7

Using the JavaHelp API for Advanced Presentation Options

Throughout this book, you have viewed HelpSets in *standalone* mode, using the HelpSet Viewer utility. However, you may also need to create online help systems that work with Java applications. This chapter discusses a number of advanced presentation options—ways to tie the help system to the application:

- The TypeFacer application
- Invoking the help system with a button, menu item, or help key
- Applying screen-level context-sensitive help
- Applying field-level context-sensitive help
- Embedding help into the application

To show you the coding effort involved, this chapter works through a progressive example, showing how to connect JavaHelp to a Java application. The material assumes you are already familiar with Java programming.

I don't discuss how to create new Java components for displaying custom information in the help topics themselves. For example, you could create a multimedia object that plays movies or audio clips, and then insert multimedia clips in your help topics.

While good practice dictates that you should plan online help when planning the application for which it is written, many times the application exists already and the help system is added later. Since this situation is an reality, that's where I'll start. Fortunately, adding JavaHelp after an application is developed is not a monumental task.

The TypeFacer Application

I will start with a simple application called Typeface Tester (TypeFacer for short) to show you all the important pieces of commercial applications without confusing you with the thousands of pages of code that go into commercial applications. You can download the code for this application and its HelpSet from "Examples" on this book's web site. To execute the examples in this section:

1. Create a new directory, called *TypeFacer*, in which you'll be editing, compiling, and running Java programs.
2. Download file *typefacer.zip* from this book's web site, placing it in the *TypeFacer* directory.
3. Unpack this ZIP file, preserving the subdirectory structure. You can use any ZIP utility, but it's easiest just to use the JAR utility:

   ```
   jar -xvf typefacer.zip
   ```

   The ZIP file contains a Java source file, *TypeFacer.java*. It also contains a subdirectory hierarchy, *TFhelp*, which contains the TypeFacer application's HelpSet: HelpSet data files, navigation files, topic files, and image files.
4. Compile the TypeFacer application:

   ```
   javac TypeFacer.java
   ```
5. Run the application:

   ```
   java TypeFacer
   ```

   The TypeFacer window is shown in Figure 7-1.

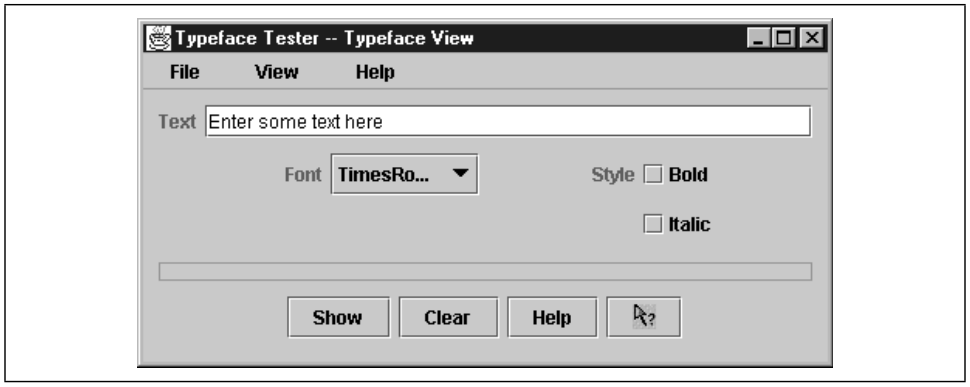

Figure 7-1. Typeface Tester application

TypeFacer enables you to select a variety of fonts, typeface styles, and colors. You can then type some sample text to see how it would look in a particular style.

The following skeleton code shows the outline of the application, before we start adding JavaHelp-related code. (See Appendix D, *TypeFacer.java Source Listing*, for a complete listing of the application, including the JavaHelp-related code.) The application defines two screens, **Typeface View** and **Color View**, which are managed by the Java Swing `CardLayout` manager. Swing GUI components implement a menu bar, an input text field, selection buttons (for the font face, style and color), and control buttons (for showing and clearing the display). Anonymous inner classes handle the events coming from the various buttons and menu items.

Since the objective here is to add help to an application, not to learn Java programming, let's move on. If you want to know more about developing graphical Java applications, refer to *Java Swing*, by Robert Eckstein, Marc Loy, and Dave Wood (O'Reilly & Associates).

```
/*
 * TypeFacer.java
 * A simple application for styling sample text.
 */

// imports of Java class libraries
...

public class TypeFacer extends JFrame {

// data items
   // screen components
   // menu components
   // CardLayout manager setup
   // fonts and colors
   // combo box choices and titles

// constructor method

  public TypeFacer() {

  // create and size a JFrame; set up content pane

  // set up top-most panel containing text-input field

    JLabel inputLabel = new JLabel("Text");
    inputField = new JTextField("Enter some text here", 30);

  // set up middle panel, in which two cards will
  // be displayed: typefCard and colorCard

    JPanel typefCard = new JPanel(new GridLayout(2,4,5,5));
    JPanel colorCard = new JPanel(new GridLayout(2,4,5,5));
```

```
// TypeFace card: create components

  JLabel fontLabel = new JLabel("Font", JLabel.RIGHT);
  fontChoice = new JComboBox(fontList);

  JLabel styleLabel = new JLabel("Style", JLabel.RIGHT);
  boldBox = new JCheckBox("Bold");
  italicBox = new JCheckBox("Italic");

// Colors card: create components

  JLabel foreLabel = new JLabel("Foreground", JLabel.RIGHT);
  foreChoice = new JComboBox(colorList);
  foreChoice.setSelectedIndex(0);  // initialize to "black"

  JLabel backLabel = new JLabel("Background", JLabel.RIGHT);
  backChoice = new JComboBox(colorList);
  backChoice.setSelectedIndex(5);  // initialize to "white"

// set up styled output panel

  displayField = new JTextField(40);
  displayField.setEditable(false);
  displayField.setFont(new Font("TimesRoman", Font.PLAIN, 16));
  displayField.setHorizontalAlignment(SwingConstants.CENTER);

// set up button panel

  showButton = new JButton("Show");
  clearButton = new JButton("Clear");

// set up menu structure

  fileMenu = new JMenu("File");
  exitItem = new JMenuItem("Exit");
  viewMenu = new JMenu("View");
  typeItem = new JMenuItem("Typefaces");
  colorItem = new JMenuItem("Colors");

  JMenuBar menuBar = new JMenuBar();
  menuBar.add(fileMenu);
  menuBar.add(viewMenu);

// fill the fonts and colors hashtables

// activate the buttons

  showButton.addActionListener(new ActionListener() {
     ...
```

```
    });
    ...

  // activate the menu items

    exitItem.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {System.exit(0);}
    });
    ...
}

// main program: instantiate a TypeFacer object

  public static void main(String args[]) {
    (new TypeFacer()).setVisible(true);
  }
}
```

To save space, the code listings later in this chapter show only the lines that must be added or revised in order to implement JavaHelp features.

Invoking Help with a Button, Menu Item, or Key

The simplest online help addition to any application is a menu item that starts the help system. Typically, a **Help** menu item is not context-sensitive: the help system always opens with the default help topic specified in the HelpSet file. The user can manually navigate through the TOC or search through the index or word-search index for other help topics.

Other standard ways to start the help system include clicking a **Help** button and pressing a **Help** key (by custom, function key F1). It's up to you to decide which mechanism(s) to use. Follow these steps to add all three help activators to the TypeFacer example: menu item, button, and function key. Instead of writing new lines of code, you just need to uncomment (that is, remove the comment indicators from) lines that are already there. For this revision of the program, all the lines to be uncommented start with `//#1`. Use your text editor's search command to locate these lines.

1. Import the Java class library for JavaHelp:

   ```
   import javax.help.*;
   ```

2. Create new instance variables for HelpSet-level classes, and for the Help menu item and Help button:

   ```
   HelpSet hs;
   HelpBroker hb;
   ```

```
...
JButton    helpButton;
 ...
JMenu helpMenu;
JMenuItem helpItem;
```

3. Create the Help menu item and button objects themselves:

```
helpButton = new JButton("Help");
 ...
buttonPanel.add(helpButton);
 ...
helpMenu = new JMenu("Help");
helpItem = new JMenuItem("Contents...");
helpMenu.add(helpItem);
...
menuBar.add(helpMenu);
```

4. Activate the Help menu item and button objects, using a convenience method from the JavaHelp API's `CSH` class to create an `ActionListener`. Programmers should familiarize themselves with this class. It contains many static helper methods, such as the `DisplayHelpFromSource()` method used here.

```
// activate the Help menu item and Help button

ActionListener helper = new CSH.DisplayHelpFromSource(hb);
helpItem.addActionListener(helper);
helpButton.addActionListener(helper);
```

5. Instantiate the HelpSet as a Java class, and create an associated *HelpSet broker* object (see the following description). Enable the F1 function key, specifying the topic to be displayed. (This call doesn't access the HelpSet's default topic.)

```
// open HelpSet, send console message
// hardcoded location: "HelpSet.hs" in "TFhelp" subdirectory

try {
  URL hsURL = new URL((new File(".")).toURL(), "TFhelp/HelpSet.hs");
  hs = new HelpSet(null, hsURL);
  System.out.println("Found help set at " + hsURL);
}
catch (Exception ee) {
  System.out.println("HelpSet not found");
  System.exit(0);
}

// create HelpBroker from HelpSet
hb = hs.createHelpBroker();

// enable function key F1
```

```
hb.enableHelpKey(getRootPane(), "overview", hs);
```

6. Compile the revised TypeFacer source file:

   ```
   javac TypeFacer.java
   ```

   The application now relies on JavaHelp support. Accordingly, the compilation will fail if you haven't placed the JavaHelp JAR file, *jh.jar*, on your Java class path.

7. Run the application:

   ```
   java TypeFacer
   ```

8. Figure 7-2 shows the resulting Help menu.

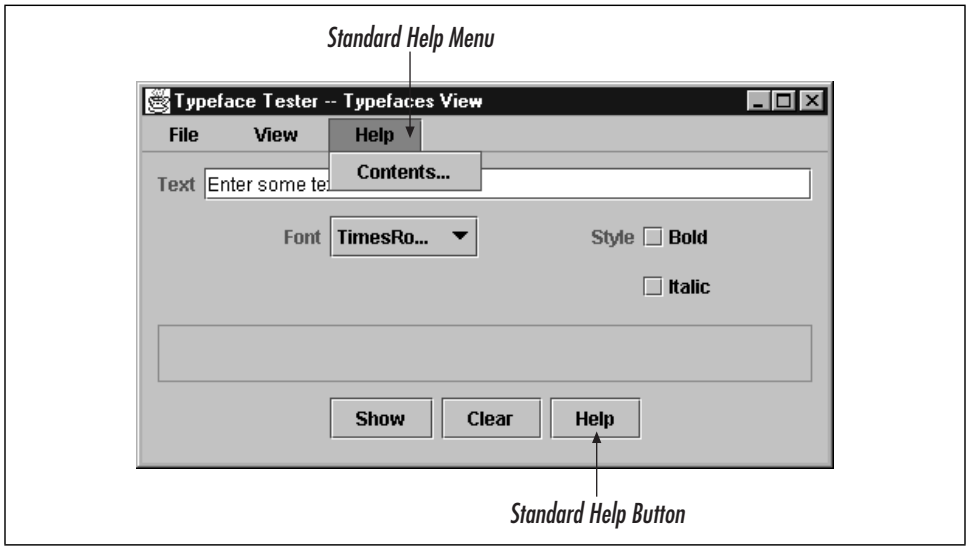

Figure 7-2. A help menu and help button

The code in Step 5 specifies the HelpSet from which to instantiate a Java `HelpSet` object. In this example, the HelpSet is stored in file *HelpSet.hs*, located in a subdirectory, *TFhelp*, of the TypeFacer application directory:

```
URL hsURL = new URL((new File(".")).toURL(), "TFhelp/HelpSet.hs");
```

Alternatively, you could retrieve the HelpSet from a web server:

```
URL hsURL = new URL("http://your.server.com/TFhelp/HelpSet.hs");
```

Or the HelpSet might be encapsulated in a local or remote JAR file:

```
URL("jar:http://your.server.com/jars/TFhelp.jar!/HelpSet.hs");
```

(For more on JAR files, see Chapter 8, *Deploying the Help System to Your Users*.)

The code creates a HelpSet broker (`HelpBroker`) object for the HelpSet. The broker handles communications between the application and the help system. JavaHelp programmers should get to know the broker interface well. It handles all the retrieval and manipulation of HelpSet properties (all of the `get()` and `set()` methods are here). You can find out what entry is currently being displayed or you can enable help for a particular component. While the example in this section doesn't use the HelpBroker extensively, you'll notice that you must pass it as an argument to many of the other parts in our application. All those other objects rely on the HelpBroker to manage the help system. Essentially, this interface is how your application plays with the HelpSets you create.

Using Screen-Level Context-Sensitive Help

You should now have a TypeFacer application in which the user can open the HelpSet Viewer to a default help topic. You can enhance the help system by using *screen-level context-sensitive help*. You need to program the help system to know the user's current location within the application. The help system can then display a relevant topic when the user calls for help.

The TypeFacer application has two screens through which the user sets text and background properties. Figure 7-3 shows these Typeface View and Color View screens.

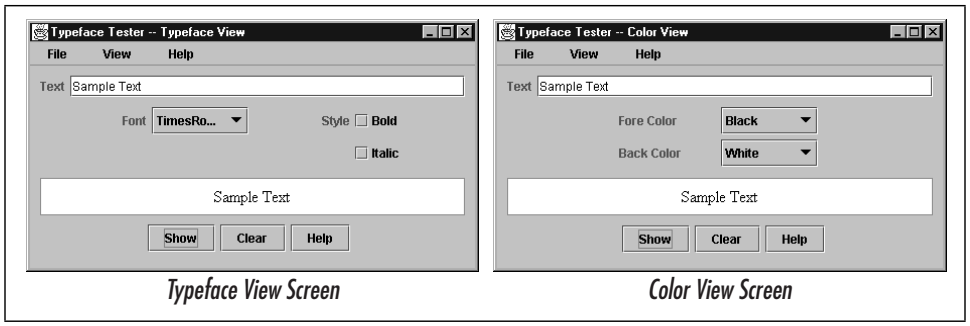

Figure 7-3. Typeface Tester's two screens

Other applications might have similar modes (for example, "edit mode" and "outline mode") or might have separate screens for different application tasks. Either way, the active mode or screen serves as a good indicator of which help topics the user might need. Rather than starting with the HelpSet's default help topic, you can go directly to the help topic for the active mode or screen.

Programming Screen-Level Context-Sensitive Help

Programming context-sensitive help is straightforward, using the JavaHelp API's `CSH` (context-sensitive help) class. You don't change how the HelpSet is loaded; you just need to connect a help topic to each combination of a screen context (Typefaces screen or Colors screen) and a help activator (menu item, button, or function key).

Use the following steps to make these specifications. For this revision of the program, all the lines to be uncommented start with `//#2`.

1. Update the `ActionListener` for the **Typefaces** menu item:

```
// configure function key F1, help button, help menu item
CSH.setHelpIDString(TypeFacer.this.getRootPane(), "typefaces");
CSH.setHelpIDString(helpItem, "typefaces");
CSH.setHelpIDString(helpButton, "typefaces");
```

2. Update the `ActionListener` for the **Colors** menu item:

```
// configure function key F1, help button, help menu item
CSH.setHelpIDString(TypeFacer.this.getRootPane(), "colors");
CSH.setHelpIDString(helpItem, "colors");
CSH.setHelpIDString(helpButton, "colors");
```

3. Compile (`javac`) and run (`java`) the revised TypeFacer application.

The `CSH.setHelpIDString()` method assigns a particular help topic to a user-interface component—in this example, to an entire screen (that is, to function key F1) to a menu item, or to a button. This method takes two arguments: the component and the help topic's map ID, as specified in the map file.

Now, when users are at the Typeface View screen, they can view the help topic with map ID `typefaces` by clicking the **Help** button, or by clicking the **Help** menu, or by pressing the F1 function key (see Figure 7-4).

Similarly, when users are at the Color View screen, they see the help topic in Figure 7-5.

Keeping an Overview Help Topic

You may have noticed a slight bug in the implementation of screen-level context-sensitive help: now that the menu item **Contents** is context-sensitive, it doesn't provide access to the HelpSet's overview topic. Typically, a help menu with multiple help items meets this need. Figure 7-6 shows a screen in which the user can access the overview help topic or a help topic for the current screen.

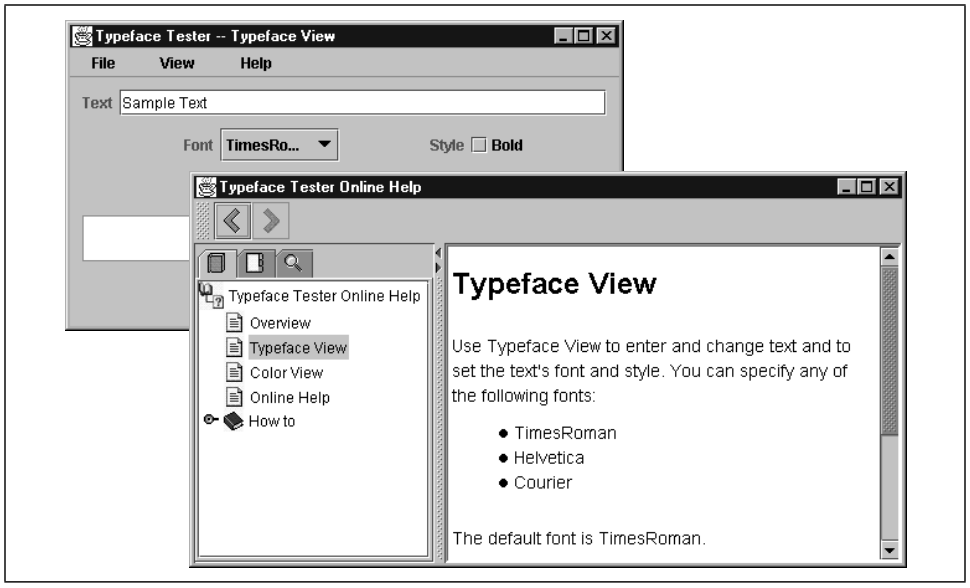

Figure 7-4. Context-sensitive help for the Typeface View screen

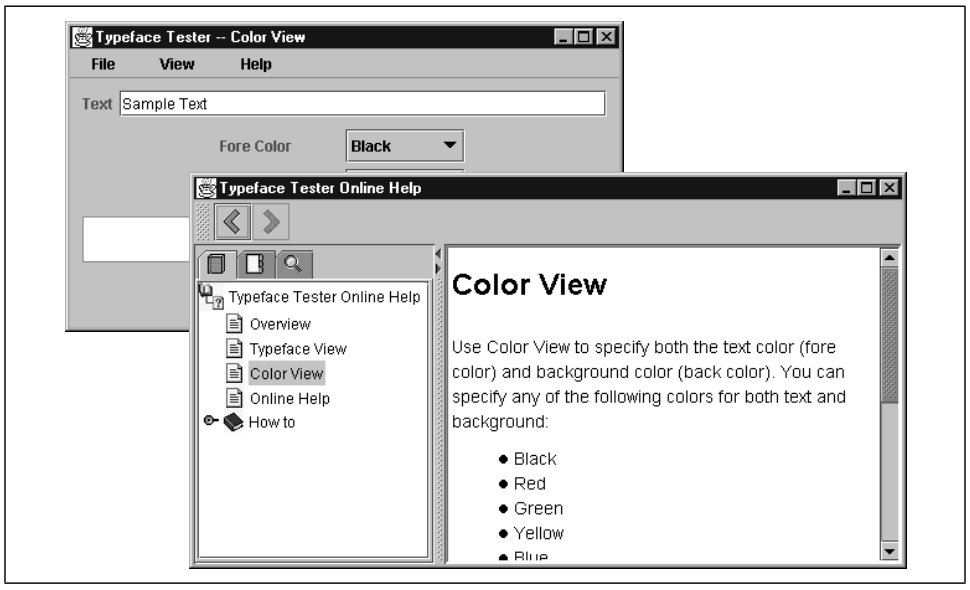

Figure 7-5. Context-sensitive help for the Color View screen

Use the following steps to set up the **Contents** menu item to launch the help system, loading the HelpSet and displaying the `overview` topic. For this revision of the program, all the lines to be uncommented start with `//#3`.

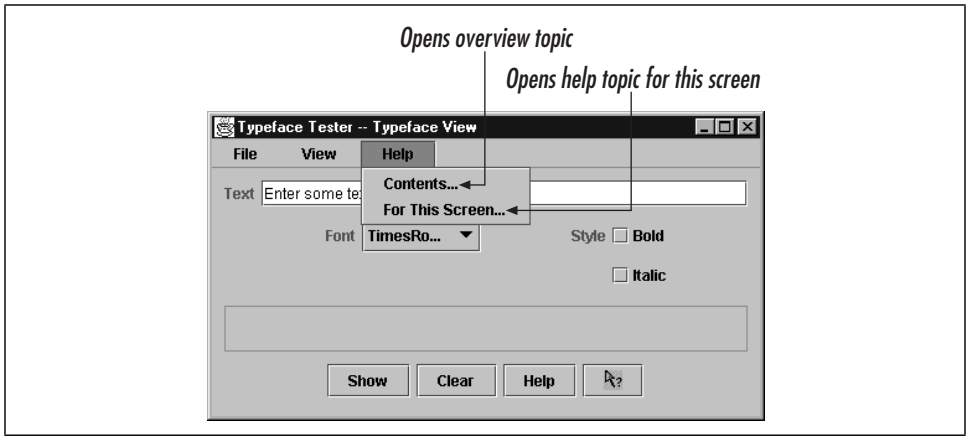

Figure 7-6. Offering both overview and context-sensitive help

1. Create a new menu item, set its associated help topic, and activate it with the same `ActionListener` you used for the **Help** button:

```
JMenuItem helpItemTOC;
 ...
helpItemTOC = new JMenuItem("Contents");
helpMenu.add(helpItemTOC);
CSH.setHelpIDString(helpItemTOC, "overview";
...
helpItemTOC.addActionListener(helper);
```

2. Change the wording of the existing menu item (so that you don't have two menu "Contents" items!):

```
helpItem = new JMenuItem("Contents...");
 ... change to
helpItem = new JMenuItem("For This Screen...");
```

3. Compile (*javac*) and run (*java*) the revised TypeFacer application.

Now users have the option as to which way they want to open the HelpSet: at an overview topic or at a topic for the current context.

Using Field-Level Context-Sensitive Help

You can take context-sensitivity one step further and implement *field-level help* (sometimes called What's This? help). In this scheme, the user clicks a button (or selects a menu item) that causes the mouse pointer to change—perhaps to a question mark. The user clicks a control, such as a button or selection box, and the application displays online help specific to that control.

The TypeFacer application consists of buttons, boxes, and text-display areas, all of which are potential targets for field-level help. For example, a user might not

know what the **Show** button does—a situation perfect for using field-level help. Figure 7-7 shows the help system after a user accesses field-level help for the **Show** button.

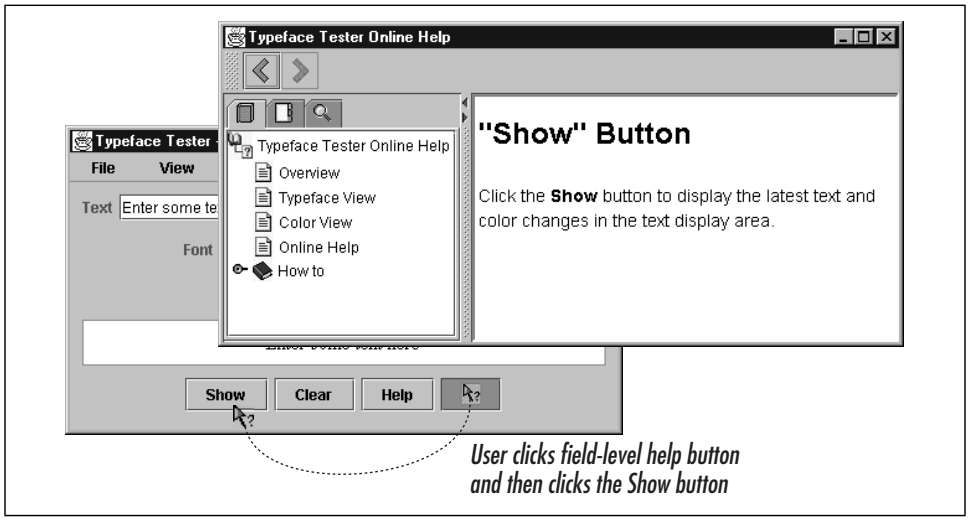

Figure 7-7. Field-level help

Programming Field-Level Context-Sensitive Help

Field-level context-sensitive help works like screen-level context-sensitive help. You use the same `setHelpIDString()` method as for screen-level help. Using this method, you provide a map ID for every component in the application, not just its several usage modes or screens.

Use the following steps to add field-level context-sensitive help to the TypeFacer application. For this revision of the program, all the lines to be uncommented start with `//#4`.

1. Associate a help topic with each user-interface component:

```
// assign map IDs for field-level context-sensitive help

CSH.setHelpIDString(inputField, "text");
CSH.setHelpIDString(fontChoice, "font");
CSH.setHelpIDString(boldBox, "bold");
CSH.setHelpIDString(italicBox, "italic");
CSH.setHelpIDString(showButton, "view");
CSH.setHelpIDString(clearButton, "clear");
CSH.setHelpIDString(helpButton, "help");
CSH.setHelpIDString(qButton, "whats_this");
CSH.setHelpIDString(displayField, "text_display");
CSH.setHelpIDString(foreChoice, "fore_color");
```

```
CSH.setHelpIDString(backChoice, "back_color");
```

2. Create a button for field-level help. (Make your own *help.gif* image or download the one at this book's web site. In any case, place the image file in the *TypeFacer* directory.)

```
JButton qButton;
 ...
qButton = new JButton(new ImageIcon("help.gif"));
...
buttonPanel.add(qButton);
```

3. Enable the field-level help button, using another convenience method in the `CSH` class:

```
qButton.addActionListener(new CSH.DisplayHelpAfterTracking(hb));
```

4. Compile (*javac*) and run (*java*) the revised TypeFacer application.

The `DisplayHelpAfterTracking()` method creates a handler that tracks the mouse movement after the user clicks the field-level help button. When the user selects a user-interface component, the method activates the help system, displaying the topic associated with the component. And all this takes just a single line of code!

If your application needs more control than this handler provides, look at the source code for the `CSH` class itself. It defines an inner class for supporting field-level help, which can serve as a good starting point for writing your own handler.

Embedding Help into the Application

To maximize the availability of help information, consider embedding online help into your application. Rather than running the HelpSet Viewer in a separate window (and a separate operating system process), you can place the JavaHelp content pane and navigation pane directly into your application window. Chapter 1, *Understanding JavaHelp*, provides a conceptual explanation of embedded help in more detail.

When embedding help, you can retrieve and arrange JavaHelp components as you do text areas and buttons, which gives you the greatest amount of control over the JavaHelp system. However, it can also increase programming requirements, depending on how responsive to user actions the help system must be.

Understanding Embedded Help

The JavaHelp API contains methods for handling the separate parts of the HelpSet Viewer individually:

- The HelpSet Viewer's content pane displays the help topics (HTML files). In the JavaHelp API, the content pane is known as a *content viewer*.
- The navigation pane (typically) includes three tabs, each providing a way to find and display help topics. In the JavaHelp API, these tabbed navigation components are known as *navigators*.

Thus, you can associate a number of different navigators with a single viewer. This modularity enables you to create customized (and often impressive) help viewers for your application. You can create custom navigators and attach them to existing viewers.

Once you decide which JavaHelp components you want to use, you can retrieve them and then add them to standard Java containers. Figure 7-8 shows an example of embedded help in the TypeFacer application.

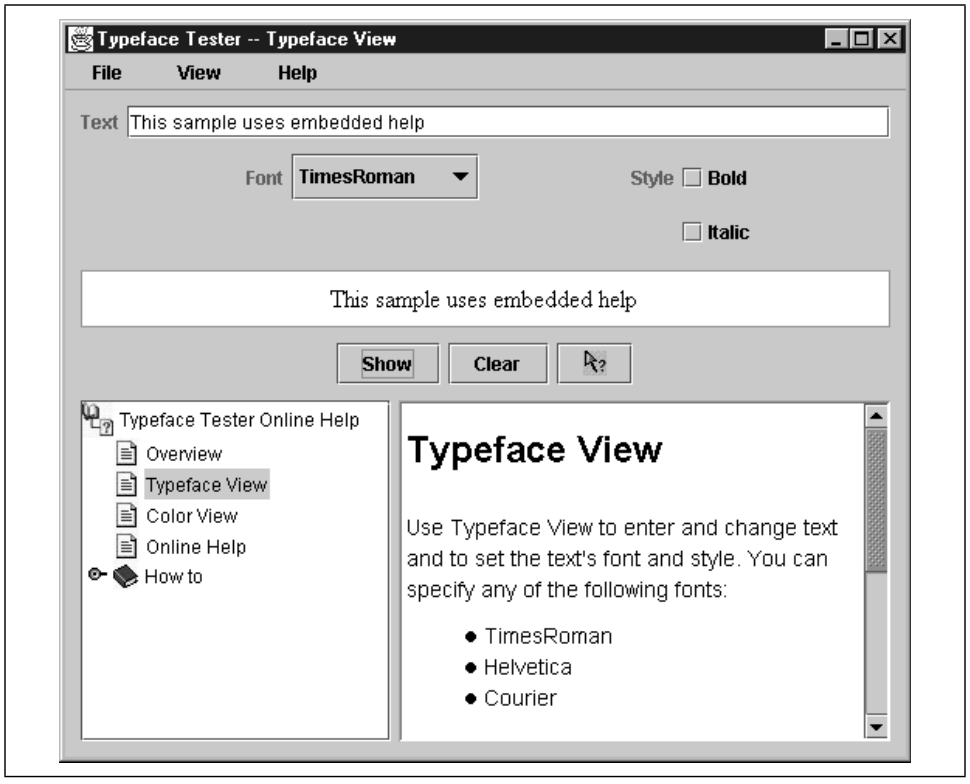

Figure 7-8. Embedded help

Programming Embedded Help

Use the following steps to implement embedded help. For this revision of the program, all the lines to be uncommented start with `//#5`.

1. Create instance variables for the embedded-help user-interface components:

```
// Instance variable for the Embedded Help pieces
JPanel helpPanel;
JHelpNavigator nav;
JHelpContentViewer viewer;
```

2. Change the size of the TypeFacer window, to accommodate the embedded help panel:

```
setSize(500, 500);
```

3. Add a panel that contains a viewer subpanel and a navigator (TOC) subpanel:

```
// create an embedded help panel
helpPanel = new JPanel(new GridLayout(1,2,5,5));
//helpPanel.add(displayPanel);

// add a content viewer
viewer = new JHelpContentViewer(hs);
viewer.setPreferredSize(new Dimension(200,220));

// add a navigator with a table of contents view
nav = (JHelpNavigator)
  hs.getNavigatorView("TOC").createNavigator(viewer.getModel());
nav.setPreferredSize(new Dimension(200,220));

// add the components to the layout
helpPanel.add(nav);
helpPanel.add(viewer);
contentPane.add(helpPanel);
```

4. Compile (*javac*) and run (*java*) the revised TypeFacer application.

The `JHelpNavigator` and `JHelpContentViewer` components you use here are the same components the HelpSet Viewer uses. They are part of the `javax.help` package, so no extra imports are required. If your HelpSet's topic are extensively cross-linked, the content viewer might be all you need. Most likely, though, you will want to set up one or more navigators in addition to the content viewer.

The `getNavigatorView()` method provides access to the views defined in your HelpSet file; you specify the view by its name: `TOC`, `Index`, or `Search`. This method returns an instance of the `NavigatorView` class, which is the base class for building navigator objects. You can use this class to create a navigator user-interface component for the view. The `createNavigator()` method instantiates such a navigator object; the `viewer.getModel()` argument is what ties the new naviga-

tor to the content viewer you just created. Now when the user selects an entry from the TOC, the viewer displays the corresponding help topic.

Programmatic Control of the Viewer Topic

In this section, you'll add context sensitivity to the TypeFacer application's embedded help:

- You'll make the embedded help system initially display the topic (`typefaces`) that matches the application's initial screen (the Typefaces screen).
- You'll have embedded help automatically switch to the appropriate help topic as the user switches between the Typefaces and Colors screens.

The technique is similar to the one used in the previous section "Programming Screen-Level Context-Sensitive Help." I don't recommend implementing context-sensitivity at the field level instead of the screen level. The increased "flashing" can be disconcerting for the user.

You'll also enhance the application to enable users to hide the embedded help panel, should it become distracting.

Implementing context sensitivity

Use the following steps to set TypeFacer's initial embedded-help topic and automatically change the help topic as the user switches between TypeFacer's two screens. For this revision of the program, all the lines to be uncommented start with `//#6`.

1. Set the initial help topic:

```
viewer.setCurrentID("typefaces");
```

2. Establish screen-level context sensitivity by updating the event handlers for the **Typefaces** and **Colors** menu items:

```
// update the embedded help content panel
viewer.setCurrentID("typefaces");
 ...
// update the embedded help content panel
viewer.setCurrentID("colors");
```

3. Compile (*javac*) and run (*java*) the revised TypeFacer application.

Updating an event handler is really the same as initializing the embedded content viewer: you use the `setCurrentID()` method to specify a map ID. Use this programmatic control of the viewer in event handlers or in response to other actions such as clicking a **Help** button or jumping to a bookmark.

NOTE This implementation has a potentially undesirable side-effect. When the user switches to a different screen in the application, the embedded help topic changes automatically, but the user might still need the information in the topic that just disappeared. You might improve this behavior by having the application query the content viewer for the currently displayed help topic and, based on that answer, decide whether or not to switch topics.

This implementation of screen-level embedded help duplicates the screen-level context-sensitive help you programmed earlier in this chapter. Thus, you might want to eliminate the **Help → For This Screen** menu item. On the other hand, it makes sense to maintain the interface to the (nonembedded, noncontext-sensitive) HelpSet Viewer, accessed through the **Help → Contents** menu item. It also makes sense to continue using the standard HelpSet Viewer to display field-level help. If you want the field-level help to appear in the embedded help viewer, you must write a custom `HelpBroker` that references your components. This task isn't trivial, but it is certainly not impossible.

Hiding embedded help

Hiding and displaying components at runtime is not really a JavaHelp topic, but it's important to embedded help systems. The idea is simple: provide a toggle for users to specify whether or not they want to see embedded help. If they don't want embedded help, remove the components. If they do want embedded help, put the components back.

Use the following steps to implement toggling of the embedded help display. For this revision of the program, all the lines to be uncommented start with `//#7`.

1. Add an item to the **Help** menu and initialize it as a "hide" toggle:

```
JMenuItem embeddedItem;
 ...
embeddedItem = new JMenuItem("Hide Embedded Help");
helpMenu.add(embeddedItem);
embeddedItem.setActionCommand("hide");
```

2. Activate the menu item, implementing the toggle functionality:

```
// activate the "Embedded Help" toggle menu item

embeddedItem.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    if (ae.getActionCommand().equals("hide")) {
      helpPanel.remove(nav);
      helpPanel.remove(viewer);
      helpPanel.validate();
      TypeFacer.this.setSize(500,250);
```

```
        embeddedItem.setText("Show Embedded Help");
        embeddedItem.setActionCommand("show");
      }
      else {
        helpPanel.add(nav);
        helpPanel.add(viewer);
        helpPanel.validate();
        TypeFacer.this.setSize(500,500);
        embeddedItem.setText("Hide Embedded Help");
        embeddedItem.setActionCommand("hide");
      }
    }
});
```

3. Compile (*javac*) and run (*java*) the revised TypeFacer application.

The `ActionListener` first determines whether it needs to hide or show the embedded help panel. It adds or removes the content viewer and navigator components, then resizes the TypeFacer window (so that the layout manager doesn't leave a gaping hole or make the remaining components oddly shaped). Finally, it implements the toggle functionality by changing the wording of the menu item and changing its state with `setActionCommand()`.

The only difference you see in a network version of the previous examples is in the code that finds the HelpSet. For example, if you install your online help on a computer with a web server, you could access the help through a regular URL, as shown in the following code:

```
// Variables to store our HelpSet and HelpBroker:
HelpSet hs;
HelpBroker hb;

// in the constructor, you add this code toward the top
// the assumption here is that the help files exist under the "jh"
// directory that resides in the same directory as the application.
try {
  URL hsURL = new URL("http://your.server.com/jh/HelpSet.hs");
  hs = new HelpSet(null, hsURL);
} catch (Exception ee) {
  System.out.println("HelpSet not found");
  return;
}
hb = hs.createHelpBroker();
```

If you have all your help bundled into a JAR file and wish to run a Java 2 SDK application, use the following JAR URL syntax:

```
URL hsURL = new URL("jar:http://your.server.com/jars/help.jar!/HelpSet.hs");
```