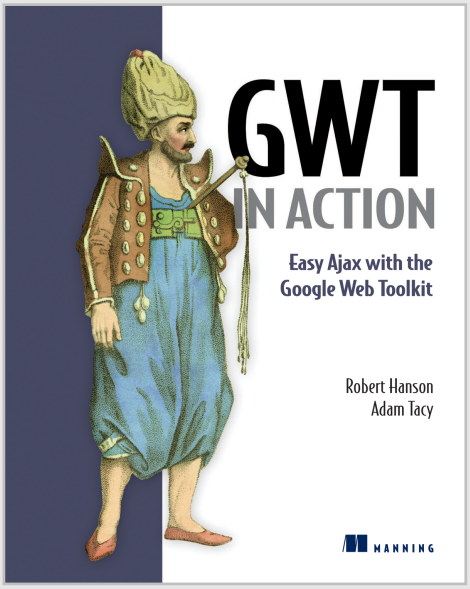

CHAPTER 4:

Working with Widgets

Excerpt From *GWT in Action*
ISBN 1-933988-23-1
©2007 Manning Publications
All rights reserved
www.manning.com

4.1 The standard GWT widgets

The standard GWT distribution comes with a wide range of widgets for use in your applications. These widgets cover the types of areas you would expect: buttons, text boxes, and so forth. There are, however, some widgets you might expect to see that are missing—things like progress bars and sliders (though we'll build one of those missing widgets later in chapter 7).

Within the set of widgets, the designers of GWT have implemented a strong hierarchy of Java classes in order to provide an element of consistency across widgets where that consistency naturally exists. Take the `TextBox`, `TextArea`, and `PasswordTextBox` widgets; it is not unreasonable to expect them to share certain properties. GWT recognizes this and captures the common properties in a `TextBoxBase` class, which these three widgets inherit. To get a snapshot of this hierarchy, cast your eye over figure 4.1.

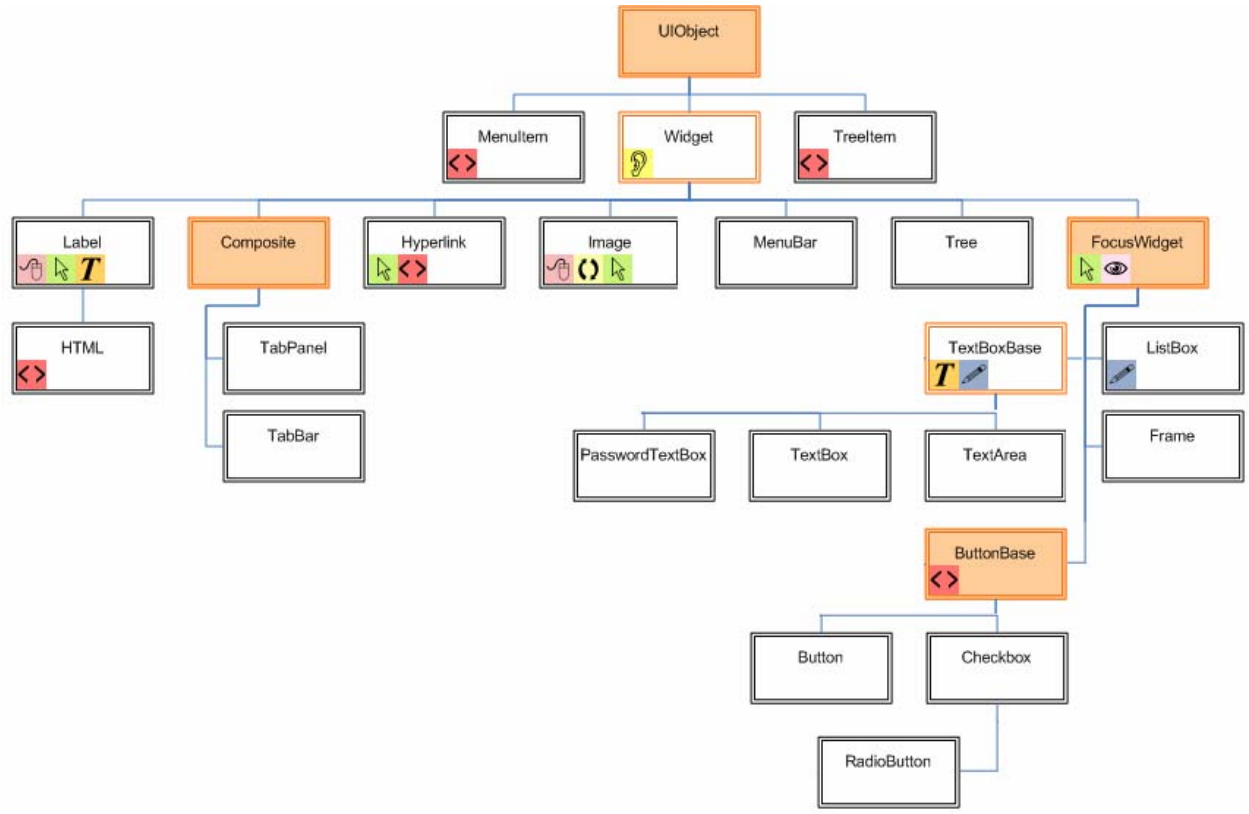

Figure 4.1 The GWT widget class hierarchy, indicating the types of event listeners that can be registered against each widget

You can see in this hierarchy that all widgets ultimately inherit from the `UIObject` class, which contains a number of essential housekeeping and property aspects. Within the `UIObject` class you will find the `setElement()` method, which we discussed previously when it is used to set the physical link between the widget's Java object and DOM views. Subclasses of `UIObject` must call this method as the *first* thing they do before any other methods are called to ensure that the link to a browser element is established.

> **Note:** All GWT widgets inherit from the `UIObject` class, which provides a common set of methods and attributes for all widgets, including setting size, visibility, and style names, as well as providing the link between the Java and DOM representations.

We won't go through all the methods in the `UIObject` class, but we will highlight the typical functionality you can expect all widgets to inherit. The `UIObject` class allows access to a wide range of DOM functionality without you having to access the DOM directly. For example, it is possible to set the height of a GWT `UIObject` using the `setHeight()` method, which in turn uses the `setStyleAttribute()` method from the DOM class.

```
public void setHeight(String height) {
   DOM.setStyleAttribute(element, "height", height);
}
```

The other methods written in this style include the ability to set the width, title (what is displayed when a mouse hovers over an element), and both width and height at the same time through the `setSize()` method. All of these methods take Strings as parameters, such as `setSize("100px","200px")`. The `setPixelSize()` method allows integers, such as `setPixelSize(100,200)`. Although these methods for setting stylistic attributes are available, we recommend that styling generally be performed by using Cascading Style Sheets (CSS).

After `UIObject`, all widgets, except `TreeItem` and `MenuItem`, must inherit from the `Widget` class, which provides widgets with their "widget-ness," including the methods that are called when a widget is attached or detached from a panel. This class also contains the default implementation of the `onBrowserEvent()` method, which allows a widget to manage any events that it has sunk (you will see this in action in chapter 6).

In the next section we'll look briefly at the widgets you get for free with the GWT distribution as well as how we'll be using them in the Dashboard application. Figure 4.2 summarizes the widgets.

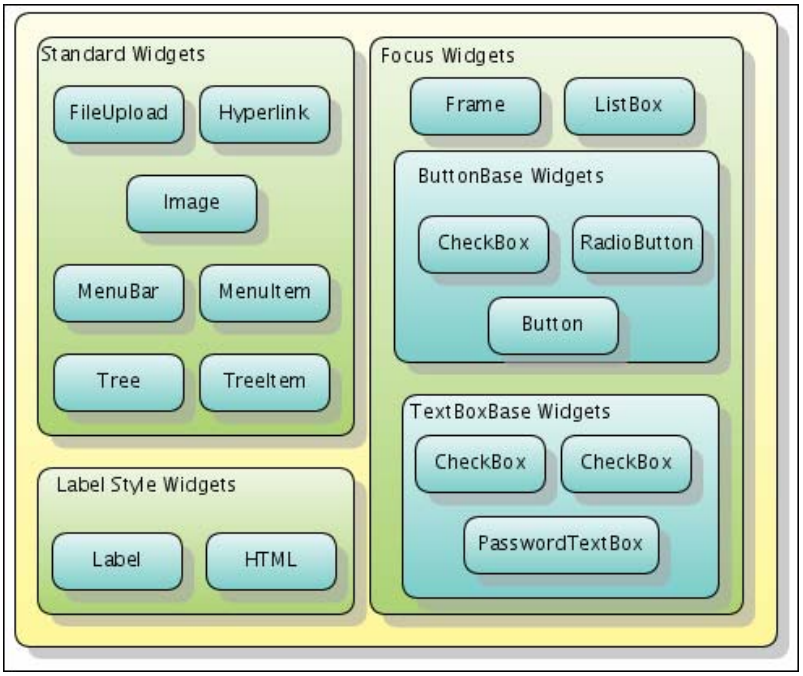

Figure 4.2 Summary of widgets included within GWT

As we discuss the widgets during this chapter, we'll try to show some simple code demonstrating their use and point out where in the Dashboard application we use them. This code might start looking a little alien in some places but don't worry; we haven't introduced most of the concepts used in the code yet—they'll be revealed in the next few chapters—but in most cases you should be able to see what is happening.

Bear in mind a couple of principles as you read these next few sections. First, this is not intended to be a complete walkthrough of the GWT API for widgets; to do that would border on the excessively boring. Instead we try to show some of the key methods and pitfalls, as well as where we have used the methods in the Dashboard application, so that you can look in the code yourself. In general, where we define a `getter` method, such as `getText()`, the GWT API usually provides the appropriate `setter` method as well, such as `setText()`. Finally, to keep focused, we often do not include names or number of parameters for a method, unless it is really necessary; this allows us to write a book focused on what can be done without getting bogged down in details—the online GWT API reference (`http://code.google.com/webtoolkit/documentation/gwt.html`) is an invaluable source of help for the details, as is the use of a good IDE.

We'll break our discussion of widgets down into the following five main categories of widgets that were shown in figure 4.2: Standard, Label, Focus, Button Base, and Text Box Base widgets. Let's start by looking at the class of widgets we have called *basic widgets*.

4.1.1 Interacting with the basic widgets

We'll define the basic widgets as those that inherit directly from the `Widget` class and have nothing else getting in the way. There are five of these widgets, which we briefly consider next in the context of the Dashboard application that we'll be building. Sometimes, however, we show a code sample in isolation to emphasize a particular point or property.

Uploading files with the FileUpload widget

The `FileUpload` widget (shown in figure 4.3) acts as a GWT wrapper to the standard browser `FileUpload` text box—the one that you need to use when the user wants to upload a file from their machine to your server.

Figure 4.3 The FileUpload widget

You must remember that this is only the client-side component—clicking the Browse button allows users to select a file on their own computer that presents the standard File Search dialog box but does not allow you to save a file to your server; that takes a little more work. This widget should be embedded within a form that has its encoding set to multipart, and when ready you should submit the form to your server to process the uploading of the file selected. This widget doesn't provide any server-side code to handle file upload; you have to provide that yourself, but you are free to do so in your favorite client-side language. We'll look at such a `FileUpload` in more detail when we discuss server-side components in chapter 12, but we'll also consider it briefly later in this chapter when we explore building our own widgets.

The `FileUpload` widget is perhaps one of the most inconsistently implemented widgets across browsers, with different browsers allowing differing security restrictions and ability to style it. Most

browsers, for example, will not permit you to set the default value of the text box since that would allow a web application to go fishing for files. As with all widgets, the thing to remember is that, if you cannot do something with the widget in HTML and JavaScript, then you cannot do it in GWT. To that end, GWT only provides a `getFilename()` method that retrieves the filename selected by a user. This method should not be confused with the `getName()` and `setName()` methods, which are used to set the DOM name of the `FileUpload` widget.

But enough of file uploading until chapter 12; next let's look at some of the other basic widgets that GWT provides.

Navigating your application with hyperlinks

The `Hyperlink` widget acts as an internal hyperlink within your GWT application. To users it appears exactly as a normal hyperlink on the web page; when they click the link, users expect some navigation within the application to occur. In your code, this action is coded as manipulating the GWT `History` object to change the application state—you can see how this works in the Dashboard's Slideshow application (`org.gwtbook.client.ui.calculator.Calculator`). The application has two hyperlinks placed at the bottom, as shown in figure 4.4, that enable the user to move the slideshow to the start or the end.

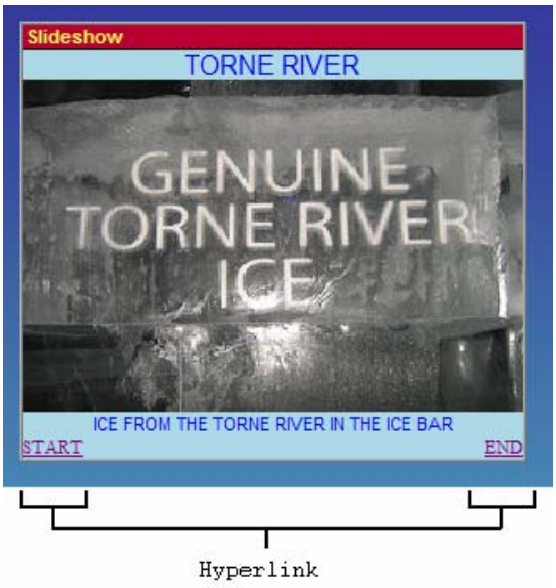

Figure 4.4 Two Hyperlink widgets (Start and End) in action at the bottom of the Slideshow Dashboard application.

Any component that uses a `Hyperlink` widget should also extend the `HistoryListener` interface and implement the `onHistoryChange()` method to catch and manage clicks on the hyperlinks. As you can see, the Dashboard's Slideshow component implements two `Hyperlinks`, which can be found in the code as

```
Hyperlink startLink = new Hyperlink("Start","0");                  #1
Hyperlink endLink = new Hyperlink("End",""+(maxNumberImages-1));   #2
```

Each `Hyperlink` widget constructor consists of two elements: the text that is to be displayed on the screen and a history token (which can be any arbitrary string). In the Slideshow component's case, all the history tokens represent numbers of a picture, starting at 0 for the first and ending at the maximum number of images in the slideshow, minus one. It is easy, then, to have a hyperlink to the start and end of the slideshow by using the appropriate values in the `Hyperlink` constructor—"0" for the start (as shown in [#1]) and the string representation of the largest image number (as shown in [#2]).

When using GWT history, you must remember to include the code shown in listing 4.1 in the body of your HTML page. Failure to do so will result in errors, which in Hosted mode are visible by errors in the Hosted Mode console, as shown in figure 4.5.

Listing 4.1 Implementing the GWT History subsystem

```
<iframe id="__gwt_historyFrame"
style="width:0;height:0;border:0"></iframe>
```

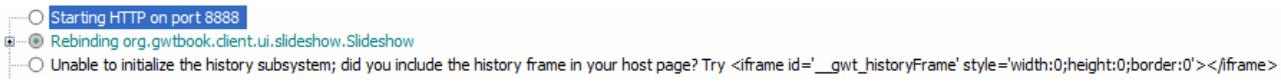

Figure 4.5 Error raised when trying to use the GWT History subsystem without it being properly initialized

> **Note:** The rebinding message you see in figure 4.5 is a result of some GWT manipulation we perform later in the book using GWT generators to take the basic component application and automatically generate some new code to show an About menu item.

Using methods in the `Hyperlink` class, it is easy to get the history token of a particular link (`getTargetHistoryToken()`) or even update the token if you wanted to (`setTargetHistoryToken()`). Similarly, you can set the hyperlink's text or get it through the `setText()` and `getText()` methods, respectively (or the HTML using `setHTML()` and `getHTML()` if you have created the hyperlink so that the text is treated as HTML, using the `Hyperlink(String, boolean, String)` constructor instead of the simpler `Hyperlink(String, String)` version).

Treating a hyperlink text as HTML means that any markup code, such as text in bold, underlines, or images, is displayed. If you want a normal hyperlink, to say another HTML page, then you should use the `HTML` widget, which we look at a little later, rather than a `Hyperlink` with text set to HTML. Hyperlinks are one way to navigate through an application; another is to use a menu system.

Navigating your application using menus

The menu system provided by GWT is based on the `MenuBar` and `MenuItem` widgets. `MenuItems` are added into `MenuBars` and `MenuBars` are added to other `MenuBars` in order to create your applications menu system. Figure 4.6 shows this system in place for the Dashboard menu system, where Clock, Calculator, and Slideshow `MenuItems` are added to a Create `MenuBar`. This Create `MenuBar` as well as a Help `MenuBar` are then added to another `MenuBar`, which is displayed on the browser page.

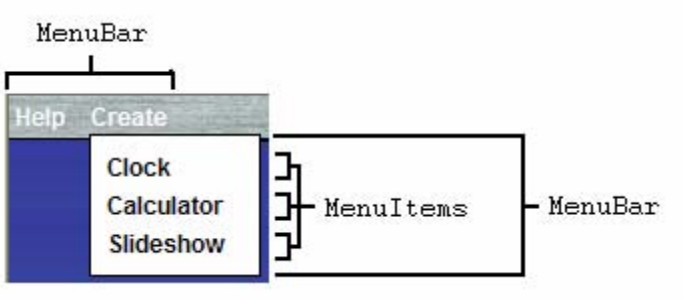

Figure 4.6 MenuBar and MenuItem widgets being used in the Dashboard application

In the Dashboard (`org.gwtbook.client.Dashboard`), we define one global `MenuBar` using the following code:

```
MenuBar menu = new MenuBar();
```

This simple line of code creates a horizontal menu bar (which we could just have easily created using the alternative constructor that takes a Boolean parameter whose value is `false` to create a horizontal menu bar, such as `new MenuBar(false)`). At present, the Dashboard will have two standard menu bars; we'll create two additional ones later in this book, and further down the Dashboard code you'll find two methods used to build the Create and Help menu bars. The method `buildHelpMenu()`, shown in listing 4.2, builds the initial Create menu bar using vertical menu bars. Creating vertical menu bars is performed by passing the Boolean value `true` as a parameter in the constructor (see [#1]).

Listing 4.2 Building the Dashboard's Create menu bar and menu items, together with the nested Locale menu bar

```
protected MenuBar buildHelpMenu(){
   MenuBar menuHelp = new MenuBar(true);                    |#1
   MenuBar menuLocale = new MenuBar(true);                  |#1

   menuLocale.addItem(constants.EnglishLocale(),
                      new ChangeLocaleToEnglishCommand());  |#2
   menuLocale.addItem(constants.SwedishLocale(),
                      new ChangeLocaleToSwedishCommand());
   menuHelp.addItem(constants.LocaleMenuItemName(), menuLocale);
#3
   return menuHelp;
}
```

<#1 Creates vertical MenuBars>
<#2 Adds MenuItem to MenuBar>
<#3 Adds MenuBar to MenuBar>

Within a `MenuBar`, you'll find one or more `MenuItems` or other `MenuBars` that go together to give the visual structure you saw in figure 4.6. It is possible to create `MenuItems` inline in the code, which is exactly what we do at step [#2], where the first parameter to the `addItem()` method is a new `MenuItem`. Each `MenuItem` is bound to a segment of code, the second parameter to the `addItem()` method, which is executed when the user clicks that menu item.

You can use something called the *command pattern* to describe the command that will be executed on a menu item click. In practice, this means that you create a new instance of either the

GWT `Command` class or a subclass, which contains an `execute()` method where your code is stored. For the Dashboard, we decided to define a number of `Command` subclasses as inner classes to the Dashboard, since this approach best met our needs. An example of one of these classes is shown in listing 4.3 (this is the command that is attached to the `MenuItems` defined in [#2] of listing 4.2).

Listing 4.3 Providing a subclass of the Command class to change the locale of the application to the English (default) locale

```
class ChangeLocaleToEnglishCommand implements Command{
   public void execute(){
      Window.removeWindowCloseListener(dashboardWindowCloseListener);
      changeLocale("");
   }
}
```

Using the command pattern allows the GWT application to turn a request for some future functionality into an object that can be passed around the code, and the defined functionality can then be invoked at a later point in time. In the case of the menu items, GWT provides the plumbing so that when a menu item is clicked, the associated command's `execute()` method is invoked. In this example, when the user clicks a menu item whose command is set to an instance of the `ChangeLocaleToEnglishCommand` class, then this `execute()` method is invoked, and a `WindowCloseListener` is removed from the application before the `changeLocale()` method is called (otherwise, we invoke both of the window closing events we set up in chapter 6—in the case of the Dashboard, these event handlers display two alert boxes, which is something we don't want to happen if we're just changing the locale).

You can interrogate a `MenuItem` to find out what its parent menu is by using its `getParentMenu()` method. In addition, you can use its `getSubMenu()` method to determine if it opens a submenu. Similarly, you can set the submenu (but not its parent menu) of a `MenuItem` through the `setSubMenu()` method. In some applications, you might need to change the command that is associated with a particular `MenuItem` (`setCommand()`) or even find out what that command is, which you can do using the `getCommand()` method.

The final implementation aspect of a `MenuItem` we want to discuss relates to the text shown as the item itself. This text can either be treated as pure text or as HTML, depending on whether a Boolean parameter is provided in the constructor of the `MenuItem`. As with any widget that can allow HTML to be set, you should always take care to avoid exposing the application to script-based security issues. Creating a menu item using new `MenuItem(String, true, Command)` treats the String as HTML.

A `MenuBar` widget allows you to specify whether its child menus should open automatically or wait for the user to click them to open. You use the `setAutoOpen()` method to achieve this, and in the Dashboard example, we do this in our `onModuleLoad()` method, where we create the whole menu system using the code shown in listing 4.4.

Listing 4.4 Creating the Dashboard's menu system

```
MenuBar menuCreate = buildCreateMenu();                |#1
MenuBar menuHelp = buildHelpMenu();                    |#1
menu.addItem(constants.HelpMenuName(),menuHelp);       |#2
menu.addItem(constants.CreateMenuName(),menuCreate);   |#2
menu.setAutoOpen(true);                                |#3
```

<#1 Builds SubMenus>

<#2 Adds SubMenus to MenuBar >
<#3 Sets SubMenus to open when mouseover >

At the top of the screen is the Dashboard's menu system, as seen in figure 4.7, which is involved a number of times in the code. We create the Help and Create menu bars as simple implementations for an Internet view (in the `Dashboard` class) and then override them to provide a more functional intranet version (in the `Dashboard_intranet` class). The version, intranet or Internet, which we've used here, is chosen by using GWT user-defined properties and setting the `externalvisibility` property in the `Dashboard.html` file; you'll learn all about this in chapter 15.

Figure 4.7 The Dashboard menu system, showing the four possible menu bars: the Help and Create menu bar are always present, the Bookmarks menu is loaded as XML from the server, and the application's Option menu bar is shown when a component application gains focus.

We create the text for these two menu bars using GWT internationalization constants set up for a default, English, locale, and an alternative Swedish locale. In chapter 15 we expand on this internationalization approach, which we looked at briefly in chapter 3.

We also make use of the two new `MenuItem` widgets we'll create later in this chapter in our Dashboard's menu system. When running in the intranet mode, the Help menu bar provides the user with the ability to turn on or off the request for confirmation when they delete a component application, and they do this through a `TwoComponentMenuItem`. In both modes, the user can change the locale through two `TwoComponentMenuItems`: one for each locale supported by our application. You can see the intranet view of the Help menu in action in figure 4.8, where both the new widgets are in use.

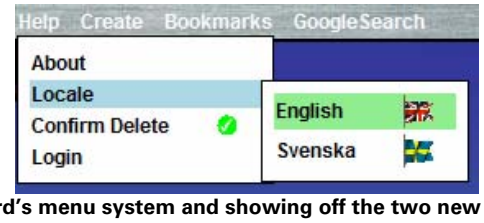

Figure 4.8 Examining the Dashboard's menu system and showing off the two new MenuItem widgets we built in chapter 4

We manipulate the menu bar in two further ways. First, we create a Bookmark menu bar whose contents are loaded from an external XML file by using the GWT implementation of the `HTTPRequest` object; we covered this topic in chapter 12. Second, each component application can register an Option menu when it gains focus, which is shown in the main menu when the application gains focus (we use a GWT Generator, as discussed in chapter 14, to automatically generate an `About` item in the Option menu for each application; this menu item lists all the methods and fields included in the application). Figure 4.8 shows the Option menu for the Google

Search component application. These component applications also have some generic functional requirements placed on them.

The final point to note about the `MenuBar` is that it implements the `PopupListener` interface, which allows functionality to be fired when the `MenuBar` is closed. If you wish to use different functionality when the `MenuBar` closes, then you can override the existing class and implement your own `onPopupClosed()` method. We don't use this functionality in the Dashboard, but that is how you would use it if you wished to do so.

Managing the view of data using trees

We have now nearly completed our look at most of the basic widgets in GWT, but there are two left. The first of these is the `Tree` widget. This widget provides your applications with a standard hierarchical tree widget consisting of `TreeItem` widgets. Figure 4.9 shows the `Tree` widget in action on the left-hand side.

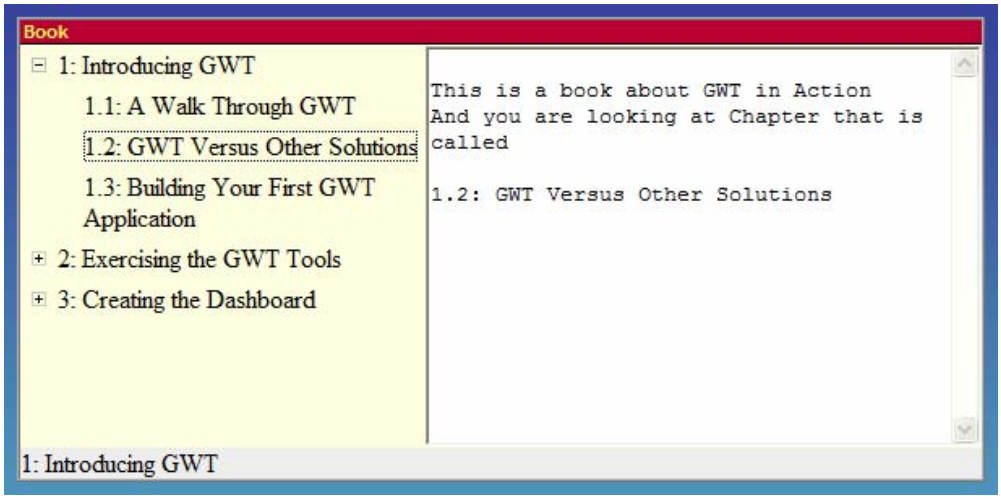

Figure 4.9 The Dashboard Book application showing the Tree widget on the left-hand side

You build a `Tree` in much the same way we built a menu earlier in this chapter. In that example we added a number of `MenuItems` to our `MenuBar`, but here we'll be adding `TreeItems` to a `Tree`. The constructors for a `TreeItem` are flexible and allow you to create an empty `TreeItem` or `TreeItem` from Strings or other widgets (which could mean a standard widget or a composite widget). In listing 4.5, which comes from the Dashboard's Book application (`org.gwtbook.client.ui.book.Book`), we build a simple tree to represent the top-level structure of our book.

Listing 4.5 Creating the Dashboard's Book tree system

```
private Tree buildTOC(){
   TreeItem chapter1 = new TreeItem("1: Introducing GWT");          |#1
   chapter1.addItem("1.1: A Walk Through GWT");                     |#1
   chapter1.addItem("1.2: GWT Versus Other Solutions");             |#1
   :
   TreeItem chapter2 = new TreeItem("2: Exercising the GWT Tools"); |#2
   chapter2.addItem("2.1: Setting up Dashboard Version 1");         |#2
   :
```

```
   TreeItem chapter3 = new TreeItem("3: Creating the Dashboard");
|#3
   chapter3.addItem("3.1: Stage 2 - Developing the Application");
|#3
   :
   Tree t = new Tree();    |#4
   t.addItem(chapter1); t.addItem(chapter2); t.addItem(chapter3);    |#4
   return t;
}
```
<#1 Builds Chapter 1 Tree>
<#2 Builds Chapter 2 Tree>
<#3 Builds Chapter 3 Tree>
<#4 Builds Table of Contents Tree >

Unlike with the menu system, we won't add commands to `MenuItems` in order to implement functionality when they are clicked or expanded; instead, we implement a `TreeListener`. This requires us to implement two methods: an `onTreeItemSelected()` method, which is fired when a `MenuItem` is selected, and the `onTreeItemStateChanged()` method, which gets invoked if the state (opened or closed) of a `TreeItem` changes.

For the Dashboard Book application, we implement the `TreeListener` shown in listing 4.6.

Listing 4.6 Creating the Dashboard's Book tree system

```
Tree theTree = buildTOC();
theTree.addTreeListener(new TreeListener(){
   public void onTreeItemSelected(TreeItem item) {        |#1
      changeText(item.getText());                         |#2
   }

   public void onTreeItemStateChanged(TreeItem item) {    |#3
      if (item.getState()){                               |#4
         currChapter.setText(item.getText());             |#5
      } else {
         currChapter.setText("----------");
      }
   }
});
```
<#1 Fires when TreeItem selected>
<#2 Gets text from selected item>
<#3 Fires when TreeItem state changes>
<#4 Gets state of item that changed>
<#5 Retrieves text>

This listener will call the Dashboard application's `changeText()` method ([#2]) to fill in the text box on the right of the Book application with some text when a `TreeItem` is selected ([#1]). When the state of a `TreeItem` changes, then the `onTreeItemStateChanged()` method ([#3]) is called. This method retrieves the state of the `MenuItem` that was changed ([#4]), and if this item is now open, it places the text of the time at the bottom of the widget by retrieving it using the `getText()` method ([#5])—otherwise, it inserts some dashes as the text.

A `TreeItem` comes with a host of helper methods that can be invoked to learn more about or change existing properties of the item. You can find out the child at a particular index (`getChild(int index)`), count the number of children (`getChildCount()`), or get the index of a particular child (`getChildIndex(TreeItem)`). Additionally, you can get the `Tree`

a particular `TreeItem` is within (`getTree()`), find out if an item's children are currently displayed (`getState()`), its parent item (`getParentItem()`), and whether it is currently selected (`isSelected()`).

`TreeItems` may have a widget associated with them, though if it does it cannot directly have text associated unless it's set up as a composite widget. Quite often you might associate a CheckBox, for example, with `TreeItems`. You associate a widget with a `TreeItem` either through the `setWidget()` method or by using the `TreeItem(Widget)` constructor.

Viewing images

Finally, for the basic widgets, if you wish to display an image in a GWT application, the `Image` widget is the one you'll use. This widget allows images to be loaded and displayed, as you can see in figure 4.10.

Figure 4.10 Looking at the Image widget in action in the Dashboard's Slideshow application

An interesting aspect of the `Image` widget is the ability to add a `LoadListener` to it so that a particular action can be performed when the image has completed loading (`onLoad()`), or another action performed if an error occurs when loading the image (`onError()`).

> **Tip:** The `LoadListener` will only work if the `Image` is actually added to the browser page, usually via a panel, which is itself added to the browser. Just creating the Java object and adding a `LoadListener` is not enough to catch the events due to the way in which the GWT event-handling system works (see chapter 6).

In listing 4.7 we show the code from the Dashboard's `Slideshow` component (`org.gwt.client.ui.Slideshow.Slideshow`) that could be used to preload images into an application. The `preloadImages` object is actually a `HorizontalPanel`, which we have added to our application specifically for the future use of preloading images. Because of the way the GWT event mechanism works, we need to have our images loading into a component that is added to the browser page (if they aren't, then there is no hook into the event mechanism, and the `LoadListener` is ignored). However, there is no requirement for the component to be visible, and so in [#1] we set it to be invisible to avoid an unsightly mess!

Listing 4.7 Preloading Slideshow Images making use of a LoadListener

```
Image[] testLoading = new Image[maxNumberImages];
preloadImages.setVisible(false);                                    |#1
for(int loop=0;loop<maxNumberImages;loop++){
   testLoading[loop] = new Image(theImages[loop][1]);
   testLoading[loop].addLoadListener(new LoadListener(){            |#2
      public void onError(Widget sender) {                          |#3
         Window.alert("Expected Error - onError() Method works.");
      }
      public void onLoad(Widget sender) {                           |#4
         Window.setTitle("Loaded Image: "+imageName);
      }
   });
   preloadImages.add(testLoading[loop]);
}
```

<#1 Hides panel>
<#2 Adds LoadListener>
<#3 Defines onError code>
<#4 Defines onLoad code >

We add a new `LoadListener` at [#2] and define the `onError()` method to display a JavaScript alert on the screen if there is an error ([#3]). Or if the image loads, we change the title bar of the browser window to display the image name using the `Window.setTitle()` method ([#4]).

Once we have an `Image` object, it is possible to use the `prefetch()` or `setUrl()` method to load a new image rather than creating new objects as we did in our example—but either way is valid and you will make your choice as to which way you set it up in your own applications.

The only other thing about images that you should be aware of is that you might encounter some issues when you're using a transparent PNG image over another image in Microsoft Internet Explorer 5.5 and 6; some ugly backgrounds may be applied. But we'll build a new widget a little later in this chapter that overcomes these problems.

4.1.2 Displaying text on the application

The `Image` widget we looked at just now was useful for displaying pictures and images on the web browser. However, a lot of your applications will need to show text either as some sort of passive information or sometimes more actively or in a funky way. As we dig down into the hierarchy of widgets, we find two widgets that allow us to present text on the screen: the `Label` and `HTML` widgets.

Showing text as a label

A `Label` widget contains arbitrary text, which is displayed exactly as written. This means that the `Label` created by the code `new Label("Hi <B>there</B>")` would appear on the browser page exactly as "Hi **there**"—that is, the word "there" is not interpreted as HTML and is shown in bold text.

It is possible to control the horizontal alignment of labels, though by default the size of a `Label` is the size of the text it encloses. So right-aligning, using the command

```
theLabel.setHorizontalAlignment(HorizontalAlignmentConstant.ALIGN_RIGHT
)
```

would have little visible effect unless you use a style sheet (or less preferably the `theLabel.setWidth()` method) to set the width of the label to be longer than the text. The alignment that is visible within the Dashboard's `ServerStatus` application (`org.gwtbook.client.ui.serverstatus.ServerStatus`), as you can see in figure 4.11, is achieved by aligning `Labels` in a `Grid` panel (see chapter 5). A `Label` may also word-wrap if the `setWordWrap()` method is called, which takes a Boolean variable set to `true` if the `Label` should word-wrap and `false` otherwise.

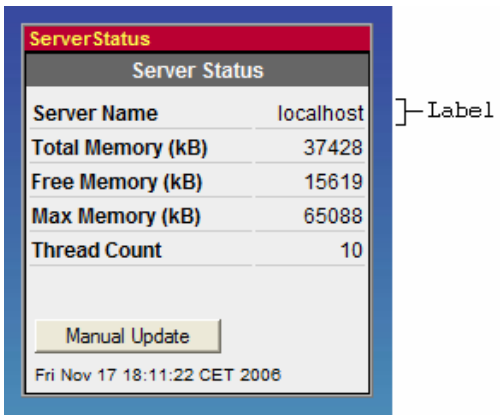

Figure 4.11 The Dashboard's Server Status application showing GWT Label widgets in action

GWT allows you to add `ClickListener` and `MouseListener` to a standard `Label` widget, offering the possibility to capture the user trying to interact with the `Label`. By default no action occurs; you have to add click or mouse listeners yourself. In the `EditabelLabel` widget from chapter 7 (`org.gwtbook.client.ui.EditableLabel`), when a user clicks the label we want to present a text box that the user can use to change the text of the label. We add the click listener as shown in listing 4.8 at [#1].

Listing 4.8 Adding a ClickListener to the GWT in Action EditabelLabel widget

```
text = new Label(labelText);
text.setStyleName("editableLabel-label");
text.addClickListener(new ClickListener()                |#1
{
   public void onClick (Widget sender){
      changeTextLabel();
   }
});
```

(annotation) <#1 Adds a ClickListener>

The `ClickListener` acts in a similar way to the `Command` we used in the `MenuItem` widget. It registers an `onClick()` method that GWT will execute when the user of the application clicks the `Label`.

Once you have a `Label`, it is possible to change the text programmatically (see listing 4.9) using the `setText()` method (as well as getting the current text using the `getText()` method). However, you can have a slightly more active text-presenting widget if you wish: the `HTML` widget.

Listing 4.9 Changing the Label text in the Clock application

```
Date d = new Date();
if (! local) {
   d = new Date(d.getTime() - (d.getTimezoneOffset() * 60 * 1000));
}
clockLabel.setText(d.getHours() +                                    |#1
           ":" + twoDigit(d.getMinutes()) +
           ":" + twoDigit(d.getSeconds()));
```

(annotation) <#1 Sets Label to new time>

Making text active using the HTML widget

If we want to provide some more funkiness to your text presentation, then the `HTML` widget could be exactly the component you're looking for. It acts in the same way as a `Label` widget, but more important, it interprets any text as arbitrary HTML. Whereas in the `Label` the text "Hi <B>there</B>" was written as is, if you write the code `new HTML("Hi <B>there</B>")`, it is displayed as "Hi **there**".

The `HTML` widget is also useful if you wish to provide a true hyperlink. In our earlier discussion of the `Hyperlink` widget, you learned that you can present what *looks* like a hyperlink to the user, but when clicked it only changes the historical aspect of the application. If you use an `HTML` widget instead, then you can provide proper links:

```
new HTML("<a href='http://www.manning.com'>Manning</a>");
```

However, you must be careful with this widget; allowing arbitrary HTML can expose security issues if maliciously constructed HTML is used. Also, consider whether the HTML panel that we discuss in chapter 5 is more appropriate for your needs.

Labels and HTML are a useful way of presenting information to a user, but there is another part to interacting with the user: capturing their input.