one specified by Start URL and checks to see if the link has already been crawled:

```
/* If specified, limit links to those
   having the same host as the start URL. */
if (limitHost &&
    !pageUrl.getHost().toLowerCase().equals(
      verifiedLink.getHost().toLowerCase()))
{
  continue;
}

// Skip link if it has already been crawled.
if (crawledList.contains(link)) {
  continue;
}
```

Finally, the **retrieveLinks( )** method ends by adding each link that passes all filters to the link list.

```
  // Add link to list.
  linkList.add(link);
```

```
}

return (linkList);
```

After the **while** loop finishes and all links have been added to the link list, the link list is returned.

The searchStringMatches( ) Method

The **searchStringMatches( )** method, shown here, is used to search through the contents of a Web page downloaded during crawling, determining whether or not the specified search string is present in the page:

```
/* Determine whether or not search string is
   present in the given page contents. */
private boolean searchStringMatches(
  String pageContents, String searchString,
  boolean caseSensitive)
{
  String searchContents = pageContents;

  /* If case-sensitive search, lowercase
     page contents before comparison. */
  if (!caseSensitive) {
    searchContents = pageContents.toLowerCase();
  }

  // Split search string into individual terms.
  Pattern p = Pattern.compile("[\\s]+");
  String[] terms = p.split(searchString);

  // Check to see if each term matches.
  for (int i = 0; i < terms.length; i++) {
    if (caseSensitive) {
      if (searchContents.indexOf(terms[i]) == -1) {
        return false;
      }
    } else {
      if (searchContents.indexOf(terms[i].toLowerCase()) == -1) {
        return false;
      }
    }
  }

  return true;
}
```

Because the search string can be either case insensitive (default) or case sensitive, **searchStringMatches( )** starts out by declaring a local variable, **searchContents**, that refers to the string to be searched. By default, the **pageContents** variable is assigned to **searchContents**. If the search is case sensitive, however, the **searchContents** variable is set to a lowercased version of the **pageContents** string.

Next, the search string is split into individual search terms using Java's regular expression library. To split the search string, first, a regular expression pattern is compiled with the **Pattern** object's static **compile( )** method. The pattern used here, "[\\s]+", states that one or more white space characters (that is, spaces, tabs, or newlines) should be matched. Second, the compiled **Pattern**'s **split( )** method is invoked with the search string, which yields a **String** array containing individual search terms.

After breaking the search string up, the individual terms are cycled through, checking to see if each term is found in the page's contents. The **indexOf( )** method defined by **String** is used to search through the **searchContents** variable. A return value of –1 indicates that the search term was not found, and thus **false** is returned since all terms must be found in order to have a match. Notice that if the search is case insensitive, the search term is lowercased in the comparison. This coincides with the value assigned to the **searchContents** variable at the beginning of this method. If the **for** loop executes in its entirety, the **searchStringMatches( )** method concludes by returning **true**, indicating that all terms in the search string matched.

The crawl( ) Method

The **crawl( )** method is the core of the search Web crawler because it performs the actual crawling. It begins with these lines of code:

```
// Set up crawl lists.
HashSet crawledList = new HashSet();
LinkedHashSet toCrawlList = new LinkedHashSet();

// Add start URL to the To Crawl list.
toCrawlList.add(startUrl);
```

There are several techniques that can be employed to crawl Web sites, recursion being a natural choice because crawling itself is recursive. Recursion, however, can be quite resource intensive, so the Search Crawler uses a queue technique. Here, **toCrawlList** is initialized to hold the queue of links to crawl. The start URL is then added to **toCrawlList** to begin the crawling process.

After initializing the To Crawl list and adding the start URL, crawling begins with a **while** loop set up to run until the **crawling** flag is turned off or until the To Crawl list has been exhausted, as shown here:

```
/* Perform actual crawling by looping
   through the To Crawl list. */
while (crawling && toCrawlList.size() > 0)
{
  /* Check to see if the max URL count has
```

```
     been reached, if it was specified.*/
  if (maxUrls != -1) {
    if (crawledList.size() == maxUrls) {
      break;
    }
  }
```

Remember that the **crawling** flag is used to stop crawling prematurely. If the Stop button on the interface is clicked during crawling, **crawling** is set to **false**. The next time the **while** loop's expression is evaluated, the loop will end because the **crawling** flag is **false**. The first section of code inside the **while** loop checks to see if the crawling limit specified by **maxUrls** has been reached. This check is performed only if the **maxUrls** variable has been set, as indicated by a value other than –1.

Upon each iteration of the **while** loop, the following code is executed:

```
// Get URL at bottom of the list.
String url = (String) toCrawlList.iterator().next();

// Remove URL from the To Crawl list.
toCrawlList.remove(url);

// Convert string url to URL object.
URL verifiedUrl = verifyUrl(url);

// Skip URL if robots are not allowed to access it.
if (!isRobotAllowed(verifiedUrl)) {
  continue;
}
```

First, the URL at the bottom of the To Crawl list is "popped" off. Thus, the list works in a first in, first out (FIFO) manner. Since the URLs are stored in a **LinkedHashSet** object, there is not actually a "pop" method. Instead, the functionality of a pop method is simulated by first retrieving the value at the bottom of the list with a call to **toCrawlList.iterator( ).next( )**. Then the URL retrieved from the list is removed from the list by calling **toCrawlList.remove( )**, passing in the URL as an argument.

After retrieving the next URL from the To Crawl list, the string representation of the URL is converted to a **URL** object using the **verifyUrl( )** method. Next, the URL is checked to see whether or not it is allowed to be crawled by calling the **isRobotAllowed( )** method. If the crawler is not allowed to crawl the given URL, then **continue** is executed to skip to the next iteration of the **while** loop.

After retrieving and verifying the next URL on the crawl list, the results are updated in the Stats section, as shown here:

```
// Update crawling stats.
updateStats(url, crawledList.size(), toCrawlList.size(),
```

```
  maxUrls);

// Add page to the crawled list.
crawledList.add(url);

// Download the page at the given URL.
String pageContents = downloadPage(verifiedUrl);
```

The output is updated with a call to **updateStats( )**. The URL is then added to the crawled list, indicating that it has been crawled and that subsequent references to the URL should be skipped. Next, the page at the given URL is downloaded with a call to **downloadPage( )**.

If the **downloadPage( )** method successfully downloads the page at the given URL, the following code is executed:

```
/* If the page was downloaded successfully, retrieve all of its
   links and then see if it contains the search string. */
if (pageContents != null && pageContents.length() > 0)
{
  // Retrieve list of valid links from page.
  ArrayList links =
    retrieveLinks(verifiedUrl, pageContents, crawledList,
      limitHost);

  // Add links to the To Crawl list.
  toCrawlList.addAll(links);

  /* Check if search string is present in
     page, and if so, record a match. */
  if (searchStringMatches(pageContents, searchString,
       caseSensitive))
  {
    addMatch(url);
  }
}
```

First, the page links are retrieved by calling the **retrieveLinks( )** method. Each of the links returned from the **retrieveLinks( )** call is then added to the To Crawl list. Next, the downloaded page is searched to see if the search string is found in the page with a call to **searchStringMatches( )**. If the search string is found in the page, the page is recorded as a match with the **addMatch( )** method.

The **crawl( )** method finishes by calling **updateStats( )** again at the end of the **while** loop:

```
  // Update crawling stats.
  updateStats(url, crawledList.size(), toCrawlList.size(),
    maxUrls);
}
```

The first call to **updateStats( )**, earlier in this method, updates the label that indicates which URL is being crawled. This second call updates all the other values because they will have changed since the first call.

Compiling and Running the Search Web Crawler

As mentioned earlier, **SearchCrawler** takes advantage of Java's new regular expression package: **java.util.regex**. The regular expression package was introduced in JDK 1.4; thus you will need to use JDK 1.4 or later to compile and run **SearchCrawler**.

Compile **SearchCrawler** like this:

```
javac SearchCrawler.java
```

Run **SearchCrawler** like this:

```
javaw SearchCrawler
```

Search Crawler has a simple, yet feature-rich, interface that's easy to use. First, in the Start URL field, enter the URL at which you want your search to begin. Next, choose the maximum number of URLs you want to crawl and whether or not you want to limit crawling to only the Web site specified in the Start URL field. If you want the crawler to continue crawling until it has exhausted all links it finds, you can leave the Max URLs to Crawl field blank. Be forewarned, however, that choosing not to set a maximum value will likely result in a search that will run for a very long time.

Next you'll notice a Matches Log File specified for you. This text field is prepopulated, specifying that the log file be written to a file called **crawler.log** in the directory that you are running the Search Crawler from. If you'd like to have the log file written to a different file, simply enter the new filename. Next, enter the string you want to search for and then select whether or not you want the search to be case sensitive. Note that entering a search string containing multiple words requires matching pages to include all the words specified.

Once you have entered all your search criteria and configured the search constraints, click the Search button. You'll notice that the search controls become disabled and the Search button changes to a Stop button. After searching has completed, the search controls will be reenabled, and the Stop button will revert back to being the Search button. Clicking the Stop button will cause the crawler to stop crawling after it has finished crawling the URL it is currently crawling. Figure 6-2 shows the Search Crawler in action.

A few key points about Search Crawler's functionality:

- Only HTTP links are supported, not HTTPS or FTP.
- URLs that redirect to another URL are not supported.
- Similar links such as "http://osborne.com" and "http://osborne.com/" (notice the trailing slash) are treated as separate unique links. This is because the Search Crawler cannot categorically know that both are the same in all instances.

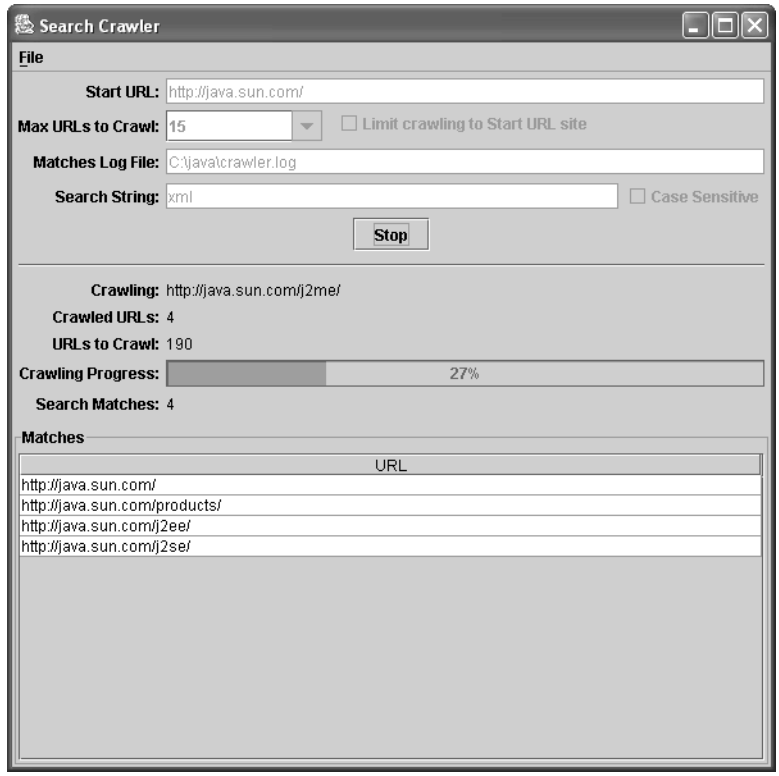

Figure 6-2 *The Search Crawler in action*

Web Crawler Ideas

Search Crawler is an excellent illustration of Java's networking capabilities. It is also an application that demonstrates the core technology associated with Web crawling. As mentioned at the start of this chapter, although Search Crawler is useful as is, its greatest benefit is as a starting point for your own crawler-based projects.

To begin, you might try enhancing Search Crawler. Try changing the way it follows links, perhaps to use depth-first crawling rather than breadth-first. Also try adding support for URLs that redirect to other URLs. Experiment with optimizing the search, perhaps by using additional threads to download multiple pages at the same time.

Next, you will want to try creating your own crawler-based projects. Here are some ideas:

- **Broken Link Crawler** A broken link crawler could be used to crawl a Web site and find any links that are broken. Each broken link would be recorded. At the end of crawling, a report would be generated listing each page that had broken links along

with a breakdown of each broken link on that page. This application would be especially useful for large Web sites where there are hundreds if not thousands of pages to check for broken links.

- **Comparison Crawler** A comparison crawler could be used to crawl several Web sites in an effort to find the lowest price for a list of products. For example, a comparison crawler might visit Amazon.com, Barnes&Noble.com, and a few others to find the lowest prices for books. This technique is often called "screen scraping" and is used to compare the price of many different types of goods on the Internet.
- **Archiver Crawler** An archiver crawler could be used to crawl a site and save or "archive" all of its pages. There are many reasons for archiving a site, including having the ability to view the site offline, creating a backup, or obtaining a snapshot in time of the Web site. In fact, a search engine's use of crawler technology is actually in the capacity of an archiver crawler. Search engines crawl the Internet and save all the pages along the way. Afterward they go back and sift through the data and index it so it can be searched rapidly.

As the preceding ideas show, crawler-based technology is useful in a wide variety of Web-based applications.