SYMANTEC.™

Visual Cafe™ Sourcebook

Symantec Visual Cafe™ Sourcebook

Copyright Notice

Released:11/98 for Visual Cafe 3.0

Trademarks

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

SYMANTEC LICENSE AND WARRANTY

The software which accompanies this license (the "Software") is the property of Symantec or its licensors and is protected by copyright law. While Symantec continues to own the Software, you will have certain rights to use the Software after your acceptance of this license. Except as may be modified by a license addendum which accompanies this license, your rights and obligations with respect to the use of this Software are as follows:

• You may:

(i) use one copy of the Software on a single computer;

(ii) make one copy of the Software for archival purposes, or copy the software onto the hard disk of your computer and retain the original for archival purposes;

(iii) use the Software on a network, provided that you have a licensed copy of the Software for each computer that can access the Software over that network;

(iv) after written notice to Symantec, transfer the Software on a permanent basis to another person or entity, provided that you retain no copies of the Software and the transferee agrees to the terms of this agreement; and

(v) if a single person uses the computer on which the Software is installed at least 80% of the time, then after returning the completed product registration card which accompanies the Software, that person may also use the Software on a single home computer.

• You may not:

(i) copy the documentation which accompanies the Software;

(ii) sublicense, rent or lease any portion of the Software;

(iii) reverse engineer, decompile, disassemble, modify, translate, make any attempt to discover the source code of the Software, or create derivative works from the Software; or

(iv) use a previous version or copy of the Software after you have received a disk replacement set or an upgraded version as a replacement of the prior version, unless you donate a previous version of an upgraded version to a charity of your choice, and such charity agrees in writing that it will be the sole end user of the product, and that it will abide by the terms of this agreement. Unless you so donate a previous version of an upgraded version, upon upgrading the Software, all copies of the prior version must be destroyed.

• Sixty Day Money Back Guarantee:

If you are the original licensee of this copy of the Software and are dissatisfied with it for any reason, you may return the complete product, together with your receipt, to Symantec or an authorized dealer, postage prepaid, for a full refund at any time during the sixty day period following the delivery to you of the Software.

• Limited Warranty:

Symantec warrants that the media on which the Software is distributed will be free from defects for a period of sixty (60) days from the date of delivery of the Software to you. Your sole remedy in the event of a breach of this warranty will be that Symantec will, at its option, replace any defective media returned to Symantec within the warranty period or refund the money you paid for the Software. Symantec does not warrant that the Software will meet your requirements or that operation of the Software will be uninterrupted or that the Software will be error-free.

THE ABOVE WARRANTY IS EXCLUSIVE AND IN LIEU OF ALL OTHER WARRANTIES, WHETHER EXPRESS OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHER RIGHTS, WHICH VARY FROM STATE TO STATE.

• Disclaimer of Damages:

REGARDLESS OF WHETHER ANY REMEDY SET FORTH HEREIN FAILS OF ITS ESSENTIAL PURPOSE, IN NO EVENT WILL SYMANTEC BE LIABLE TO YOU FOR ANY SPECIAL, CONSEQUENTIAL, INDIRECT OR SIMILAR DAMAGES, INCLUDING ANY LOST PROFITS OR LOST DATA ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE EVEN IF SYMANTEC HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

IN NO CASE SHALL SYMANTEC'S LIABILITY EXCEED THE PURCHASE PRICE FOR THE SOFTWARE. The disclaimers and limitations set forth above will apply regardless of whether you accept the Software.

• U.S. Government Restricted Rights:

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c) (1) and (2) of the Commercial Computer Software-Restricted Rights clause at 48 CFR 52.227-19, as applicable, Symantec Corporation, 10201 Torre Avenue, Cupertino, CA 95014.

Language Addendum

If the Software is a Symantec language product, then you have a royalty-free right to include object code derived from the Symantec component (java source or class) files in programs that you develop using the Software and you also have the right to use, distribute, and license such programs to third parties without payment of any further license fees, so long as a copyright notice sufficient to protect your copyright in the program is included in the graphic display of your program and on the labels affixed to the media on which your program is distributed. You have the right to make changes to the Symantec components, but only to the extent necessary to correct bugs in such components, and not for any other purpose. You also have a royalty-free right to include unmodified (except as stated in the previous sentence) Symantec component files required by your programs, but not as components of any development environment or component library you are distributing. The Symantec component files that may be redistributed are in the following folder in the Visual Cafe directory - VisualCafe\redist. The Java Virtual Machine (VM) or Just In Time (JIT) compiler may not be redistributed.

C O N T E N T S

CHAPTER 1

Introduction

This book has example applications with in-depth explanations so you can use them, learn from them, or adapt code from them for your own programming purposes.

Purpose

We want to help you program effectively. Some people learn better by working with examples. Our examples are tailored to work with our product.

Conventions

The small pieces of code that are explained are referred to as *snippets*.

All code is in `Courier.`

How to Use This Book

In addition to the code snippets printed throughout this book, you may also view the complete source code for each example inside Visual Cafe. You can run any of the examples from `VisualCafe\Sourcebook\`.

Additional Information

Before you use this book, you should be familiar with how to use Visual Cafe. You can learn about Visual Cafe by using the tour or the *Getting Started Guide*. If you need to learn how to do a certain task, we will refer you to the *User's Guide* or online help for that information. For general Java information, see one of the many Java books available, the Visual Cafe online help files, or the web-based documentation you can find at www.java.sun.com.

What you should know

You should be familiar with programming in general and Java in particular. In addition, you should be familiar with the general use of Visual Cafe.

The programs in this book

This book contains two sample programs.

- Weblog is a program with a graphical user interface that reads Web usage log files, analyzes them, and produces charts from the information.
- ESDServlet is a simulated electronic software distribution servlet. A servlet is the program that runs in connection with a web server and extends the capabilities of the server in the same way an applet extends the capabilities of a browser.

CHAPTER 2

Web Log Analysis Tool

This chapter discusses a Java application called Weblog which processes website traffic information and presents it in a report. The code created for this example is available in:

```
VisualCafe\Sourcebook\Weblog
```

You can run Weblog, view the source code in Visual Cafe, and copy classes and methods into your own applications or applets. This chapter discusses parts of the code in detail to illustrate how you can implement typical application tasks using Visual Cafe.

Overview of the Weblog application

Weblog is a complete application that has a user interface, displays a splash screen while initializing, allows the user to control the application with buttons and menus, collects information with modal dialogs, and reads and writes files.

Your company, organization, or you as an individual may have set up a Web server or may have plans to do so in the future. When you set up a server, you can generate an access log file that tracks user traffic on your website. Reading a raw log file can be difficult, so many website administrators purchase a Web log analysis tool that can arrange the raw data into more useful statistics. Weblog is such a tool.

Weblog allows you to answer the following questions about the traffic on your Web server:

- When do I have the most and least activity on my site?

- How many times was my home page visited?

Because Weblog is an application rather than an applet, it does not require a Web browser to run. (Java security rules keep applets from accessing local files, so an applet couldn't read local log files or save reports to the local disk.)

Weblog uses standard Java components and Visual Cafe components to create the user interface (UI). Much of the program was created without manual coding by using Visual Cafe's Interaction Wizard, and Form Designer.

Weblog illustrates:

- Saving and loading program data (serializing)
- Using threads
- Giving the user progress-tracking feedback
- Creating a wizard
- Validating input with the Wizard component
- Parsing and validating date entries
- Accessing files using `URL` and `URLConnection`
- Adding a splash screen

Weblog can analyze any Common Log File or Combined Log File from a Web server such as Netscape, NCSA, O'Reilly WebSite, Apache, Quarterdeck, Oracle, and other Windows, Unix, and Mac OS Web servers.

Using Weblog

This section shows you the user interface of Weblog, and shows you how to:

- Start Weblog
- Define a report
- Change a report definition
- Analyze data
- View a report

Weblog comes with some sample log files that you can use to see what the program does; if you want to analyze your own log files, you need to set

up your server to generate log files. See your server administrator documentation for how to do that.

Starting Weblog

You can open Weblog as a project in Visual Cafe and run it there or you can run it from a DOS command line.

To start Weblog in Visual Cafe:

1. Choose Open Project from the File menu.
2. Find `weblog.vep` in `VisualCafe\Sourcebook\Samples\Weblog` and open it.
3. Choose Execute from the Project menu.

To start Weblog in a DOS window:

Since Weblog is written in Java, and is an application and not an applet, you must start Weblog by using the Java virtual machine (JVM).

1. Make certain that the directory where `javaw.exe` is included in the value of your `PATH` environment variable.
2. Open a DOS command window.
3. Change to the directory:
   `VisualCafe\Sourcebook\Samples\Weblog\class`
4. Enter:

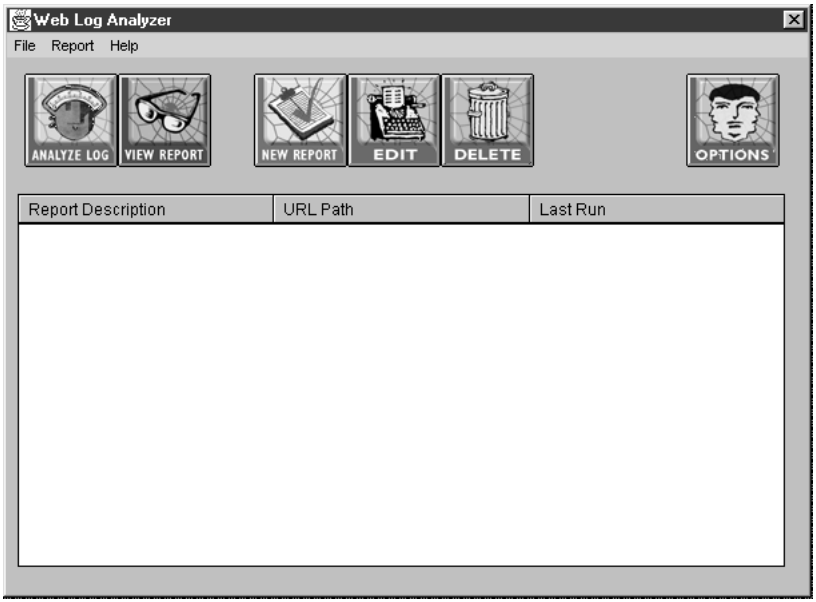
   ```
   java WebLogAnalyzer
   ```

   If you receive an error, check to make sure your `java.exe` file is in your path. You can also try typing the following:

   ```
   c:\VisualCafe\java\bin\java WebLogAnalyzer
   ```

   where *c:\VisualCafe\java\bin* is the location of `javaw.exe`.

When Weblog starts, you see its main window:



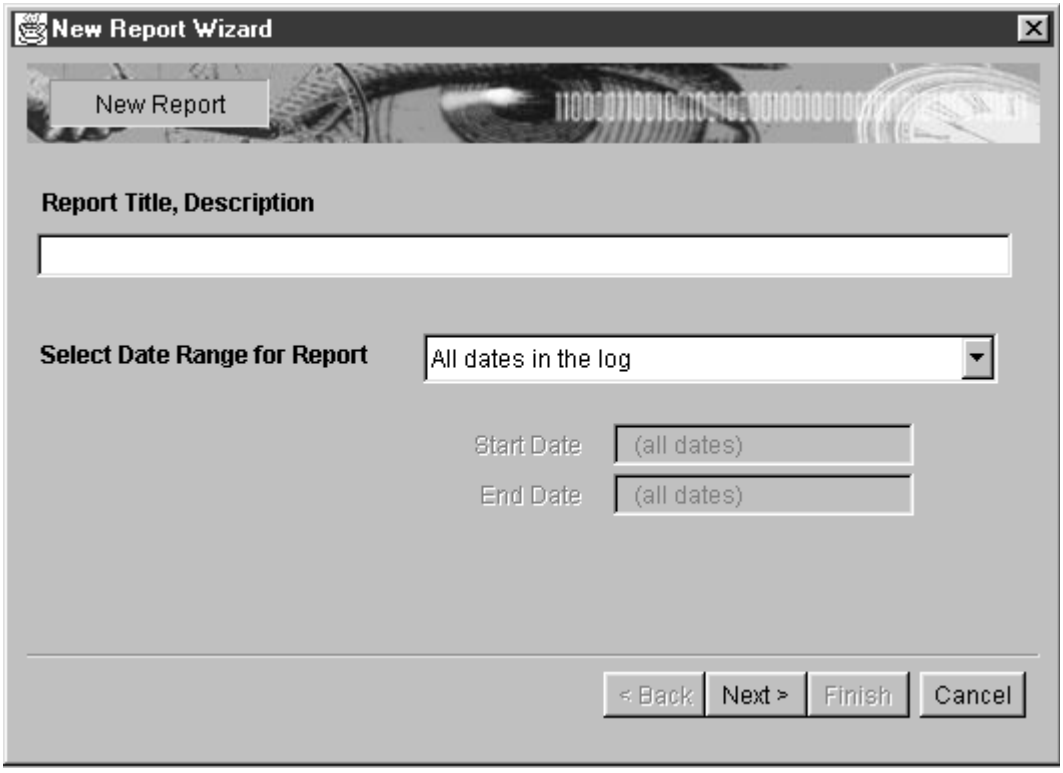
Defining a report

Before you can analyze a log file, you must set up the report.

To create a new report for a log file:

1 Click the New Report button.

The New Report Wizard appears.



2 Enter a report title or description.

3 Pull down the Select Date Range for Report list and choose a date range.

If you want to create your own date range, choose Specific Date Range, and type in the beginning and ending dates.

4 Click Next.

You see the second page of the wizard:

5. Enter the location of the log file or use the Browse button to find a file.

   The Weblog sample code includes a number of log files that you can use. If you wish, you can choose your own log file, as long as it's in Common Log File or Combined Log File format.

Note: In order to access a log file on a server, you need to have anonymous FTP access to that server. If you don't, you need to copy the log files to your local disk.

---

6. If you know the log file format you can select it, or choose Detect Format Automatically.
7. If you wish, enter a home page file, generally `index.htm` or `default.htm`.

   You do not have to enter a home page file.
8. Click Next.

You see the third page of the wizard:

9 Select the items you want to show in your report.

The default setting is for all options to be selected.

10 Click Finish.

You are now ready to analyze your log file.

Editing a Report Definition

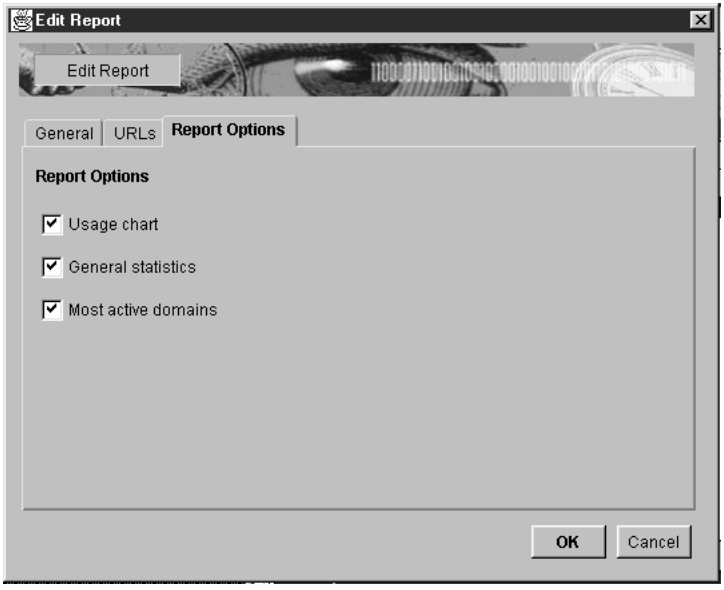
You can change a report definition after you've created it.

To edit report analysis settings:

1 Select a report from the Report list in the main window.

2 Click Edit.

The Edit Report dialog looks like this:



3 Make changes to the options on each tab as you wish.

4 Click OK.

Analyzing a log file

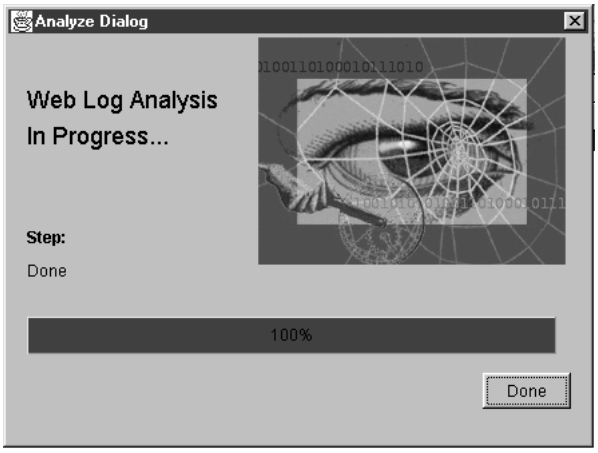
Once you've defined a report, you can tell Weblog to analyze the file.

To analyze a log file:

1 Select a report from the Report window.

2 Click Analyze Log.

The Analyze Dialog appears:



3 When the Analyzer finishes, click Done.

You are now ready to view the report of your log file.

Viewing a report

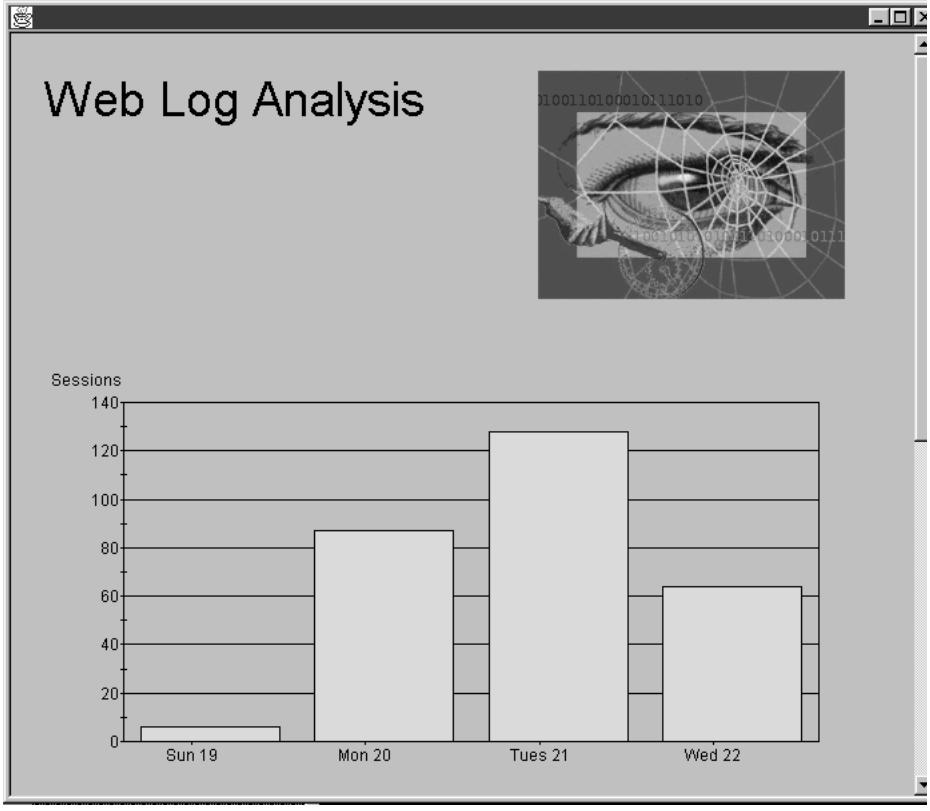
Once the log file is analyzed, the Weblog activates the View Report button.

To view a log file's report:

1 Select a report from the Report window.

2 Click View Report.

The report displays in a scrollable window:



3 When you're done viewing the report, click the close box in the upper right corner of the window.

Ending your Weblog session

You can exit Weblog by choosing Exit from the File menu or clicking the close button on the upper right side of the application window. Weblog saves the current report definitions (though not the reports themselves) and closes.

How Weblog works

When you run Weblog, the `WebLogAnalyzer.main` method executes. This method has three lines:

- It creates a `WebLogAnalyzer` object, calling the `WebLogAnalyzer` constructor. The constructor, which is automatically generated by Visual Cafe, instantiates and initializes the main window and its controls.
- It calls the `splashNLoad` method, which puts up the splash screen and loads previously saved data.
- It shows the main window.

Once the program shows the main window, it waits for user actions. The user can click on buttons or pick menu commands. Any of those actions cause events, which are caught by the corresponding components and processed. As is typical of well-behaved applications, Weblog thus spends most of its time waiting for the user to ask it to do something and then goes off and does it.

You can build an application like this in Visual Cafe by using the Form Designer to lay out components and then giving behavior to the components using the Interaction Wizard, which lets you associate code or actions with events. See the Chapter 9 of the Visual Cafe User's Guide for information. The rest of this chapter shows you some of the more interesting parts of Weblog, so you can see how you can do similar things in your own programs.

Serializing: saving and loading program data

Weblog keeps its data in an object of type `Data`, a class defined as part of this project, in `Data.java`. The program saves the contents of the data object so they will be available the next time you open the project.

When you want to save an object in Java, its class needs to implement the `Serializable` interface. To implement the `Serializable` interface you simply state that your class implements it; `Serializable` doesn't have any methods or fields, and just exists to tell Java that this class expects to save itself to disk. The Weblog `Data` class thus needs to fully define how the program's data is going to be saved. `Data` also contains methods for manipulating the data—creating new reports, deleting old ones, and so

on—that are called by the program when the user gives a command that changes the data.

The fields of class `Data` hold the program's data. You can break that data up into two groups:

- A set of program options
- The reports and an index for the current report

There is also a field that defines the name of the data file and a field, `theData`, that needs some more explanation.

Here's the part of the class definition that defines the fields:

```
public class Data implements Serializable {
    static final String INI_FILENAME = "WLA.dat";
    static Data theData = null;
    // Program Options
    int userSessionIdleLimit = 30;
    boolean ignoreUnexpectedLogFileErrors = false;
    // All of the reports
    Vector reports = new Vector();
    // The index of the currently selected report, or
    // -1 if none selected
    int currentReportIndex = -1;
```

The second field, `theData`, is an object of type `Data`. It is the only field that's initialized in the class constructor:

```
public Data() {
     theData = this;
 }
```

Whenever the `Data` object needs to access the current data, it uses the field `theData`. Initially, this is the same as accessing the current object directly, but the program can create a new `Data` object and replace the program's data by simply setting the value of `theData`.

You can see how this works by looking at the `loadDataInstance` method, which runs when the program starts and tries to load a previously serialized `Data` object. (Most of the `try` and `catch` statements were removed from the code here for clarity.)

```
public static String loadDataInstance() {
       String msg = null;
       Data data = null;
```

```
        try
            {FileInputStream istream = istream = new
            FileInputStream(INI_FILENAME);
            ObjectInputStream p = new
                ObjectInputStream(istream);
            data = (Data)p.readObject();
               p.close();
            istream.close();
        } catch(java.io.FileNotFoundException x) {}
```

The one `catch` statement that this copy of the code includes executes if the data file isn't found. In that case, the value of the variable `data` is never reset, so its value is `null`. The next lines in the method deal with that situation, which would occur the first time the program is run.

```
        }   if(data == null) {
            data = new Data();
            }
```

The next lines of code deal with the situation when a file is found and there is data in it.

```
        else {
               Data.setDataInstance(data);
            }
}
```

The `setDataInstance` method sets the value of `theData` so that it points to the object passed in, so `theData` now contains the previously saved data.

The method `saveDataInstance` saves the data when the program quits. The `saveDataInstance` method first calls the method `freeUpMem`, which decreases the size of the resulting file by clearing out the vectors that contain the reports. It then opens a stream and uses the Java `writeObject` method to serialize the program's data.

```
public static String saveDataInstance() {
    freeUpMem();
    String msg = null;
    try {
        FileOutputStream ostream = ostream = new
                  FileOutputStream(INI_FILENAME);
        try {
            ObjectOutputStream p = new
```

```
                    ObjectOutputStream(ostream);
                try {
                    p.writeObject(getDataInstance());
                   } catch(java.io.OptionalDataException x) {
                       msg = x.toString();
                    }
                // close the object output stream
                p.close();
               } catch(java.io.StreamCorruptedException x) {
                  msg = x.toString();
                }
            ostream.close();
          } catch(java.io.IOException x) {
              x.printStackTrace();
              msg = x.toString();
            }
          return msg;
    }
```

Using threads

A Java program can have multiple threads of execution. This means, conceptually, that there can be more than one thing going on at a time. In most cases, since most computers have only one processor, a multi-threaded program really does only one thing at a time, but having multiple threads allows the program to take actions in one area, while waiting for something to happen in another area and allows you to conceptually divide up tasks.

Weblog uses a separate thread, defined in `Analyzer.java`, to analyze the log files.

A thread needs to either subclass `java.Thread` or implement the `Runnable` interface, and needs to define a `run` method. The `Thread.run` method is similar to the `main` method of an application in that it's the entry point for the thread. The `run` method is also like `main` in that you never call `run`. When you want to run the thread, you call `start`. When you call the `start` method, Java sets up the thread and then calls `run`.

When the Analyze dialog opens, Weblog creates an analyzer thread and calls `start`. Here's the method that creates and starts the analyzer thread:

```
void AnalyzeDialog_WindowOpen
                    (java.awt.event.WindowEvent event)
{
    analyzer = new Analyzer(this);
    analyzer.start();
}
```

The program calls this method when a window open event occurs and the window that's opening is the Analyze dialog.

The `Analyzer` constructor does only one thing: it creates a progress tracking object that will show the user how the analysis is going. Here's the code:

```
public Analyzer(TrackProgress trackProgress) {
     this.trackProgress = trackProgress;
 }
```

("Giving the user progress-tracking feedback" on page 2-16 discusses the progress tracking code.)

`Analyzer` extends the `Thread` class, so it doesn't need to implement the `start` method. It implements the `run` method, which the Java code calls when the thread starts. The `Analyzer.run` method does two things:

- It uses the program's data to get the current report
- It tells the report to run, passing it the `trackProgress` object.

Here's the `Analyzer.run` method:

```
public void run() {
     Report report =
              Data.getDataInstance().getCurrentReport();
     report.run(trackProgress);
 }
```

The analyzer runs until the user dismisses the Analyze dialog. When that happens, the program generates a window-closing event, and this code executes:

```
void AnalyzeDialog_WindowClosing
                    (java.awt.event.WindowEvent event)
{
    // Stop the analyzer
```

```
        Analyzer.cancel(analyzer);
         // get rid of this dialog
         dispose();
    }
```

The method that stops a thread is actually `Thread.stop`. `Thread.stop` can leave objects in an uncertain state if it's called abruptly. That might happen if the user closed the dialog before the analysis was finished, so the `Analyzer` class provides a `cancel` method that waits for a while before calling `Thread.stop`. You may want to look at the code in `Analyzer.java` to see how you can make sure your program waits before stopping a thread.

Giving the user progress-tracking feedback

When a process might take a while, you want to give the user feedback about what's going on and how the process is progressing. Typically, you do that with a progress bar in a dialog box. The component library in Visual Cafe includes two progress bars: a Swing progress bar and a "heavyweight" progress bar for use in AWT applications. Weblog uses the heavyweight progress bar in its analyze dialog box.

Weblog organizes the progress tracking code by defining the `TrackProgress` interface in `TrackPrgress.java`, which is implemented by the `AnalyzeDialog` class in `AnalyzeDialog.java`. By using an interface, `AnalyzeDialog` encapsulates the progress-tracking code and separates it from the standard dialog box code. `TrackProgress` declares the following methods:

- `step`. This method takes a message and a progress percentage as parameters. It uses the progress percentage to set the current value of the progress bar and displays the message in a text label in the dialog box.
- `done`. This method changes the Cancel button to a Done button.
- `okCancelAlert`. This method puts up an OK/Cancel dialog that lets a user confirm a choice.
- `okAlert`. This method puts up a confirmation dialog that only has an OK button.

When the user clicks the Analyze Log button, and thus asks Weblog to analyze a log file, the code creates an `AnalyzeDialog` object. The

`AnalyzeDialog` code starts the analyzer thread, which actually does the analysis. Periodically, the analyzer thread calls the `TrackProgress.step` method with a call like this:

```
trackProgress.step("Parsing log file...", percent);
```

This call updates the position of the progress bar and also updates the progress message.

Creating a wizard

Weblog uses a wizard to gather information for a new report. In Visual Cafe, you create a wizard by putting a `Wizard` component onto a form and adding panels. Weblog uses a wizard in its New Report dialog; if you want to see the Weblog implementation, see `NewReportDialog.java` and the Form Designer for the `NewReportDialog` object.

To create a wizard:

1. Create a new project or open an existing project.

   The project can use any project template.

2. In the Objects view of the Project window, select the top-level container in the project.

   The top-level container is at the top of the Objects list.

3. Choose Form from the Insert menu.

   The form is the top-level container for the wizard. Usually you put a wizard in a dialog box so that you can make it appear and disappear at appropriate times.

4. Choose `Dialog` or, for a Swing project, `JDialog`.

   A new Form Designer appears, showing an empty dialog box.

5. From the AWT Additions tab of the Component Palette, add a `Wizard` component.

6. Expand the `Wizard` component so it fills the part of the dialog box that you want it to fill.

   You will often want the wizard to fill the entire box, but you might want to reserve part of the box for some constant information, such as a picture. Weblog does this.

   Notice that the wizard contains the usual set of wizard buttons (Back, Next, Finish, Cancel, and Help). This lower section of the wizard

doesn't change when the user moves between pages, except that the wizard enables and disables buttons as appropriate.

7 From the AWT tab of the Component Palette, choose `Panel` or, for a Swing project, from the Swing Containers tab of the Component Palette, choose `JPanel`.

You can add any component to a wizard, but you usually want to add panels.

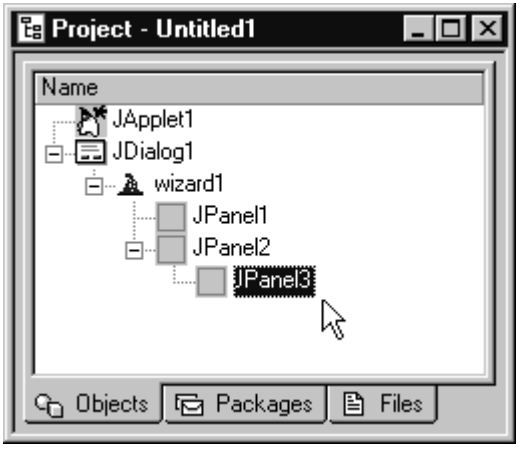
8 Drop the panel into the `Wizard`.

The panel is automatically sized to fill the wizard's page area.

9 Add a panel for each wizard page that you want.

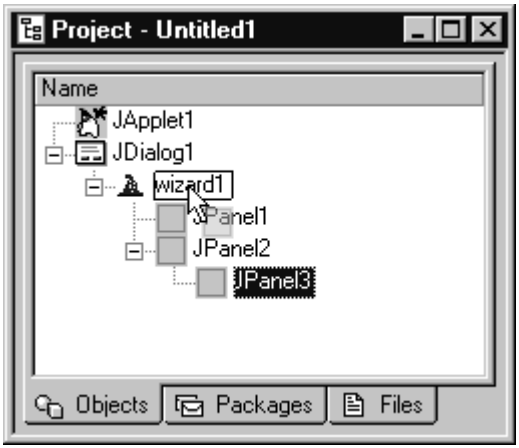
Check in the Objects view of the Project window to make sure that the panels are added at the correct level. They should all be contained in the wizard.

For example, in this example, `JPanel3` is at the wrong level; it is contained in `JPanel2` rather than directly in the wizard:

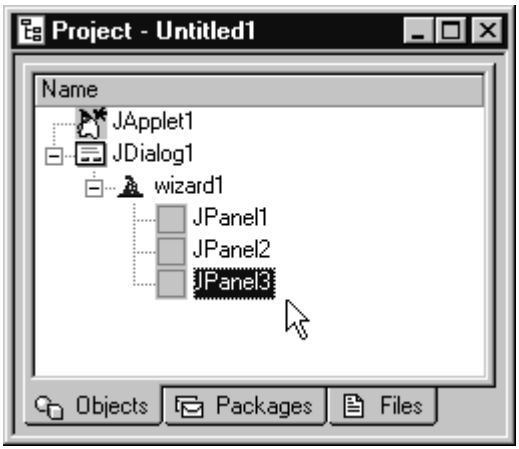


You can change the container of an object by dragging it in the Objects view. Click the object you want to move and drag it to the

object that you want to contain it:. (A shadow of the object follows the mouse pointer.)



After performing this move operation, the `JPanel` is on the correct level:



Once you have more than one panel in the wizard, the Back and Next buttons in the wizard become enabled and you can use them to navigate between the panels, even in the Form Designer

10 Design the panels as you would any GUI.

The `Wizard` component handles the Back, Next, and Cancel buttons for you; unless you want to do something special when the user presses any of those buttons, you don't have to do anything about them. You wait until

the user presses Finish, and then you collect the data from the wizard and take whatever action the wizard promised.

To handle the Finish button:

1. In Visual Cafe, right click on the `Wizard` component.
2. From the pop-up menu, choose Add Interaction.
3. Add an `actionPerformed` event interaction. Make the action to hide the wizard. (Doing this requires several steps in the Add Interaction wizard. See the *User's Guide* or the *Getting Started Guide* for details of how to use the wizard.)
4. In the Visual Cafe Source window, open up the Java file for the form that contains the wizard.
5. Find the `wizard1_actionPerformed_Interaction1` method.

   If the name of the `Wizard` component is something other than `wizard1`, that will show in the name of this method. Similarly, if you've already added an interaction to this wizard, the interaction number will be something other than 1.

   The code will be something like this:

```
wizard1_actionPerformed_Interaction1(
                    java.awt.event.ActionEvent event)
   {
   try {
      wizard1.setVisible(false);
   } catch (Exception e) {}
}
```

   This code executes when any action-performed event occurs in the wizard, which isn't normally what you want to do. You need to take action only if the user pressed the Finish button.

   The event includes the name of the button that caused the event, and you can use that to make the method do what it should do.

6. Add the following code to the beginning of the method, before the `try` statement:

```
String cmd = event.getActionCommand();
if(cmd.equals("Finish")) {
   //your code goes here
```

If you want to take special action with the Next, Previous, Cancel, or Help buttons, you can do so by looking for those command names.

7 Add a `}` before the end of the method.

The method should now look like this:

```
wizard1_actionPerformed_Interaction1(
                    java.awt.event.ActionEvent event)
{
    String cmd = event.getActionCommand();
    if(cmd.equals("Finish")) {
        //your code goes here
        try {
            wizard1.setVisible(false);
        } catch (Exception e) {}
    }
}
```

Validating input with the Wizard component

When the user presses the Next button in your wizard, you should check the entries on that page to make sure that everything is valid. The best way to do that is by extending `SimpleWizardControlle` .

A `Wizard` component is controlled by a wizard controller, which is an object that implements the `WizardController` interface. `SimpleWizardController` is the default controller, which is generally sufficient except for its lack of validation. Extending `SimpleWizardController` to add validation is fairly easy—the only method you need to override is `validatePage`. The wizard calls this method whenever the user presses any of the wizard buttons. If you return `false` from this method, the wizard ignores the button press and stays on the same page.

Here's the header of `validatePage`:

```
public boolean validatePage(  Component comp,
                              Component target,
                              int action)
```

The first parameter, `comp`, is the component that occupies the page of the wizard that the user was on when the button was pressed.

The second parameter, `target`, is the component that occupies the page of the wizard that will show next.

The third parameter, `action`, indicates which button the user pressed. It is an integer, but `WizardController` defines a set of constants (`NEX` , `PREVIOUS`, `FINISH`, `CANCEL`, and `HELP`) that you should use instead of the integers.

The body of Weblog's `validatePage` method is described below.

First, `validatePage` gives the standard validation method a chance to pass on the page:

```
if(!super.validatePage(comp, target, action)) {
   return false;
        }
```

Before checking if the page is valid, this version of the method checks if the user has pressed the Next button. In general, you probably don't want to validate a page when the user presses Cancel, Previous, or Help. You might want to when the user presses Finish. Weblog doesn't need to validate the last page because it's not possible for the entries on that page to be invalid (that page just contains a set of checkboxes).

```
if(action != WizardController.NEXT) {
            return true;
         }
```

The validation methods need to know what kind of data to expect, so Weblog checks which page (`panel1` or `panel2`) of the wizard the user is leaving, and calls the appropriate validation method. If the validation method returns `false`, then the `validatePage` method returns `false`; otherwise, `validatePage` returns `true`.

```
// Handle according to page leaving
      if(comp == panel1) {
   if(!validatePage1()) {
      return false;
   }
} else if(comp == panel2) {
   if(!validatePage2()) {
      return false;
   }
}
// Passed validation check OK
return true;
```

```
        }
    }
```

Returning `false` from `validatePage` makes the wizard ignore the button click and remain on the same page, but you also need to give the user feedback about what's happening. The validation methods put up dialog boxes notifying the user of the validation problem.

Parsing and validating date entries

Weblog encapsulates a number of date-format validation and conversion methods in the `WLAUtil` class, defined in `WLAUtil.java`. These methods are used as static calls—you can't instantiate `WLAUtil`. You may find this class useful if you write a program that needs to validate and parse dates.

`WLAUtil` has a large number of utility methods. This section discusses only a small part of what's available.

The wizard validation code uses the `validateDateField` to check if an entered date is valid. Here's the method that calls `validateDateField`:

```
boolean validatePage1() {
     // Only validate start/end date if specific
    // user-supplied dates
     int timeframe = timeframeChoice.getSelectedIndex();
     if(timeframe == Report.TIMEFRAME_SPECIFIC) {
         if(!WLAUtil.validateDateField(this,startDateText)){
             return false;
         }
         if(!WLAUtil.validateDateField(this,endDateText)){
             return false;
         }
     }
     return true;
}
```

`TIMEFRAME_SPECIFIC` is a constant that indicates that the user chose to put in specific dates. (This page of the wizard allows the user to choose a range of dates such as Past two days or Yesterday or to enter a specific date range.) If the user did enter specific dates, then the dates need to be validated. That's done by `WLAUtil.validateDateField`. That method gets passed the text field that contains the data that needs to be checked. If

the method can parse the value as a date, it returns `true`. If the method can't parse the value as a date, the method:

- moves the focus to the text field,
- selects the contents so that they're highlighted for the user,
- puts up a dialog box that tells the user of the problem, and
- returns `false`

Here's the method:

```
static boolean validateDateField(Window dialogOrFrame,
                                 TextField dateField) {
   try {
      // Parse the date to ensure OK
      string2Date(dateField.getText());
      // Parsed OK. Passes validation
      return true;
   } catch(java.text.ParseException x) {
   // Can't parse the given date. Move focus to
   // offending field
   dateField.requestFocus();
   dateField.selectAll();
   // Alert user
   new AlertDialog(getFrame(dialogOrFrame), false,
                                 x.getMessage());
        return false;
    }
}
```

The method that's called to parse the text is `string2Date`. It's a very simple method:

```
static Date string2Date(String dateString)
                throws java.text.ParseException {
    return dateFormat.parse(dateString);
}
```

Java supplies the `dateFormat.parse` method, so the hard work of actually parsing the text is taken care of for you.

`WLAUtil` has a number of methods that you may find useful in handling date and time strings. See the code for information.

Accessing files using URL and URLConnection

You can access files on a local system or over a network using the Java `URL` and `URLConnection` classes. In Weblog, the `parse` method that's part of the `Parser` class (defined in `LogFile.java`) uses these classes to open the log file that the user specifies in the New Report wizard.

Letting the user browse for a file

The New Report wizard lets the user enter a file specification or browse the file system to find a file. When the user clicks on the Browse button, the button code calls the `WLAUtil.browseForURL` method. That method returns a URL string.

```
public static String browseForURL(String template) {
    // Access main frame which has the open dialog off it
    WebLogAnalyzer wla = WebLogAnalyzer.theWLA;
    // set the default suffix
     wla.openFileURLDialog.setFile(template);
    // Show the OpenFileDialog
    wla.openFileURLDialog.show();

    // get the results
    String results = wla.openFileURLDialog.getFile();
    if(results != null) {
        results = wla.openFileURLDialog.getDirectory() +
                                                  results;
       results = WLAUtil.cleanupURLName(results);
    }
    return results;
}
```

When you create an application project using the Visual Cafe application template, Visual Cafe automatically adds an `OpenFileDialog` object. In Weblog, the name of that object was changed to `openFileURLDialog`, but it's still just the standard Open File dialog box. The method `WLAUtil.cleanupURLName` modifies the result of the file dialog to produce a URL. (URLs consist of a protocol string, such as `http` or `file`, a colon, and a path string.) In addition to being called from `browseForURL`, this method is called when the URL text field loses focus, so it must be able to understand and fix URLs that the user typed in as well as file information returned by the Open File dialog. The

`cleanupURLName` method actually looks for only a few features of correctly written URLs, and will accept any string that contains a colon. When it receives the result of the Open File dialog, it adds `file:///` to the beginning of the string.

```
static String cleanupURLName(String messyURLText) {
    // Infer starting "file://", etc as needed
    try {
        char ch;
        StringBuffer buf;
        // Colon?
        int idx = messyURLText.indexOf(':');
        if(idx < 0) {
            // No colon.
            buf = new StringBuffer(messyURLText.length() +
                                                   9);
            ch = messyURLText.charAt(0);
            if(ch == java.io.File.separatorChar) {
                // Presume file entry
                buf.append("file:///");
                buf.append(messyURLText);
            } else {
                //Prepend http:// by default
                buf.append("http://");
                buf.append(messyURLText);
                // if no slashes in messyURL, presume
                // only domain given and end with one
                   if(-1 == messyURLText.indexOf('/')) {
                      buf.append('/');
                   }
                }
                return buf.toString();
            }
            // Have a colon in the messyURLText,
            // look at char after the colon
            ch = messyURLText.charAt(++idx);
            if(ch == '/') {
                ch = messyURLText.charAt(++idx);
                if(ch == '/') {
                   idx = messyURLText.indexOf(':',
                                                ++idx);
```

```
                        if(idx < 0) {
                            // only initial "//" provided.
                            // Add ending one.
                            buf = new StringBuffer
                                (messyURLText.length() + 2);
                            buf.append(messyURLText);
                            buf.append('/');
                            return buf.toString();
                        }
                    }
                    // protocol specified. Nothing to do.
                    return messyURLText;
                }
                if(ch == '\\') {
                    // MSDOS file path. Prefix file:
                    // protocol
                    return "file:///" + messyURLText;
                }
                //colon, but no slash...do nothing
            } catch(StringIndexOutOfBoundsException x) {
            // if get this, no cleanup
        }
    return messyURLText;
}
```

Opening a connection to a URL

The `parse` method, defined in `LogFile.java`, begins by

- creating a URL object,
- opening a connection to it,
- opening an input stream using the connection, and
- creating a stream reader for the file

Between each of these steps, the method checks to make sure that the thread hasn't been cancelled and updates the progress information in the New Report dialog.

Here's the beginning of the `parse` method:

```
public void parse(TrackProgress trackProgress)
```

```
                                throws ParseLogException {
            URL url;
            URLConnection con;
            InputStream is;
            int contentLength;
            int lineCount = 0;
        // Collect garbage mem before doing this
        // potentially memory-intesive task
            System.gc();
            Vector recordVector = new Vector();
            // Initialize File
            try {
                // Create URL
                Analyzer.throwExceptionIfCurrentThreadCancelled();
                trackProgress.step("Creating URL...", 1);
                url = new URL(logFileURL);
                // Get connection to log file
                Analyzer.throwExceptionIfCurrentThreadCancelled();
                trackProgress.step("Opening connection...",2);
                con = url.openConnection();
                // Get input stream of log file
                Analyzer.throwExceptionIfCurrentThreadCancelled();
                trackProgress.step("Accessing logfile...",3);
                is = con.getInputStream();
               logReader = new BufferedReader(
                           new InputStreamReader(is));
```

Once the stream reader has been instantiated, you can use it to read the file:

```
if(null == (aLine = logReader.readLine())) {
                    break;
                }
```

Adding a splash screen

A splash screen lets the user know what a product is, who makes the product, and who owns the copyright. It also places something on the screen so the user knows that the program is starting.

A splash screen can be created easily in Visual Cafe by using the Form Designer and the `Dialog` and `ImageViewer` components.

The code that's automatically generated by Visual Cafe:

- instantiates/creates a dialog,
- sets the size of the dialog,
- sets the background color to light gray,
- instantiates the `ImageViewer` component as `imageViewer1,`
- sets the size of the `imageViewer1` component,
- provides the location of the image to view, and
- provides exception handing using `try` and `catch`

To autogenerate code similar to the code in SplashDialog.java:

1. Create a new project.
2. Drag the `Dialog` component from the Forms folder in the Component Library into the Objects tab of the Project window.

   Visual Cafe instantiates the `Dialog` component as `dialog1` and displays a Form Designer for the dialog.
3. Double-click on the `Bounds` property in the Property List. Change the `Width` to 419 and the `Height` to 290.

   This setting determines how large the splash screen will be.
4. Change the `Background` color to light gray.
5. Change the name from `dialog1` to `SplashDialog`.
6. Drag the `ImageViewer` component from the Multimedia folder in the Component Library into the `SplashDialog` Form Designer

   Visual Cafe instantiates the `ImageViewer` as `imageViewer1`.
7. Make sure `imageViewer1` is selected and double-click on the `Bounds` property in the Property List. Change:
   a. Width to 419
   b. Height to 290
   c. X to 0
   d. Y to 0
8. Double-click `ImageURL` in the Property List. Click the Browse button to select `weblog\images` from the Visual Cafe CD.

Displaying the splash screen during loading

You want the user to be able to see the splash screen for a few seconds, to appreciate your artistry and read the copyright, but you don't want things to grind to a halt while the user is reading. That's easy enough to do. Weblog keeps the splash screen visible long enough for the user to read it, while at the same time continuing to load the application; it does so in a method called `SplashNLoad`.

`SplashNLoad`:

- displays the splash screen,
- checks the time,
- loads serialized data,
- uses the data to initialize the report list,
- enables the components,
- checks the time again,
- if it hasn't been three and a half seconds yet, goes to sleep until three-and-a-half seconds have passed, and
- takes down the splash screen

Here's the code:

```
   void splashNLoad() {
       // display the splash screen
       SplashDialog splash = new SplashDialog(this,false);
       splash.show();

       // note time that it was displayed
       long startMillisecs = System.currentTimeMillis();

        // Load data
       String msg = Data.loadDataInstance();
        // Display msg, as needed
        if(msg != null) {
             new AlertDialog(this, false, "Error reading
program initialization file (" + msg + "). Using
defaults.");
        }
        // Get my data instance
```

```
        data = Data.getDataInstance();

        // Initialize report list component with program data
        initializeReportList();
        // Enable/disable components as needed
        enableComponents();

      // be sure splash shows for at least a few seconds
      long millisecs = System.currentTimeMillis();
      // get length of time to load data
      millisecs -= startMillisecs;
      // want to splash for about 3.5 secs, total
      millisecs = 3500 - millisecs;
      if(millisecs > 0) {
          try {
                Thread.sleep(millisecs);

          } catch(java.lang.InterruptedException x) {
            }
      }

        // take down splash screen
        splash.hide();
}
```

Limitations and known problems

If your log file is extremely large, you may get a `java.lang.OutOfMemory` error. You can use the `-mx` option to increase the amount of memory the VM is using for Weblog. Do not exceed the total amount of physical RAM on your PC as the VM hangs when it runs out of physical memory.

To increase the amount of memory the JVM uses:

- Enter a command like this:

  ```
  java -mx64M weblog
  ```

  where 64 is the total amount of physical memory (in megabytes) present in your computer.

Limitation due to Java

A dialog's parent must be a frame. This means you can't have a modal dialog (like `AlertDialog`) that modally appears when a dialog (like `EditReportDialog`) requests it. When an alert appears in this situation it is modal to the main frame, *not* to the dialog that created the alert. A similar problem exists with the `FileDialog`.

Note: The parsed log file is not stored when the user quits the program. So, if the user re-runs the program and changes a report parameter, the log file needs to be read again so that it can be reanalyzed.

CHAPTER 3

Electronic Software Distribution Servlet

This chapter discusses the Electronic Software Distribution (ESD) servlet, which allows a user to buy a piece of software over the Internet. The code created for this example is available in :

```
VisualCafe\Sourcebook\Servlet
```

You can view the source code in Visual Cafe and copy classes and methods into your own servlets, applications, or applets. This chapter discusses the code in detail to show how the servlet was implemented in Visual Cafe.

A *servlet* is a Java program that can be thought of as a server-side applet. That is, a servlet extends the capabilities of a server in the way that an applet extends the capabilities of a browser.

More precisely, a servlet is a Java class based on the Java `Servlet` interface. A servlet runs on a Web server and, in general, does the kinds of things that you might do with CGI scripts, with these advantages:

- Servlets are written in Java.
- Where a CGI script needs a new process to handle each request, a servlet can handle many requests at a time. The fact that there is only one instance of a servlet also means that the servlet only needs to load once. Also, a servlet can share information between users.

Since a servlet runs on a server, it has no graphical user interface. Servlets can extend server functionality in any way you want, but they generally serve Web pages to users and read and respond to user input on HTML forms. The sample servlet in this chapter serves preexisting Webpages, but many servlets create Web pages in response to user input.

Setting up the servlet

You need a server that can run Java servlets in order to use the ESD servlet. Sun's Java Web Server, which is written in Java, is one choice, but most other Web servers currently support the servlet extension.

To set up the servlet:

1. Install a Web server that can run Java servlets, if you don't already have one installed. The rest of these instructions assume that you've installed the Web server in a directory with the name `serve` .
2. Set up the server to respond to requests on port 80, which is the default port for Web services.
3. You need to copy some of the files in `VisualCafe\Sourcebook\Servlet` into the server's directory structure.
   a. The server should have a `public_html`, a `private_html` directory, and a `servlets` directory. If it doesn't, create them.
   b. Create a `symantec` directory in `public_html`.
   c. Find the `image` directory in `VisualCafe\Sourcebook\Servlet\ESDServlet` and copy it into the `server\public_html\symantec` directory.
   d. In the `server\servlets` directory, create a `symantec\sourcebook\` directory.
   e. Copy the `servlet` directory from `ESDServlet\symantec\sourcebook` into the `symantec\sourcebook\` directory you just created,
   f. Copy the `creditcheck` directory from `CCServlet\symantec\sourcebook` into the same directory.
4. Using your server's administrative tool, you need to define aliases for the ESDServlet and CreditCheck:

Alias	Servlet
`purchase`	`symantec.sourcebook.servlet.ESDServlet`
`creditcheck`	`symantec.sourcebook.creditcheck.CreditCheck`

5 Change the value of the `successCGI` field in `verifyinfo.html` to reflect the actual address of your server. For example:

```
<P><INPUT TYPE="HIDDEN" NAME="successCGI" SIZE="-1"
 VALUE="http://www.mysite.com/servlet/purchase/down">
 <!--fieldPosition--></P>
```

Replace `www.mysite.com` with the domain you are using.

Caution: Purchasing servlets must be run on a secure server (https protocol) to assure credit card security. `Private_html` should be a private directory, not accessible to the Web page service. Even though the ESD servlet does not actually make purchases, you should observe these security precautions if you plan on using real credit card numbers for testing.

Operational overview of the ESD servlet

The ESD servlet mimics a servlet that accepts a user's credit card number over the Internet, charges the card, and download the software package that the user has purchased. (It doesn't use a real credit checking service, and doesn't actually charge the credit card.)

The ESD servlet works as follows:

1 The client performs an HTTP `GET` operation on the servlet's base URL (for example, `http://www.yoursite.com/servlet/purchase`).

2 The servlet serves a blank form containing the following fields:

`page` - A hidden field containing the form's page number, in this case "1".

`name` - The name on user's credit card.

`company` - The name of company.

`email` - The user's e-mail address.

`address` - The user's physical address.

`city` - The user's city.

`state` - The user's state.

`zip` - The user's zip.

`country` - The user's country.

`phone` - The user's phone.

3 The user fills in the fields and presses the Continue button. This causes the client to do an HTTP `POST` command, sending the field data to the servlet.

4 The servlet examines the posted fields. If the fields are ok, the servlet serves the verification form. The verification form contains the following fields:

`successCGI` - The URL to be accessed if the credit card is processed successfully.

`orderId` - The unique ID assigned for this order.

`ccTotal` - The total dollar amount to charge the user's credit card.

`ccName` - The name that appears on the credit card.

5 The user examines the values on the screen and presses the Continue button if everything looks OK. This causes the client to perform an HTTP `POST` command to the Credit Check servlet. (In this example the Credit Check servlet does no actual card verification. In a live system this form would be processed by a third party credit card house.)

6 The Credit Check servlet serves a form that allows the user to enter the card number, expiration date, and card type. When the user presses the Continue button, the servlet theoretically charges the credit card. If the charge is accepted, the Credit Check servlet redirects the browser to the URL indicated by the `successCGI` field. The `successCGI` is passed the unique orderID for this order. We have set the `successCGI` field to direct control back to the ESD servlet.

7 The ESD servlet then validates the orderId. If the orderId is valid, the ESD servlet checks with the Credit Check servlet to be sure the card has cleared (this step is required to prevent someone from simply running the `successCGI` directly, bypassing the credit check). If things check out OK, the servlet serves the software owner's page. The owner's page contains the following fields:

`orderId` - Hidden field with this owner's unique order ID.

`page` - Hidden field with the current page number

8 The user presses the Download button and this form is posted to the servlet. The servlet verifies the order ID and the card's approval (again to prevent hacking). If all is in order the servlet downloads the file that the user has just purchased.

How the servlet works

The ESD servlet is based on the Sun `HttpServlet` class, which is an abstract class designed to handle HTTP requests.

When a server receives an HTTP request directed to a servlet, the server calls the servlet's `service` method. In general, servlets receive `GET` requests and `POST` requests—a `GET` is usually a request for a Web page while `POST` is usually information sent from a filled-out form.

When the `HttpServlet.service` method receives a `GET` or a `POST` request, it calls the `servlet.doGet` or `servlet.doPost` method, as appropriate. When you want to write a servlet that handles HTTP requests, you usually subclass `HttpServlet` and override the appropriate "`do`" methods rather than overriding the `service` method. In the ESD servlet, `doGet` calls `doPost`, so `POST` and `GET` requests are handled in the same way. The ESD servlet's `doPost` method does this:

- If this is the first request, it puts up an initial HTML page, which is a form asking for user information.
- For subsequent requests, it determines which form was filled out and creates an object, called a *page handler*, that knows how to handle that form.
- It asks the page handler to check if submitted information is valid.
- It asks the page handler what the next page should be.
- It creates a page handler for the next page.
- It asks the new page handler to serve up the next page.

Thus, the servlet class itself is fairly simple. The "knowledge" about where the forms are located, how to understand the information in the form, the order the forms need to be presented, and what to do when information is invalid, is in the page handlers. If you want to make your own servlet that handles similar tasks, you may be able to simply use the ESD servlet code, and implement your own page handlers. In the case of this servlet, the pages are fixed HTML pages, rather than pages created on-the-fly by the page handlers, but you could modify the page handler classes to build custom pages for each user.

Deeper into the classes

The Java `Servlet` interface is defined in `javax.servlet.Servlet`. (The "`x`" in "`javax`" indicates that the `servlet` package is a Java extension, rather than a required part of every Java VM.) The `Servlet` interface defines the minimum set of methods that a servlet needs to have. The most important method for our purposes is `service`. The `service` method handles a single request from a client. At this level, "a client" is not defined, and neither is a request.

The `javax.servlet` package also defines the abstract class `GenericServlet`. It adds more definition to the `Servlet` interface, and provides default implementations for some of the methods.

Finally, the `javax.servlet` package defines the abstract `HttpServlet` class. `HttpServlet` extends `GenericServlet` to handle the HTTP protocol by defining the `service` method so that it understands HTTP service requests such as `POST` and `GET`, and calls corresponding `HttpServlet` methods like `doPost` and `doGet`. Thus, we now have a definition for a client and a request: a client is a Web client, and a request is an HTTP request. To define a servlet that can handle HTTP requests, all you have to do is subclass `HttpServlet` and define the methods that correspond to the HTTP requests that you want to handle.

The `symantec.sourcebook.ESDservlet` package defines the `ESDServlet` class, which extends `HttpServlet`. `ESDServlet` implements `doGet` so that it calls `doPost` and implements `doPost` so that it can handle a series of HTML forms by creating page handlers that correspond to the forms.The `ESDservlet` package also defines a set of pages and page handlers. All of the "intelligence" needed to supply the forms and understand the input is pushed down into the page handlers.

HTML forms

HTML forms are outside the scope of this book, but here's a little background to enable you to understand how the servlets use them.

HTML includes the `FORM` tag. The `FORM` tag defines a fill-in form. A form can contain `INPUT` tags, which define and name input fields, and also declare the types of the fields. Aside from the obvious types like text fields and checkboxes, you can have `INPUT` fields that are hidden. *Hidden fields*

send information back to the server that the user doesn't see. Hidden fields are used by the ESD servlet to identify the different HTML pages.

If you're wondering how the page knows when to send information back to the server, one of the `INPUT` types defines a Submit button, which allows the user to send the information in the form to the server.

Here's a very simple form:

```
<HTML>
<FORM METHOD = POST>
<INPUT TYPE = "HIDDEN" NAME = "SILLY FORM" VALUE = "1">
<INPUT TYPE = "TEXT" NAME = "NAME">
<INPUT TYPE = "SUBMIT">
</FORM>
</HTML>
```

In a browser, this code produces a form like this:

The `METHOD` parameter of the `FORM` tag defines the HTTP request type. Thus, when the user presses the Submit Query button, this code sends an HTTP `POST` request with two parameters: one called `NAME`, whose value is the text the user entered in the text box and one called `SILLY FORM` whose value is 1. Since the input named `SILLY FORM` is marked as `TYPE=HIDDEN`, the user does not see it in the form, but it is still in the stream of data that the servlet receives.

ESD servlet Code

The ESD servlet example consists of two Visual Cafe projects.

- The ESDServlet project defines the `ESDServlet` class, which is the class for the ESD servlet, and a set of page handlers for the forms used by this servlet. It also defines a `Transaction` class that handles the data of the transaction and a `PageData` class to hold the data for a single page.
- The creditcheck project defines a "place holder" servlet that checks the validity of the credit card number that the user submits. In a real servlet, you would have the credit card validated by an outside service;

this example includes the Credit Check servlet so that you can see the servlet work without setting up a real credit-checking account. Since this is just a place holder class, it is not discussed in this chapter.

The ESDServlet project

The Servlet project defines the `ESDServlet` class, the `Transaction` class, the `PageData` class, and seven page handler classes.

The ESDServlet class

The `ESDServlet` class, defined in `ESDServlet.java`, extends the `HttpServlet` class, filling in the specific behavior required for the ESD servlet. It was made by:

- Choosing the New Project command from the File menu
- Choosing the Servlet template for the project
- Filling out the pages of the Servlet wizard to create the basis for the code
- Finishing the project by hand-coding

As noted, the class extends the HttpServlet class, so it begins like this:

```
public class ESDServlet extends HttpServlet
{
```

The class defines constants for seven page names:

```
    protected static int NOFORM_PAGE_NUM = 0;
    protected static int ENTERINFO_PAGE_NUM = 1;
    protected static int VERIFYINFO_PAGE_NUM = 2;
    protected static int OWNER_PAGE_NUM = 3;
    protected static int DOWNLOAD_PAGE_NUM = 4;
    protected static int SUCCESS_PAGE_NUM = 5;
    protected static int ERROR_PAGE_NUM = 6;
```

The class adds a field that stores an array of page handler names. The servlet uses these to create page handler objects that actually do the work of the servlet.

```
public String[] pageHandlerNames = {
                "symantec.sourcebook.servlet.PHNoForm",
```

```
                    "symantec.sourcebook.servlet.PHEnterInfo",
                    "symantec.sourcebook.servlet.PHVerifyInfo",
                    "symantec.sourcebook.servlet.PHOwner",
                    "symantec.sourcebook.servlet.PHDownload",
                    "symantec.sourcebook.servlet.PHSuccess",
                    "symantec.sourcebook.servlet.PHError"};
```

The `getServletInfo` method returns a brief description of the servlet. This method may be called by server administrator functions. It was created by the Servlet wizard.

```
public String getServeltInfo()
{
   return "Sample ESD servlet";
}
```

In general, the HTML pages that the ESD servlet uses send HTTP `POST` requests, but when the user first invokes the servlet it may receive a `GET` request. In any case, `ESDServlet` forwards `GET` requests to the method that handles `POST` requests.

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

The `doPost` method is the center of `ESDServlet`.

```
public void doPost(
                    HttpServletRequest request,
                    HttpServletResponse response)
                    throws  ServletException, IOException}
```

The method first takes a look at the grossest characteristic of the information received. If it is very large, then there is certainly something wrong with it, and it should be rejected.

```
if(request.getContentLength() > 4096)
{
```

```
    resp.sendError(HttpServletResponse.SC_BAD_REQUEST);
    return;
}
```

Once the message has passed that simple test, `doPost` finds out what form was submitted.

```
int pageNumber = getPageNumber(req);
```

(The `getPageNumber` method is shown later.)

Once `doPost` knows what the page number of the form is, it can get the page handler for that page from the `getPageHandler` method (which is also shown later).

```
PageHandler pageHandler = getPageHandler(pageNumber);
```

The page handler knows what the information from this form is supposed to look like, so `doPost` asks the page handler if the information is valid.

```
Object pageData = pageHandler.validate
                    (getServletConfig(),req,resp);
```

The `doPost` method then asks the page handler for the page number of the next form. It then creates a page handler for *that* page, and serves the page to the user

```
    pageNumber = pageHandler.nextPage();
    pageHandler = getPageHandler(pageNumber);
    pageHandler.serve(pageData,resp);
}
```

That is the end of the `doPost` method.

The next method gets the page handler for a given page number.

```
protected PageHandler getPageHandler(int pageNumber)
                                    throws ServletException
{
```

First, the `getPageHandler` method gets the page handler's name from the list of page handler names that the class holds. It then uses that string in a Java static method that creates an instance of a class with a given name. Finally, it returns that new object. The method ends with some code to handle potential errors.

```
String pageHandlerName = pageHandlerNames[pageNumber];
    try
    {
        PageHandler pageHandler = (PageHandler)
```

```
            (Class.forName(pageHandlerName)).newInstance();
        return pageHandler;
    }
    catch(Exception e)
    {
    }
    // some sort of exception occurred trying to instantiate
    throw new ServletException(
                    "(Order) Internal servlet error (1)");
}
```

The server calls the `init` method when it instantiates the servlet.

```
public  void  init(ServletConfig config) throws
 ServletException
{
    // perform standard init
    super.init(config);
    // try to initialize the recent transaction info
    try
    {
        Transaction.InitializeTransactions();
    }
    catch(Exception e)
    {
        // Some exception occurred. Let server know that this
        // servlet is unavailable.
        UnavailableException u = new
            UnavailableException(this,e + " " +
                                    e.getMessage());
        throw u;
    }
}
```

The server calls the `destroy` method when the servlet is shut down.

```
public void destroy()
{
    try
    {
        Transaction.WriteTransactions();
    }
```

```
    catch(IOException e) {}
    super.destroy();
}
```

The `getPageNumber` method returns the current page number, which is kept in a hidden field in each page and can be obtained from the HTTP request. If there isn't a page parameter available (`pageString == null`), then that means that either:

- this is the initial request, so this method should return "page 0" so that the next page served up is page 1,
- or, this is a request to re-download the purchased program, so the download page (`SUCCESS_PAGE_NUM`) should be reloaded.

```
public int getPageNumber(HttpServletRequest request)
{
    int pageNumber = NOFORM_PAGE_NUM;
    String pageString = request.getParameter("page");

    if(pageString == null)
    {
        // we've set up the HTML so that the download page
        // has a specific URL... this allows the user to
 bookmark
        // the page and return there if the initial
        // download attempt fails
        if(request.getRequestURI().startsWith("/servlet/
 purchase/down"))
            pageNumber = SUCCESS_PAGE_NUM;
            db("request.getRequestURI()="
                            +request.getRequestURI());
            db("getParameter(orderid)="+request.getParameter
                            ("orderId"));
            return pageNumber;
    }

    // ok, we have a page number, make sure it's good and
 within
    // range. (It always should be unless we made an error on
 one
    // of our HTML forms, or a user has modified the form)
    try
```

```
        {pageNumber = Integer.parseInt(pageString);
        if(pageNumber >= pageHandlerNames.length)
            pageNumber = NOFORM_PAGE_NUM;}
    catch(NumberFormatException e)
    {
        pageNumber = NOFORM_PAGE_NUM;
    }
    return pageNumber;
}
```

The final two methods in this class log errors. They were used in the debugging phase of the project.

```
private static void logError(String methodName,
                    String messageText,Exception e)
{
    ByteArrayOutputStream bout = new ByteArrayOutputStream();
    PrintStream p = new PrintStream(bout);

    db(methodName + ": " + messageText);
    if(e != null)
         {
             db(e + " " + e.getMessage());
             e.printStackTrace(p);
             db(bout.toString());
         }
}

protected static void db(String s)
{
    FileOutputStream out = null;
    DataOutputStream dout = null;
    try
         {
             out  = new FileOutputStream("debug.log",true);
             dout = new DataOutputStream(out);

             dout.writeBytes(s + "\r\n");

             dout.flush();
             dout.close();
```

```
            out.close();
            System.out.println(s);
        }
    catch(IOException e)
        {
            try
            {
                if(dout != null) dout.close();
                if(out != null) out.close();
            }
            catch(IOException x)
            {
            }
        }
     }
}
```

The Transaction class

The ESD servlet uses a `Transaction` object to store the details of a single transaction.

The `Transaction` class declares that it implements the `Serializable` interface so that the class can be saved and later reloaded.

```
class Transaction extends Object implements Serializable
{
```

The class has a fairly large set of member variables that store the details of the transaction. The first member variable is a `static Hashtable` that is available to all `Transaction` objects. This `Hashtable` contains all of the `Transaction` objects so that you can reach all of the other transactions if you have any transaction.

```
    // static transaction table shared with other threads
    static Hashtable transactions;

    String orderID;     // ID of the order
    long   recordCreated;// time this record was created
    boolean verified;   // order ID has been verified with
                        // Web order
    boolean accepted;   // if verified is true,
```

```
                    // accepted = true
                    // if order OK, false if
                    // order declined
long   downloadCount;// number of times this order has
                    // attempted download

// pricing information
int price;
int tax;
int total;
boolean chargeTax;

// license information:
String name;
String company;
String email;
String address;
String city;
String state;
String zip;
String country;
String phone;

// authorization information
String AuthCode;
String ccOID;
```

The constructor loads data into the `Transaction` object.

```
public Transaction(String name,
                String email,
                String address,
                String city,
                String company,
                String state,
                String country,
                String zip,
                String phone)
{
   orderID = null;
   recordCreated = System.currentTimeMillis();
```

```
        verified = false;
        accepted = false;
        downloadCount = 0;

        // pricing information (in cents)

         price = 9900;
         tax = 817;
         total = 10717;
         chargeTax = false;

        // license information:

         this.name = name;
         this.email = email;
         this.address = address;
         this.city = city;
         this.state = state;
         this.country = country;
         this.zip = zip;
         this.phone = phone;
         this.company = company;

        // authroization information

        AuthCode = "";
        ccOID = "";
    }
```

The `findTransaction` method takes an order ID and uses the static hashtable to return the corresponding `Transaction` object. The `synchronized` declaration insures that only one thread will try to access the hashtable at a time. (That isn't a problem if two threads are trying to read the hashtable, but might be if one thread is trying to write to the hashtable while another is reading.)

```
  public static synchronized Transaction findTransaction(
                                              String orderId)
  {
       Transaction t = (Transaction)
 transactions.get(orderId);
```

```
        return t;
    }
```

The `record` method, which is also synchronized, saves the transaction to disk. Since each transaction contains the hashtable of all transactions, saving one transaction saves all of them.

```
public synchronized void record() throws IOException
{
    transactions.put(orderID,this);
    WriteTransactions();
}
```

The `selectID` method returns an ID for a new transaction. This sample gets an ID in a way that is not secure; if you do this in a real servlet, you should use a way of creating an ID that is guaranteed to be unique and impossible to guess.

```
    public synchronized void selectId() throws IOException
    {
        File fOrderId = new File("order.id");
        RandomAccessFile inOrderId = new
                            RandomAccessFile(fOrderId,"rw");

        // Order IDs are assured uniqueness using a
        // counter on disk. Read and update the counter
        // creating it if needed.
        int id = 1;
        if(inOrderId.length() > 0)
        {
            id = inOrderId.readInt();
            inOrderId.seek(0);
        }

        inOrderId.writeInt(id+1);
        inOrderId.close();

        // Now add some data to the order ID to make it more
        // difficult to guess.
        long now = System.currentTimeMillis();
        long orderid =   ((((now & 0x0000000000000FFF) ^
```

```
                        0x0000000000000b2E) )  ) |
                        ( ((long)id) << 12) ;

        // ENCHANCE: to make the order ID unguessable you can
        // encrypt it with a fixed key at this stage.

        orderID =  Long.toString(orderid,46);
    }
```

The `ESDServlet.init` method calls `InializeTransactions` method to load any pre-existing transations. It tries to open three files: `transactions.current`, `transactions.new`, and `transactions.log`.

```
    public synchronized static void InitializeTransactions()
                                throws
                                    ServletException,
                                    IOException,
                                    OptionalDataException,
                                    ClassNotFoundException
    {
        File currentTransactions = new
                                File("transactions.current");
        File newTransactions = new File("transactions.new");
```

The `transactions.new` file is a temporary file that is supposed to be deleted, so if it exists already the servlet must have terminated abnormally.

```
        if(newTransactions.exists())
        {
            System.out.println("Transaction files in an
 inconsistent state");
            throw new ServletException("Transaction files in
 an inconsistent state");
        }

        // if the current transaction file exists, read it in
        if(currentTransactions.exists() )
        {
            FileInputStream fIn = new
                      FileInputStream(currentTransactions);
            ObjectInputStream in = new ObjectInputStream(fIn);
```

```
            transactions = (Hashtable) in.readObject();
            in.close();
            fIn.close();
        }
        else
        { // if there is no transaction file, start a new one
            transactions = new Hashtable();
        }
    }
```

The `ESDServlet.destroy` method calls the `WriteTransactions` method to save the transaction table to disk.

```
    public synchronized static void WriteTransactions()
                                    throws IOException
    {

        // Remove transactions more than 48 hours old
        Enumeration key = transactions.keys();
        while(key.hasMoreElements())
        {
            String keyName = (String) key.nextElement();
            Transaction t =
                      (Transaction)transactions.get(keyName);
            if(t.recordCreated + (48 * 3600 * 1000) <
                                System.currentTimeMillis())
            {
                transactions.remove(keyName);
            }

        }

        // write updated transactions to file
        File newTrans = new File("transactions.new");
        FileOutputStream fOut = new
                      FileOutputStream(newTrans);
        ObjectOutputStream out = new
                      ObjectOutputStream(fOut);
        out.writeObject(transactions);
        out.close();
        fOut.close();
```

```
        // rename files, and delete old ones
        // Note: failure during the execution of this section
        //  of code may leave files in an inconsistent state

        if(oldTrans.exists()) oldTrans.delete();
        if(currentTrans.exists())
                        currentTrans.renameTo(oldTrans);
        newTrans.renameTo(currentTrans);
    }
}
```

The PageData class

This class encapsulates the servlet's information about a page.

```
public class PageData
{
    HttpServletRequest  request;
    HttpServletResponse response;
    ServletConfig config;
    Transaction transaction;
    String messageText;

    public PageData( ServletConfig config,
                     HttpServletRequest request,
                     HttpServletResponse response)
    {
        this.config = config;
        this.request = request;
        this.response = response;
    }
}
```

The PageHandler class

`PageHandler` is a superclass for the page handlers that know how to handle the forms used in the ESD servlet.

```
public class PageHandler
{
```

The page handler has a field that stores the HTML code for this page. When the servlet tells the page handler to serve the page, the page handler first loads the page into this field. It then calls a method that can customize the page, and sends the customized code to the user.

```
public String pageHtml;
```

`PageHandler` then defines a constant that holds the base location for the ESD servlet's HTML pages. If you want to use `ESDServlet` as a basis for your own servlets, you need to change this value.

```
protected static final String PAGEBASE =    "private_html" +
                                            File.separator +
                                            "symantec" +
                                            File.separator;
```

The default constructor does nothing.

```
public PageHandler()
{
}
```

The `pageName` method exists to return the name, or page number, of this page. Since this is a superclass that won't be instantiated, this version returns nothing.

```
public String pageName()
    {
        return null;
    }
```

Next comes the `serve` method, which reads the HTML page, calls a method that can customize it, and writes the resulting page. Following that is the method that reads the HTML file. Subclasses don't need to override these methods.

```
protected void serve(Object pageData,
                     HttpServletResponse response)
                     throws IOException
{
   read();
   customize(pageData);
   write(response);
}
protected void read() throws IOException
{
```

```
    FileInputStream in = new FileInputStream(pageName());
    byte[] b = new byte[in.available()];
    in.read(b);
    in.close();
    pageHtml = new String(b);
}
```

Subclasses can override the `customize` method to alter the HTML based on input to the previous page.

```
protected void customize(Object pageData) throws IOException
{
}
```

The following method serves the HTML page to the user. Subclasses don't need to override this.

```
protected void write(HttpServletResponse response)
                                        throws IOException
{
    response.setContentType("text/html");
    response.setContentLength(pageHtml.length());
    ServletOutputStream out = response.getOutputStream();
    out.print(pageHtml);
}
```

Subclasses need to override the `validate` method so that it validates the data the user put in the form and returns a `PageData` object that contains either the page data or error information. The default version returns `null`.

```
protected Object validate(ServletConfig
 config,HttpServletRequest request,HttpServletResponse
 response) throws IOException
{
       return null;
}
```

The `nextPage` method needs to pass back the number for the next page that the user should see. The subclasses usually determine the next page number in `validate`, and `nextPage` just passes the number back. Thus, if information was invalid, the `validate` method simply sets the next page value so that the same page is served again, probably customized with error text.

```
public int nextPage()
```

```
{
   return 0;
}
```

The `insert` method replaces a special marker in the HTML file with some specified text. A page handler can call this from its `customize` method.

```
protected void insert(String marker,String text)
{

   String replaceThis = "<!--"+marker+"-->";
   int pos = pageHtml.indexOf(replaceThis);
   if(pos >= 0)
      {String newpageHtml = "";
      if(pos > 0) newpageHtml += pageHtml.substring(0,pos);
      newpageHtml += text;
      if(pos+replaceThis.length() < pageHtml.length()  )
         newpageHtml +=
            pageHtml.substring(pos+replaceThis.length());
         pageHtml = newpageHtml;
      }
}
```

The `getParameter` method returns the request parameter or, if there is no parameter, returns a default value.

```
protected String getParameter(HttpServletRequest request,
                    String name,String defaultValue)
{
   String value = request.getParameter(name);
   if(value != null) return value;
   return defaultValue;
}
```

Page handlers

The rest of the classes in the Servlet project are page handlers. They are all fairly similar. Here is the page handler for `enterinfo.html`:

```
public class PHEnterInfo extends PageHandler
{
```

This page handler adds two member variables: the constant `PAGENAME`, which has the path for the enterinfo page and the variable `pagenumbe` , which gives the number for the page that should be served next.

```
private final static String PAGENAME = PAGEBASE +
                                       "enterinfo.html";
int pagenumber = ESDServlet.VERIFYINFO_PAGE_NUM;
```

The constructor doesn't do anything.

```
public PHEnterInfo()
{
}
```

The `pageName` method is overridden to return the value of the `PAGENAME` constant. The `serve` method uses this to get the text of the HTML page.

```
public String pageName()
{
    return PAGENAME;
}
```

The `nextPage` method is overridden to return the value of the `pagenumber` variable. The servlet code uses this to create a page handler that will be used to serve the next page.

```
public int nextPage()
{
    return pagenumber;
}
```

The `validate` method checks the information entered in the page.

```
public Object validate( ServletConfig config,
                        HttpServletRequest request,
                        HttpServletResponse response)
{
```

First, the method creates a `PageData` object. This object will eventually store the information from the form or error text information.

```
      PageData pageData = new
 PageData(config,request,response);
      String errorText = "";
```

The method loads the information from the form.

```
      String name   = getParameter(request,"name","");
      String email  = getParameter(request,"email","");
      String address= getParameter(request,"address","");
```

```
String city   = getParameter(request,"city","");
String company= getParameter(request,"company","");
String state  = getParameter(request,"state","");
String country= getParameter(request,"country","");
String zip    = getParameter(request,"zip","");
String phone  = getParameter(request,"phone","");
```

Next, the method does some basic validity checks on the information and sets the error text appropriately if it finds a problem.

```
if(name.length() < 2)   errorText += "Please enter
your full name in the name field<br>";
if(email.length() < 5)  errorText +="Your e-mail
address is required.  (It will not be sold to
spammers)<br>";
if(address.length() < 2)errorText += "Please provide
your full mailing address<br>";
if(city.length() < 2)   errorText +="Please include a
city in your address<br>";
```

If there is any error text, then there was a problem with the information entered on the form. The method sets the `pagenumber` so that the enterinfo page will be served again, places the error text in the `pageData` object, and returns the `pageData` object.

```
if(errorText.length() > 0)
{
    pagenumber = ESDServlet.ENTERINFO_PAGE_NUM;
    pageData.messageText = errorText;
    return pageData;
}
```

If there was no problem found with the information on the form, then the method creates a new `Transaction` object using that information, stores the `Transaction` object in the `pageData` object, and returns the `pageData` object.

```
pageData.transaction = new Transaction(name,
                                       email,
                                       address,
                                       city,
                                       company,
                                       state,
                                       country,
                                       zip,
                                       phone);
```

```
            return pageData;
        }
    }
```

I N D E X