Chapter 21. Using Applets As Front Ends to Server-Side Programs

Topics in This Chapter

- Sending `GET` data and having the browser display the results
- Sending `GET` data and processing the results within the applet (HTTP tunneling)
- Using object serialization to exchange high-level data structures between applets and servlets
- Sending `POST` data and processing the results within the applet (HTTP tunneling)
- Bypassing the HTTP server altogether

HTML forms, discussed in Chapter 18, provide a simple but limited way of collecting user input and transmitting it to a servlet or CGI program. Occasionally, however, a more sophisticated user interface is required. Applets give you more control over the size, color, and font of the GUI controls; provide more built-in capability (sliders, line drawing, pop-up windows, and the like); let you track mouse and keyboard events; support the development of custom input forms (dials, thermometers, draggable icons, and so forth); and let you send a single user submission to multiple server-side programs. This extra capability comes at a cost, however, as it tends to require more effort to develop an interface with the Java programming language than it does with HTML forms, particularly if the interface contains a lot of formatted text. So, the choice between HTML forms and applets depends upon the application.

With HTML forms, `GET` and `POST` requests are handled almost exactly the same way. All the input elements are identical; only the `METHOD` attribute of the `FORM` element needs to change. With applets, however, there are three distinct approaches. In the first approach, covered in Section 21.1, the applet imitates a `GET`-based HTML form, with `GET` data being transmitted and the resultant page being displayed by the browser. Section 21.2 (A Multisystem Search Engine Front End) gives an example. In the second approach, covered in Section 21.3, the applet sends `GET` data to a servlet and then processes the results itself. Section 21.4 (A Query Viewer That Uses Object Serialization and HTTP Tunneling) gives an example. In the third approach, covered in Section 21.5, the applet sends `POST` data to a servlet and then processes the results itself. Section 21.6 (An Applet That Sends POST Data) gives an example. Finally, Section 21.7 serves as a reminder that an applet can bypass the HTTP server altogether and talk directly to a custom server program running on the applet's home machine.

This chapter assumes that you already have some familiarity with basic applets (see Chapter 9) and focuses on the techniques to allow applets to communicate with server-side programs.

21.1 Sending Data with GET and Displaying the Resultant Page

The `showDocument` method instructs the browser to display a particular URL. You can transmit `GET` data to a servlet or CGI program by appending it to the program's URL after a question mark (?). Thus, to send `GET` data from an applet, you simply need to append the data to the string from which the URL is built, then create the `URL` object and call `showDocument` in the normal manner. A basic template for doing this in applets follows. Assume that `baseURL` is a string representing the URL of the server-side program and that `someData` is the information to be sent with the request.

```
try {
  URL programURL = new URL(baseURL + "?" + someData);
  getAppletContext().showDocument(programURL);
} catch(MalformedURLException mue) { ... }
```

When data is sent by a browser, it is *URL encoded,* which means that spaces are converted to plus signs (+) and nonalphanumeric characters are changed into a percent sign (%) followed by the two hex digits representing that character, as discussed in Section 18.2 (The FORM Element). The preceding example assumes that `someData` has already been encoded properly and fails if this assumption is wrong. JDK 1.1 and later have a `URLEncoder` class with a static `encode` method that can perform this encoding. So, if an applet is contacting a server-side program that normally receives `GET` data from HTML forms, the applet needs to encode the value of each entry, but not the equal sign (=) between each entry name and its value or the ampersand (&) between each name/value pair. Therefore, you cannot simply call `URLEncoder.encode (someData)` but instead need to selectively encode the value parts of each name/value pair. This encoding could be accomplished as follows:

```
String someData =
  name1 + "=" + URLEncoder.encode(val1) + "&" +
  name2 + "=" + URLEncoder.encode(val2) + "&" +
  ...
  nameN + "=" + URLEncoder.encode(valN);
try {
  URL programURL = new URL(baseURL + "?" + someData);
  getAppletContext().showDocument(programURL);
} catch(MalformedURLException mue) { ... }
```

The following section gives a full-fledged example.

21.2 A Multisystem Search Engine Front End

Listing 21.1 shows an applet that creates a textfield to gather user input. When the user submits the data, the applet URL-encodes the textfield value and generates three distinct URLs with embedded `GET` data: one each for the Google, Infoseek, and Lycos search engines. The applet then uses `showDocument` to instruct the browser to display the results of those URLs in three different frame cells. HTML forms cannot be used for this application since a form can submit its data to only a single URL. Listing 21.2 shows the `SearchSpec` class used by the applet to generate the specific URLs needed to redirect requests to various search engines. The `SearchSpec` class can also be used by servlets, as discussed in Chapter 19. The applet results are shown in Figure 21-1 and 21-2.

Figure 21-1. `SearchApplet` allows the user to enter a single search string for multiple search engines.

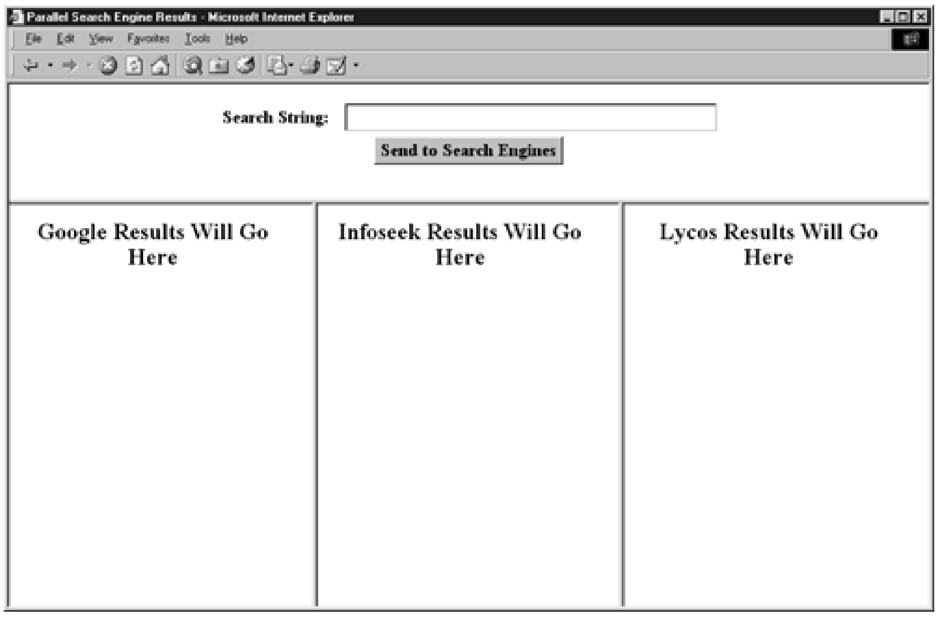
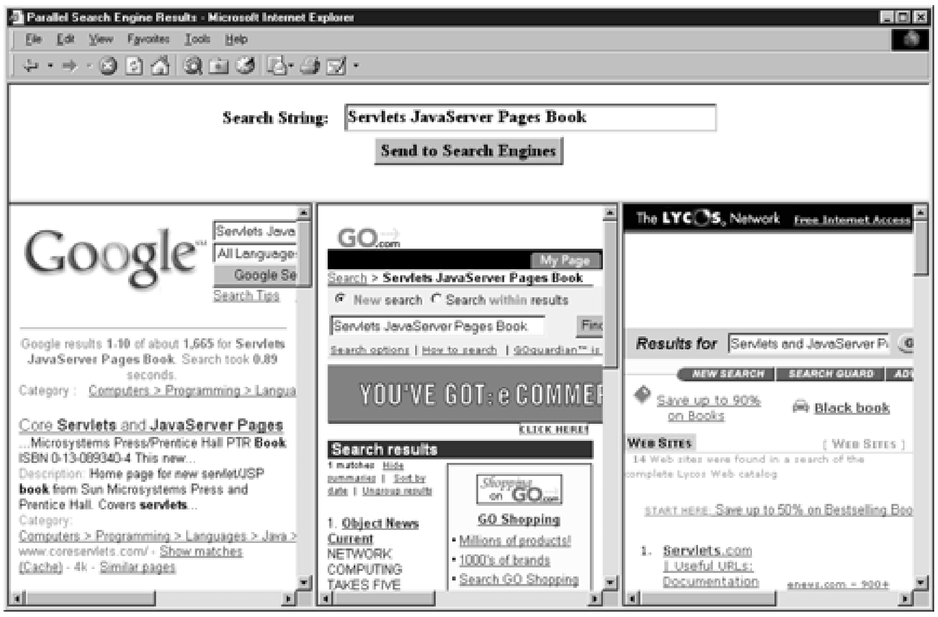

Parallel Search Engine Results - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Search String:
Send to Search Engines
Google Results Will Go Here
Infoseek Results Will Go Here
Lycos Results Will Go Here

Figure 21-2. Submitting the query yields side-by-side results from three search engines.



Google results 1-10 of about 1,665 for Servlets JavaServer Pages Book. Search took 0.89 seconds.

Category: Computers > Programming > Langua

Core Servlets and JavaServer Pages
...Microsystems Press/Prentice Hall PTR Book ISBN 0-13-089340-4 This new...
Description: Home page for new servlet/JSP book from Sun Microsystems Press and Prentice Hall. Covers servlets...
Category: Computers > Programming > Languages > Java >
www.coreservlets.com/ - Show matches
(Cache) - 4k - Similar pages

New search / Search within results

1. Object News Current NETWORK COMPUTING TAKES FIVE

Millions of products! · 1000's of brands · Search GO Shopping

Save up to 90% on Books · Black book

14 Web sites were found in a search of the complete Lycos Web catalog

START HERE: Save up to 50% on Bestselling Boo

1. Servlets.com | Useful URLs: Documentation

Listing 21.3 shows the top-level HTML document, and Listing 21.4 shows the HTML used for the frame cell actually containing the applet. If you are curious about the three tiny HTML files used for the initial contents of the bottom three frame cells shown in Figure 21-1, please refer to this book's Web site (http://www.corewebprogramming.com/).

Listing 21.1 `SearchApplet.java`

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
```

```
/** An applet that reads a value from a TextField,
 *  then uses it to build three distinct URLs with embedded
 *  GET data: one each for Google, Infoseek, and Lycos.
 *  The browser is directed to retrieve each of these
 *  URLs, displaying them in side-by-side frame cells.
 *  Note that standard HTML forms cannot automatically
 *  perform multiple submissions in this manner.
 */
public class SearchApplet extends Applet
                          implements ActionListener {
  private TextField queryField;
  private Button submitButton;

  public void init() {
    setBackground(Color.white);
    setFont(new Font("Serif", Font.BOLD, 18));
    add(new Label("Search String:"));
    queryField = new TextField(40);
    queryField.addActionListener(this);
    add(queryField);
    submitButton = new Button("Send to Search Engines");
    submitButton.addActionListener(this);
    add(submitButton);
  }

  /** Submit data when button is pressed <B>or</B>
   *  user presses Return in the TextField.
   */

  public void actionPerformed(ActionEvent event) {
    String query = URLEncoder.encode(queryField.getText());
    SearchSpec[] commonSpecs = SearchSpec.getCommonSpecs();
    // Omitting HotBot (last entry), as they use JavaScript to
    // pop result to top-level frame. Thus the length-1 below.
    for(int i=0; i<commonSpecs.length-1; i++) {
      try {
        SearchSpec spec = commonSpecs[i];
        // The SearchSpec class builds URLs of the
        // form needed by some common search engines.
        URL searchURL = new URL(spec.makeURL(query, "10"));
        String frameName = "results" + i;
        getAppletContext().showDocument(searchURL, frameName);
      } catch(MalformedURLException mue) {}
    }
  }
}
```

Listing 21.2 `SearchSpec.java`

```
/** Small class that encapsulates how to construct a
 *  search string for a particular search engine.
 */

public class SearchSpec {
  private String name, baseURL, numResultsSuffix;

  private static SearchSpec[] commonSpecs =
    { new SearchSpec("google",
                     "http://www.google.com/search?q=",
                     "&num="),
      new SearchSpec("infoseek",
                     "http://infoseek.go.com/Titles?qt=",
                     "&nh="),
      new SearchSpec("lycos",
                     "http://lycospro.lycos.com/cgi-bin/" +
                        "pursuit?query=",
                     "&maxhits="),
      new SearchSpec("hotbot",
                     "http://www.hotbot.com/?MT=",
                     "&DC=")
    };

  public SearchSpec(String name,
                    String baseURL,
                    String numResultsSuffix) {
    this.name = name;
    this.baseURL = baseURL;
    this.numResultsSuffix = numResultsSuffix;
  }

  public String makeURL(String searchString,
                        String numResults) {
    return(baseURL + searchString +
           numResultsSuffix + numResults);
  }

  public String getName() {
    return(name);
  }

  public static SearchSpec[] getCommonSpecs() {
    return(commonSpecs);
  }
}
```

Listing 21.3 `ParallelSearches.html`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Frameset//EN">
<HTML>
<HEAD>
  <TITLE>Parallel Search Engine Results</TITLE>
</HEAD>
```

```
<FRAMESET ROWS="120,*">
  <FRAME SRC="SearchAppletFrame.html" SCROLLING="NO">
  <FRAMESET COLS="*,*,*">
    <FRAME SRC="GoogleResultsFrame.html" id="results0">
    <FRAME SRC="InfoseekResultsFrame.html" id="results1">
    <FRAME SRC="LycosResultsFrame.html" id="results2">
  </FRAMESET>
</FRAMESET>
```

Listing 21.4 `SearchAppletFrame.html`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Search Applet Frame</TITLE>
</HEAD>

<BODY BGCOLOR="WHITE">
<CENTER>
<APPLET CODE="SearchApplet.class" WIDTH=600 HEIGHT=100>
  <B>This example requires a Java-enabled browser.</B>
</APPLET>
</CENTER>
</BODY>
</HTML>
```

21.3 Using GET and Processing the Results Directly (HTTP Tunneling)

In the previous example, an applet instructs the browser to display the output of a server-side program in a particular frame. Using the browser to display results is a reasonable approach when you are working with existing services, since most CGI programs are already set up to return HTML documents. However, if you are developing *both* the client and the server sides of the process, it seems a bit wasteful to always send back an entire HTML document. In some cases, it would be nice to simply return data to an applet that is already running. The applet could then present the data in a graph or some other custom display. This approach is sometimes known as *HTTP tunneling* since a custom communication protocol is embedded within the HTTP packets: proxies, encryption, server redirection, connections through firewalls, and all.

There are two main variations to this approach. Both make use of the `URLConnection` class to open an input stream from a URL. The difference lies in the type of stream they use. The first option is to use a `BufferedInputStream` or some other low-level stream that lets you read binary or ASCII data from an arbitrary server-side program. That approach is covered in the first subsection. The second option is to use an `ObjectInputStream` to directly read high-level data structures. That approach, covered in the second subsection, is available only when the server-side program is also written in the Java programming language.

Reading Binary or ASCII Data

An applet can read the content sent by the server by first creating a `URLConnection` derived from the URL of the server-side program and then attaching a `BufferedInputStream` to it. Seven main steps are required to implement this approach on the client, as described below. We are omitting the server-side code since the client code described here works with arbitrary server-side programs or static Web pages.

Note that many of the stream operations throw an `IOException`, so the following statements need to be enclosed in a `try`/`catch` block.

1. **Create a `URL` object referring to applet's home host.** You can pass an absolute URL string to the `URL` constructor (e.g., `"http://host/path"`), but since browser security restrictions prohibit connections from applets to machines other than the home server, it makes more sense to build a URL based upon the hostname from which the applet was loaded.

   ```
   URL currentPage = getCodeBase();
   String protocol = currentPage.getProtocol();
   String host = currentPage.getHost();
   int port = currentPage.getPort();
   String urlSuffix = "/servlet/SomeServlet";
   URL dataURL = new URL(protocol, host, port, urlSuffix);
   ```

2. **Create a `URLConnection` object.** The `openConnection` method of `URL` returns a `URLConnection` object. This object will be used to obtain streams with which to communicate.

   ```
   URLConnection connection = dataURL.openConnection();
   ```

3. **Instruct the browser not to cache the URL data.** The first thing you do with the `URLConnection` object is to specify that the browser not cache it. This approach guarantees that you get a fresh result each time.

   ```
   connection.setUseCaches(false);
   ```

4. **Set any desired HTTP headers.** If you want to set HTTP request headers (see Section 19.7), you can use `setRequestProperty` to do so.

   ```
   connection.setRequestProperty("header", "value");
   ```

5. **Create an input stream.** There are a variety of appropriate Create an input stream. There are a variety of appropriate streams; a common one is `BufferedReader`. The connection to the Web server is established when you create the input stream.

   ```
   BufferedReader in =
     new BufferedReader(new InputStreamReader(
                            connection.getInputStream()));
   ```

6. **Read each line of the document.** The HTTP specification stipulates that the server closes the connection when it is done. When the connection is closed, `readLine` returns `null`. So, simply read until you get `null`.

   ```
   String line;
   while ((line = in.readLine()) != null) {
     doSomethingWith(line);
   }
   ```

7. **Close the input stream.**

```
in.close();
```

Reading Serialized Data Structures

The approach shown in the previous subsection makes good sense when your applet is talking to an arbitrary server-side program or reading the content of static Web pages. However, when an applet talks to a servlet, you can do even better. Rather than sending binary or ASCII data, the servlet can transmit arbitrary data structures by using the Java serialization mechanism. The applet can read this data in a single step by using `readObject`; no long and tedious parsing is required. The steps required to implement HTTP tunneling are summarized below.

The Client Side

An applet needs to perform the following seven steps to read serialized data structures sent by a servlet. Only Steps 5 and 6 differ from what is required to read ASCII data. These steps are slightly simplified by the omission of the `try`/`catch` blocks.

1. **Create a `URL` object referring to the applet's home host.** As before, since the URL must refer to the host from which the applet was loaded, it makes the most sense to specify a URL suffix and construct the rest of the URL automatically.

   ```
   URL currentPage = getCodeBase();
   String protocol = currentPage.getProtocol();
   String host = currentPage.getHost();
   int port = currentPage.getPort();
   String urlSuffix = "/servlet/SomeServlet";
   URL dataURL = new URL(protocol, host, port, urlSuffix);
   ```

2. **Create a `URLConnection` object.** The `openConnection` method of `URL` returns a `URLConnection` object. This object will be used to obtain streams with which to communicate.

   ```
   URLConnection connection = dataURL.openConnection();
   ```

3. **Instruct the browser not to cache the URL data.** The first thing you do with the `URLConnection` object is to specify that the browser not cache it. This approach guarantees you get a fresh result each time.

   ```
   connection.setUseCaches(false);
   ```

4. **Set any desired HTTP headers.** If you want to set HTTP request headers (see Section 19.7), you can use `setRequestProperty` to do so.

   ```
   connection.setRequestProperty("header", "value");
   ```

5. **Create an `ObjectInputStream.`** The constructor for this class simply takes the raw input stream from the `URLConnection`. The connection to the Web server is established when you create the input stream.

   ```
   ObjectInputStream in =
     new ObjectInputStream(connection.getInputStream());
   ```

6. **Read the data structure with `readObject.`** The return type of `readObject` is `Object`, so you need to make a typecast to whatever more specific type the server actually

sent.

```
SomeClass value = (SomeClass)in.readObject();
doSomethingWith(value);
```

7. **Close the input stream.**

```
in.close();
```

The Server Side

A servlet needs to perform the following four steps to send serialized data structures to an applet. Assume that `request` and `response` are the `HttpServletRequest` and `HttpServletResponse` objects supplied to the `doGet` and `doPost` methods. Again, these steps are simplified slightly by the omission of the required `try`/`catch` blocks.

1. **Specify that binary content is being sent.** Designate `application/x-java-serialized-object` as the MIME type of the response. This is the standard MIME type for objects encoded with an ObjectOutputStream, although in practice, since the applet (not the browser) is reading the result, the MIME type is not very important. See the discussion of Content-Type in Section 19.10 (The Server Response: HTTP Response Headers) for more information on MIME types.

```
String contentType =
  "application/x-java-serialized-object";
response.setContentType(contentType);
```

2. **Create an `ObjectOutputStream`.**

```
ObjectOutputStream out =
  new ObjectOutputStream(response.getOutputStream());
```

3. **Write the data structure by using `writeObject`.** Most built-in data structures can be sent with `writeObject`. Classes *you* write, however, must implement the `Serializable` interface. This is a simple requirement, however, since `Serializable` defines no methods. Simply declare that your class implements it.

```
SomeClass value = new SomeClass(...);
out.writeObject(value);
```

4. **Flush the stream to be sure all content has been sent to the client.**

```
out.flush();
```

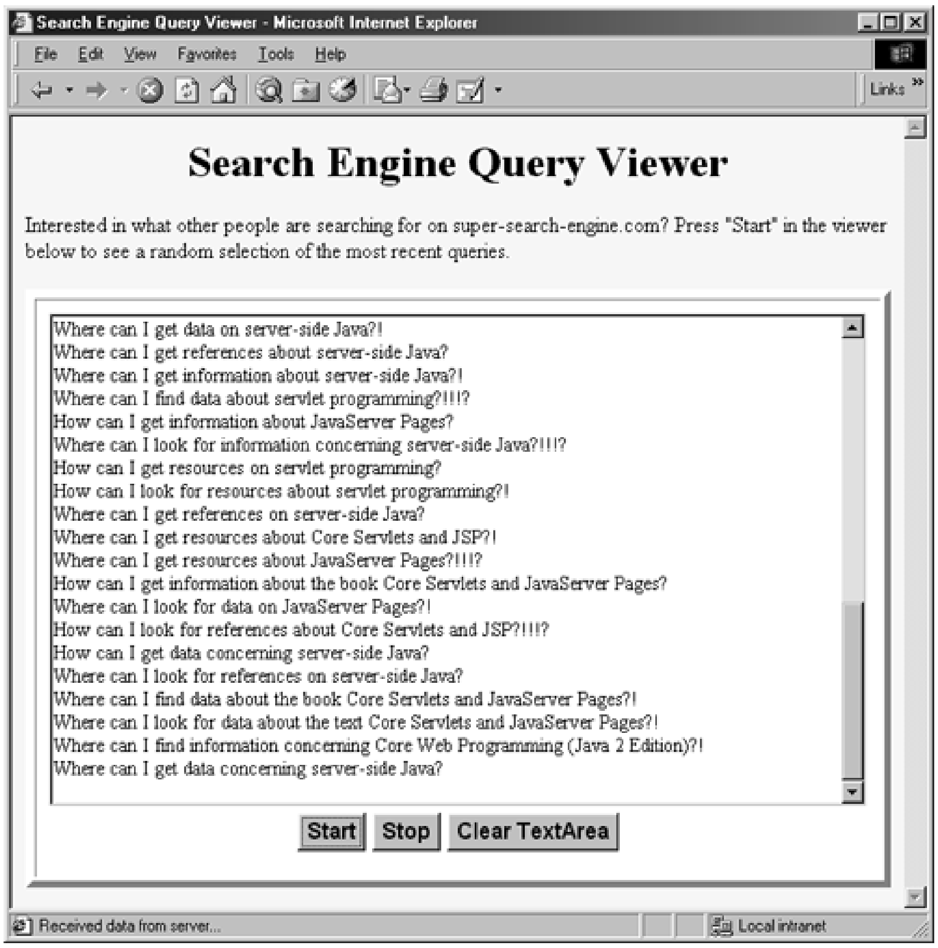
The following section gives an example of this HTTP tunneling approach.

21.4 A Query Viewer That Uses Object Serialization and HTTP Tunneling

Many people are curious about what types of queries are sent to the major search engines. This is partly idle curiosity ("Is it really true that 64 percent of the queries at AltaVista are from employers looking for programmers that know Java technology?") and partly so that HTML authors can arrange their page content to fit the types of queries normally submitted, hoping to improve their site's ranking with the search engines.

This section presents an applet/servlet combination that displays the fictitious `super-search-engine.com` "live," continually updating sample queries to visitors that load their query viewer page. Listing 21.5 shows the main applet, which makes use of an auxiliary class (Listing 21.6) to retrieve the queries in a background thread. Once the user initiates the process, the applet places a sample query in a scrolling text are every half-second, as shown in Figure 21-3. Finally, Listing 21.7 shows the servlet that generates the queries on the server. It generates a random sampling of actual recent user queries and sends 50 of them to the client for each request. Servlet details are explained in Chapter 19.

Figure 21-3. The `ShowQueries` applet in action.


Search Engine Query Viewer - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Received data from server...
Local intranet

Search Engine Query Viewer

Interested in what other people are searching for on super-search-engine.com? Press "Start" in the viewer below to see a random selection of the most recent queries.

```
Where can I get data on server-side Java?!
Where can I get references about server-side Java?
Where can I get information about server-side Java?!
Where can I find data about servlet programming?!!!?
How can I get information about JavaServer Pages?
Where can I look for information concerning server-side Java?!!!?
How can I get resources on servlet programming?
How can I look for resources about servlet programming?!
Where can I get references on server-side Java?
Where can I get resources about Core Servlets and JSP?!
Where can I get resources about JavaServer Pages?!!!?
How can I get information about the book Core Servlets and JavaServer Pages?
Where can I look for data on JavaServer Pages?!
How can I look for references about Core Servlets and JSP?!!!?
How can I get data concerning server-side Java?
Where can I look for references on server-side Java?
Where can I find data about the book Core Servlets and JavaServer Pages?!
Where can I look for data about the text Core Servlets and JavaServer Pages?!
Where can I find information concerning Core Web Programming (Java 2 Edition)?!
Where can I get data concerning server-side Java?
```

Start Stop Clear TextArea

If you download the applet and servlet source code from http://www.corewebprogramming.com/ and try this application yourself, be aware that it will only work when you load the top-level HTML page by using HTTP (i.e., by using a URL of the form `http://...` to request the page from a Web server). Loading the applet directly off your disk fails—the applet connects back to its home site to contact the servlet. Besides, `URLConnection` fails for non-HTTP applets in general.

Listing 21.5 `ShowQueries.java`

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.net.*;

/** Applet reads arrays of strings packaged inside
 *  a QueryCollection and places them in a scrolling
 *  TextArea. The QueryCollection obtains the strings
 *  by means of a serialized object input stream.
 */

public class ShowQueries extends Applet
                         implements ActionListener, Runnable {
  private TextArea queryArea;
  private Button startButton, stopButton, clearButton;
  private QueryCollection currentQueries;
  private QueryCollection nextQueries;
  private boolean isRunning = false;
  private String address =
    "/servlet/cwp.QueryGenerator";
  private URL currentPage;

  public void init() {
    setBackground(Color.white);
    setLayout(new BorderLayout());
    queryArea = new TextArea();
    queryArea.setFont(new Font("Serif", Font.PLAIN, 14));
    add(queryArea, BorderLayout.CENTER);
    Panel buttonPanel = new Panel();
    Font buttonFont = new Font("SansSerif", Font.BOLD, 16);
    startButton = new Button("Start");
    startButton.setFont(buttonFont);
    startButton.addActionListener(this);
    buttonPanel.add(startButton);
    stopButton = new Button("Stop");
    stopButton.setFont(buttonFont);
    stopButton.addActionListener(this);
    buttonPanel.add(stopButton);
    clearButton = new Button("Clear TextArea");
    clearButton.setFont(buttonFont);
    clearButton.addActionListener(this);
    buttonPanel.add(clearButton);
    add(buttonPanel, BorderLayout.SOUTH);
    currentPage = getCodeBase();
    // Request a set of sample queries. They
    // are loaded in a background thread, and
    // the applet checks to see if they have finished
    // loading before trying to extract the strings.
    currentQueries = new QueryCollection(address, currentPage);
    nextQueries = new QueryCollection(address, currentPage);
  }

  /** If you press the "Start" button, the system
   *  starts a background thread that displays
   *  the queries in the TextArea. Pressing "Stop"
   *  halts the process, and "Clear" empties the
   *  TextArea.
   */
  public void actionPerformed(ActionEvent event) {
    if (event.getSource() == startButton) {
      if (!isRunning) {
        Thread queryDisplayer = new Thread(this);
        isRunning = true;
        queryArea.setText("");
        queryDisplayer.start();
        showStatus("Started display thread...");
      } else {
        showStatus("Display thread already running...");
      }
    } else if (event.getSource() == stopButton) {
      isRunning = false;
      showStatus("Stopped display thread...");
    } else if (event.getSource() == clearButton) {
      queryArea.setText("");
    }
  }
  /** The background thread takes the currentQueries
   *  object and every half-second places one of the queries
   *  the object holds into the bottom of the TextArea. When
   *  all of the queries have been shown, the thread copies
   *  the value of the nextQueries object into
   *  currentQueries, sends a new request to the server
   *  in order to repopulate nextQueries, and repeats
   *  the process.
   */

  public void run() {
    while(isRunning) {
      showQueries(currentQueries);
      currentQueries = nextQueries;
      nextQueries = new QueryCollection(address, currentPage);
    }
  }

  private void showQueries(QueryCollection queryEntry) {
    // If a request has been sent to server but the result
    // isn't back yet, poll every second. This should
    // happen rarely but is possible with a slow network
    // connection or an overloaded server.
    while(!queryEntry.isDone()) {
      showStatus("Waiting for data from server...");
      pause(1);
    }
    showStatus("Received data from server...");
    String[] queries = queryEntry.getQueries();
    String linefeed = "\n";
    // Put a string into TextArea every half-second.
    for(int i=0; i<queries.length; i++) {
      if (!isRunning) {
        return;
      }
      queryArea.append(queries[i]);
      queryArea.append(linefeed);
      pause(0.5);
    }
  }

  public void pause(double seconds) {
    try {
      Thread.sleep((long)(seconds*1000));
    } catch(InterruptedException ie) {}
  }
}
```

Listing 21.6 `QueryCollection.java`

```
import java.net.*;
import java.io.*;

/** When this class is built, it returns a value
 *  immediately, but this value returns false for isDone
 *  and null for getQueries. Meanwhile, it starts a Thread
 *  to request an array of query strings from the server,
 *  reading them in one fell swoop by means of an
 *  ObjectInputStream. Once they've all arrived, they
 *  are placed in the location getQueries returns,
 *  and the isDone flag is switched to true.
 *  Used by the ShowQueries applet.
 */

public class QueryCollection implements Runnable {
  private String[] queries;
  private String[] tempQueries;
  private boolean isDone = false;
  private URL dataURL;

  public QueryCollection(String urlSuffix, URL currentPage) {
    try {
      // Only the URL suffix need be supplied, since
      // the rest of the URL is derived from the current page.
      String protocol = currentPage.getProtocol();
      String host = currentPage.getHost();
      int port = currentPage.getPort();
      dataURL = new URL(protocol, host, port, urlSuffix);
      Thread queryRetriever = new Thread(this);
      queryRetriever.start();
    } catch(MalformedURLException mfe) {
      isDone = true;
    }
  }

  public void run() {
    try {
      tempQueries = retrieveQueries();
      queries = tempQueries;
    } catch(IOException ioe) {
      tempQueries = null;
      queries = null;
    }
    isDone = true;
  }
  public String[] getQueries() {
    return(queries);
  }

  public boolean isDone() {
    return(isDone);
  }

  private String[] retrieveQueries() throws IOException {
    URLConnection connection = dataURL.openConnection();
    // Make sure browser doesn't cache this URL, since
    // I want different queries for each request.
    connection.setUseCaches(false);
    // Use ObjectInputStream so I can read a String[]
    // all at once.
    ObjectInputStream in =
      new ObjectInputStream(connection.getInputStream());
    try {
      // The return type of readObject is Object, so
      // I need a typecast to the actual type.
      String[] queryStrings = (String[])in.readObject();
      return(queryStrings);
    } catch(ClassNotFoundException cnfe) {
      return(null);
    }
  }
}
```

Listing 21.7 `QueryGenerator.java`

```
package cwp;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Servlet that generates an array of strings and
 *  sends them via an ObjectOutputStream to applet
 *  or other Java client.
 */
public class QueryGenerator extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    boolean useNumbering = true;
    String useNumberingFlag =
      request.getParameter("useNumbering");
    if ((useNumberingFlag == null) ||
        useNumberingFlag.equals("false")) {
      useNumbering = false;
    }
    String contentType =
      "application/x-java-serialized-object";
    response.setContentType(contentType);
    ObjectOutputStream out =
      new ObjectOutputStream(response.getOutputStream());
    String[] queries = getQueries(useNumbering);
    // If you send a nonstandard data structure, be
    // sure it is defined with "implements Serializable".
    out.writeObject(queries);
    out.flush();
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }

  private String[] getQueries(boolean useNumbering) {
    String[] queries = new String[50];
    for(int i=0; i<queries.length; i++) {
      queries[i] = randomQuery();
      if (useNumbering) {
        queries[i] = "" + (i+1) + ": " + queries[i];
      }
    }
    return(queries);
  }

  // The real, honest-to-goodness queries people have sent :-)
  private String randomQuery() {
    String[] locations = { "Where ", "How " };
    String[] actions =
      { "can I look for ", "can I find ", "can I get " };
    String[] sources =
      { "information ", "resources ", "data ", "references " };
    String[] prepositions = { "on ", "about ", "concerning " };
    String[] subjects =
      { "the book Core Servlets and JavaServer Pages",
        "the text Core Servlets and JavaServer Pages",
        "Core Servlets and JavaServer Pages",
        "Core Servlets and JSP",
        "the book Core Web Programming (Java 2 Edition)",
        "Core Web Programming (Java 2 Edition)",
        "servlet programming", "JavaServer Pages", "JSP",
        "Java alternatives to CGI", "server-side Java" };
    String[] endings = { "?", "?", "?", "?!", "?!!!?" };
    String[][] sentenceTemplates =
      { locations, actions, sources,
        prepositions, subjects, endings };
    String query = "";
    for(int i=0; i<sentenceTemplates.length; i++) {
      query = query + randomEntry(sentenceTemplates[i]);
    }
    return(query);
  }

  private String randomEntry(String[] strings) {
    int index = (int)(Math.random()*strings.length);
    return(strings[index]);
  }
}
```

21.5 Using POST and Processing the Results Directly (HTTP Tunneling)

With GET data, an applet has two options for the results of a submission: tell the browser to display the results (construct a URL object and call getAppletContext().showDocument) or process the results itself (construct a URL object, get a URLConnection, open an input stream, and read the results). These two options are discussed in Sections 21.1 and 21.3, respectively. With POST data, however, only the second option is available since the URL constructor has no method to let you associate POST data with it. Sending POST data has some of the same advantages and disadvantages as when applets send GET data. The two main disadvantages are that the server-side program must be on the host from which the applet was loaded, and that the applet is required to display all the results itself: it cannot pass HTML to the browser in a portable manner. On the plus side, the server-side program can be simpler (not needing to wrap the results in HTML) and the applet can update its display without requiring the page to be reloaded. Furthermore, applets that communicate by using POST can use serialized data streams to send data *to* a servlet, in addition to reading serialized data *from* a servlet. This is quite an advantage, since serialized data simplifies communication and HTTP tunneling lets you piggyback on existing connections through firewalls even when direct socket connections are prohibited. Applets using GET can read serialized data (see Section 21.4) but are unable to send it since it is not legal to append arbitrary binary data to URLs.

Thirteen steps are required for the applet to send POST data to the server and read the results, as shown below. Although there are many required steps, each step is relatively simple. The code is slightly simplified by the omission of try/catch blocks around the statements.

1. **Create a URL object referring to the applet's home host.** As before, since the URL must refer to the host the applet came from, it makes the most sense to specify a URL suffix and construct the rest of the URL automatically.

```
URL currentPage = getCodeBase();
String protocol = currentPage.getProtocol();
String host = currentPage.getHost();
int port = currentPage.getPort();
String urlSuffix = "/servlet/SomeServlet";
URL dataURL =
  new URL(protocol, host, port, urlSuffix);
```

2. **Create a `URLConnection` object.** This object will be used to obtain input and output streams that connect to the server.

```
URLConnection connection = dataURL.openConnection();
```

3. **Instruct the browser not to cache the results.**

```
connection.setUseCaches(false);
```

4. **Tell the system to permit you to send data, not just read it.**

```
connection.setDoOutput(true);
```

5. **Create a `ByteArrayOutputStream` to buffer the data that will be sent to the server.** The purpose of the `ByteArrayOutputStream` here is to determine the size of the output so that the applet can set the `Content-Length` header, a required part of `POST` requests. The `ByteArrayOutputStream` constructor specifies an initial buffer size, but this value is not critical since the buffer will grow automatically if necessary.

```
ByteArrayOutputStream byteStream =
  new ByteArrayOutputStream(512);
```

6. **Attach an output stream to the `ByteArrayOutputStream`.** Use a `PrintWriter` to send normal form data. To send serialized data structures, use an `ObjectOutputStream` instead.

```
PrintWriter out = new PrintWriter(byteStream, true);
```

7. **Put the data into the buffer.** For form data, use `print`. For high-level serialized objects, use `writeObject`.

```
String val1 = URLEncoder.encode(someVal1);
String val2 = URLEncoder.encode(someVal2);
String data = "param1=" + val1 +
              "&param2=" + val2; // Note '&'
out.print(data);  // Note print, not println
out.flush(); // Necessary since no println used
```

8. **Set the `Content-Length` header.** This header is required for `POST` data, even though it is unused with `GET` requests.

```
connection.setRequestProperty
  ("Content-Length", String.valueOf(byteStream.size()));
```

9. **Set the `Content-Type` header.** Netscape uses `multipart/form-data` by default,

but regular form data requires a setting of `application/x-www-form-urlencoded`, which is the default with Internet Explorer. So, for portability you should set this value explicitly when sending regular form data. The value is irrelevant when you are sending serialized data.

```
connection.setRequestProperty
  ("Content-Type", "application/x-www-form-urlencoded");
```

10. **Send the real data.**

```
byteStream.writeTo(connection.getOutputStream());
```

11. **Open an input stream.** You typically use a `BufferedReader` for ASCII or binary data and an `ObjectInputStream` for serialized Java objects.

```
BufferedReader in =
  new BufferedReader(new InputStreamReader
                       (connection.getInputStream()));
```

12. **Read the result.** The specific details depend on what type of data the server sends. Here is an example that does something with each line sent by the server:

```
String line;
while((line = in.readLine()) != null) {
  doSomethingWith(line);
}
```

13. **Pat yourself on the back.** Yes, the procedure for handling `POST` is long and tedious. Fortunately, it is a relatively rote process. Besides, you can always download an example from www.corewebprogramming.com and use it as a starting point.

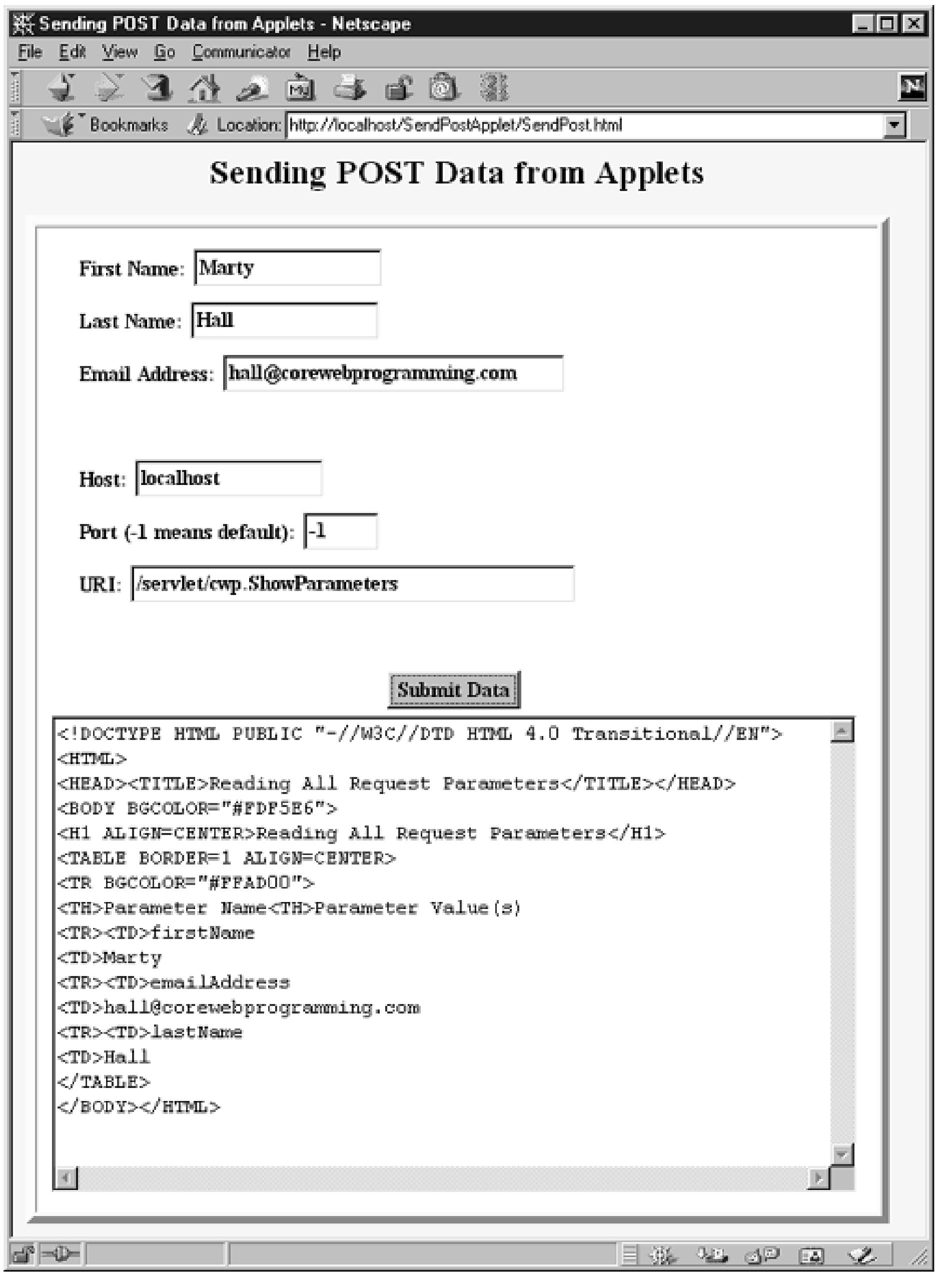
The next section gives an example of an applet that performs these steps.

21.6 An Applet That Sends POST Data

Listing 21.8 presents an applet that follows the approach outlined in the previous section. The applet uses a `URLConnection` and an attached `ByteArrayOutputStream` to send `POST` data to a URL the user specifies. The applet also makes use of the `LabeledTextField` class, available fordownload from http://www.corewebprogramming.com/.

Figure 21-4 shows the results of submitting the data to the `ShowParameters` servlet, a small servlet that builds a Web page illustrating all of the query parameters sent to it; see Section 19.6 (The Client Request: Form Data).

Figure 21-4. Result of using `SendPost` to send `POST` data to the `ShowParameters` servlet.


Sending POST Data from Applets - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost/SendPostApplet/SendPost.html

Sending POST Data from Applets

First Name: Marty

Last Name: Hall

Email Address: hall@corewebprogramming.com

Host: localhost

Port (-1 means default): -1

URI: /servlet/cwp.ShowParameters

Submit Data

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Reading All Request Parameters</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN=CENTER>Reading All Request Parameters</H1>
<TABLE BORDER=1 ALIGN=CENTER>
<TR BGCOLOR="#FFAD00">
<TH>Parameter Name<TH>Parameter Value(s)
<TR><TD>firstName
<TD>Marty
<TR><TD>emailAddress
<TD>hall@corewebprogramming.com
<TR><TD>lastName
<TD>Hall
</TABLE>
</BODY></HTML>
```

Listing 21.8 SendPost.java

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;

/** Applet that reads firstName, lastName, and
 *  emailAddress parameters and sends them via
 *  POST to the host, port, and URI specified.
 */

public class SendPost extends Applet
                      implements ActionListener {
  private LabeledTextField firstNameField, lastNameField,
                         emailAddressField, hostField,
                         portField, uriField;
  private Button sendButton;
  private TextArea resultsArea;
  URL currentPage;
public void init() {
  setBackground(Color.white);
  setLayout(new BorderLayout());
  Panel inputPanel = new Panel();
  inputPanel.setLayout(new GridLayout(9, 1));
  inputPanel.setFont(new Font("Serif", Font.BOLD, 14));
  firstNameField =
    new LabeledTextField("First Name:", 15);
  inputPanel.add(firstNameField);
  lastNameField =
    new LabeledTextField("Last Name:", 15);
  inputPanel.add(lastNameField);
  emailAddressField =
    new LabeledTextField("Email Address:", 30);
  inputPanel.add(emailAddressField);
  Canvas separator1 = new Canvas();
  inputPanel.add(separator1);
  hostField =
    new LabeledTextField("Host:", 15);

  // Applets loaded over the network can only connect
  // to the server from which they were loaded.
  hostField.getTextField().setEditable(false);

  currentPage = getCodeBase();
  // getHost returns empty string for applets from local disk.
  String host = currentPage.getHost();
  String resultsMessage = "Results will be shown here...";
  if (host.length() == 0) {
    resultsMessage = "Error: you must load this applet\n" +
                     "from a real Web server via HTTP,\n" +
                     "not from the local disk using\n" +
                     "a 'file:' URL. It is fine,\n" +
                     "however, if the Web server is\n" +
                     "running on your local system.";
    setEnabled(false);
  }
  hostField.getTextField().setText(host);
  inputPanel.add(hostField);
  portField =
    new LabeledTextField("Port (-1 means default):", 4);
  String portString = String.valueOf(currentPage.getPort());
  portField.getTextField().setText(portString);
  inputPanel.add(portField);
     uriField =
      new LabeledTextField("URI:", 40);
    String defaultURI = "/servlet/cwp.ShowParameters";
    uriField.getTextField().setText(defaultURI);
    inputPanel.add(uriField);
    Canvas separator2 = new Canvas();
    inputPanel.add(separator2);
    sendButton = new Button("Submit Data");
    sendButton.addActionListener(this);
    Panel buttonPanel = new Panel();
    buttonPanel.add(sendButton);
    inputPanel.add(buttonPanel);
    add(inputPanel, BorderLayout.NORTH);
    resultsArea = new TextArea();
    resultsArea.setFont(new Font("Monospaced", Font.PLAIN, 14));
    resultsArea.setText(resultsMessage);
    add(resultsArea, BorderLayout.CENTER);
  }

  public void actionPerformed(ActionEvent event) {
    try {
      String protocol = currentPage.getProtocol();
      String host = hostField.getTextField().getText();
      String portString = portField.getTextField().getText();
      int port;
      try {
        port = Integer.parseInt(portString);
      } catch(NumberFormatException nfe) {
        port = -1; // I.e., default port of 80
      }
      String uri = uriField.getTextField().getText();
      URL dataURL = new URL(protocol, host, port, uri);
      URLConnection connection = dataURL.openConnection();

      // Make sure browser doesn't cache this URL.
      connection.setUseCaches(false);

      // Tell browser to allow me to send data to server.
      connection.setDoOutput(true);

      ByteArrayOutputStream byteStream =
        new ByteArrayOutputStream(512); // Grows if necessary
      // Stream that writes into buffer
      PrintWriter out = new PrintWriter(byteStream, true);
      String postData =
        "firstid=" + encodedValue(firstNameField) +
        "&lastid=" + encodedValue(lastNameField) +
        "&emailAddress=" + encodedValue(emailAddressField);

      // Write POST data into local buffer
      out.print(postData);
      out.flush(); // Flush since above used print, not println
      // POST requests are required to have Content-Length
      String lengthString =
        String.valueOf(byteStream.size());
      connection.setRequestProperty
        ("Content-Length", lengthString);

      // Netscape sets the Content-Type to multipart/form-data
      // by default. So, if you want to send regular form data,
      // you need to set it to
      // application/x-www-form-urlencoded, which is the
      // default for Internet Explorer. If you send
      // serialized POST data with an ObjectOutputStream,
      // the Content-Type is irrelevant, so you could
      // omit this step.
      connection.setRequestProperty
        ("Content-Type", "application/x-www-form-urlencoded");

      // Write POST data to real output stream
      byteStream.writeTo(connection.getOutputStream());

      BufferedReader in =
        new BufferedReader(new InputStreamReader
                             (connection.getInputStream()));
      String line;
      String linefeed = "\n";
      resultsArea.setText("");
      while((line = in.readLine()) != null) {
        resultsArea.append(line);
        resultsArea.append(linefeed);
      }
    } catch(IOException ioe) {
      // Print debug info in Java Console
      System.out.println("IOException: " + ioe);
    }
  }
  // LabeledTextField is really a Panel with a Label and
  // TextField inside it. This extracts the TextField part,
  // gets the text inside it, URL-encodes it, and
  // returns the result.

  private String encodedValue(LabeledTextField field) {
    String rawValue = field.getTextField().getText();
    return(URLEncoder.encode(rawValue));
  }
}
```

21.7 Bypassing the HTTP Server

Although applets can only open network connections to the same machine they were loaded from, they need not necessarily connect on the same *port* (e.g., 80, the HTTP port). So, applets are permitted to use raw sockets, JDBC, or RMI to communicate with custom clients running on the server host.

Applets do these operations in exactly the same manner as do normal Java programs, so you can use whatever approaches to socket, JDBC, and RMI programming you are already familiar with, provided that the network server is on the same host as the Web server that delivered the applet.

21.8 Summary

HTML forms provide the simplest and most common front end to server-side programs. Applets, however, provide richer user interfaces, support continuously updating displays, and simplify the transmission of large complex data structures. The general rule of thumb is to use HTML forms whenever possible. But it is nice to be able to fall back to applets when HTML forms are too limiting.

CONTENTS