**Using the Multiplexing Look and Feel**

*This document is based on an article originally published in* [*The Swing Connection*](http://java.sun.com/products/jfc/tsc/)*.*

The Multiplexing look and feel lets you supplement an ordinary look and feel (called the *default* look and feel) with one or more *auxiliary* look and feels. For example, you could simultaneously provide text-to-speech and Braille outputs, in addition to the ordinary visual output that a Swing-based application generates, by adding two auxiliary look and feels (one for text-to-speech, the other for Braille) to the default look and feel. The default look and feel can be any ordinary look and feel -- the Java or Windows look and feel, for example -- and requires no modifications to work with auxiliary look and feels.

This document has the following sections:

* [Overview](#gjdgxs)
* [How to Use Auxiliary Look and Feels](#30j0zll)
* [Tips for Writing an Auxiliary Look and Feel](#1fob9te)
  + [Dos and Don'ts](#3znysh7)
  + [Extending UIDefaults](#2et92p0)
  + [Examining Other UI Objects](#tyjcwt)
* [How the Multiplexing Look and Feel is Implemented](#3dy6vkm)
* [How to Provide a Custom Multiplexing Look and Feel](#1t3h5sf)

Before reading further, you should be familiar with the concept of pluggable look and feels. For basic information, see [Choosing the Look and Feel](http://java.sun.com/docs/books/tutorial/uiswing/start/swingTour.html#plaf), a section in *The Java Tutorial*. For architectural details, you can read [Pluggable look-and-feel architecture](http://java.sun.com/products/jfc/tsc/articles/architecture/#pluggable), a section within a *Swing Connection* article.

**Overview**

The classes in the javax.swing.plaf.multi package implement a *multiplexing look and feel*. A multiplexing look and feel transparently creates -- and simultaneously supports -- UI objects from several different look and feels in response to a component requesting its UI object (with the getUI method).

Without a multiplexing look and feel, a developer who wanted to enhance a particular look and feel would need to extend the classes supporting that look and feel. For example, to add text-to-speech support to the Java look and feel without using a multiplexing look and feel, the developer would need to create a group of classes that extend those of the Java look and feel, and add text-to-speech support to the new classes. If the developer also wanted to add text-to-speech support to other look and feels, such as Motif or Windows, the developers would need to create subclasses of those classes as well.

This approach has at least two shortcomings:

* First, each subclass must use what is essentially a copy of the same code, potentially creating a difficult support situation for the developer.
* Second, and more significantly for the end user, some application developers might force a particular look and feel to be used. When this approach is used, the end user can't even use the enhanced look and feel.

A multiplexing look and feel both these problems simultaneously because it allows multiple look and feels to be combined. The first problem (having to use what amounts to a second copy of the same code) is solved because the developer can create a specialized look and feel that can then be combined with other look and feels.

The second problem (having to force the use of a particular look and feel) is solved because a specialized look and feel can be used with whatever default look and feel the application may have locked in place.

The default multiplexing look and feel implementation, represented by the MultiLookAndFeel class in the javax.swing.plaf.multi package, is called (unsurprisingly) the Multiplexing look and feel.

**How to Use Auxiliary Look and Feels**

It's easy to use auxiliary look and feels with Swing. To instruct Swing to use the Multiplexing look and feel, all an application has to do is modify the $JDKHOME/lib/swing.properties file to include a definition of the swing.auxiliarylaf property. Swing treats the swing.auxiliarylaf property as a comma-separated list of LookAndFeel subclasses that specify what auxiliary look and feels should be used in addition to the default look and feel. If at least one valid LookAndFeel subclass is specified in the swing.auxiliarylaf property, Swing automatically uses the Multiplexing look and feel to load and support the default and auxiliary look and feels.

For example, let's assume that an application makes use of a look and feel that supports text-to-speech feedback, and also uses an look and feel that adds support for a device that emits perfume. Let's assume that the text-to-speech look and feel is named com.myco.TextTalkerLookAndFeel, and the look and feel that adds support for perfume is named com.smellco.OlfactoryLookAndFeel.

To tell Swing to use both these look and feels -- and to use a default look and feel at the same time -- your application could simply add the following line to the $JDKHOME/lib/swing.properties file:

    swing.auxiliarylaf=com.myco.TextTalkerLookAndFeel,

        com.smellco.OlfactoryLookAndFeel

This statement tells Swing to obtain a component's UI from the Multiplexing look and feel automatically, instead of obtaining it directly from the default look and feel. The resulting multiplexing UI is a small delegate that obtains and maintains UIs from the default and auxiliary look and feels. As a result, when a method is invoked in a multiplexing UI object, the multiplexing UI invokes the same method on each of the UIs obtained from the default and auxiliary look and feels.

**Tips for Writing an Auxiliary Look and Feel**

An auxiliary look and feel is like any other look and feel, except that it doesn't have to provide the complete support that a default look and feel must. For example, an auxiliary look and feel that supports just text-to-speech feedback doesn't need to provide any code for painting. Also, it might not need to support all components -- JSeparators, for example, might be ignored.

Auxiliary look and feels tend to be simple, so developing one can be easier than developing a visual look and feel. The developer can concentrate solely on providing the specialized functionality.

Because the primary purpose of an auxiliary look and feel is to enhance the default look and feel, auxiliary look and feels tend be nonvisual. Since an auxiliary look and feel is a genuine look and feel, however, there is nothing to prevent it from rendering information on the display.

Just like for any other look and feel, you implement an auxiliary look and feel by writing a subclass of javax.swing.LookAndFeel and creating subclasses of the *Foo*UI classes defined in the javax.swing.plaf package.

**Dos and Don'ts**

The following paragraphs provide some general recommendations for developing auxiliary look and feels.

**Use the installUI method to perform all initialization, and the uninstallUI method to perform all cleanup.**

The installUI and uninstallUI methods are invoked when a component's look and feel is set. The installUI method gives the new UI object a chance to add listeners on the component and its data model. Similarly, the uninstallUI method lets the previous UI object remove its listeners.

**Don't extend visual look and feels.**

We recommended that you *don't* implement UI classes of an auxiliary look and feel as subclasses of the UI classes of a visual look and feel.�Why not? Because they might accidentally inherit code that installs listeners on a component object or renders the component on the display.�As a result, your auxiliary look and feel would compete with the default look and feel rather than cooperating with it.  
  
Instead, we recommend that the UI classes of an auxiliary look and feel directly extend the abstract UI classes in the javax.swing.plaf package.�By using this strategy, the developer of an auxiliary look and feel can avoid competing with the default look and feel.

**Override all UI-specific methods your UI classes inherit.**

We recommend that each UI class of an auxiliary look and feel override the methods defined in the javax.swing.plaf UI classes it descends from The reasons for this recommendation are similar to those for not extending a visual look and feel. For example, the ComponentUI class, from which all UI classes descend, provides a default implementation for the update method. This default implementation paints on the display if the component is opaque. If a UI class from a non-visual auxiliary look and feel does not override this method, all opaque components appear as blank areas on the screen!

**Extending UIDefaults**

In many cases, you might want an auxiliary look and feel to be "incomplete."�That is, you might not need to support the complete set of components. For example, an auxiliary look and feel might choose to provide a ButtonUI subclass but no LabelUI subclass. This option is allowed, and the multiplexing look and feel gracefully handles such situations.

By default, however, Swing issues an error message when it asks a look and feel for a UI object and the look and feel does not support that UI. This message can be annoying, especially to auxiliary look-and-feel developers who don't want to support a particular component.

Fortunately, you can prevent this error message by creating a subclass of the UIDefaults class and returning an instance of it from the getDefaults method of your LookAndFeel class. For example:

public class MyAuxLookAndFeel extends LookAndFeel {

��� ...

��� public UIDefaults getDefaults() {

������� **UIDefaults table =**

**new MyAuxUIDefaults();**

������� Object[] uiDefaults = {

��������� "ButtonUI", "MyAuxButtonUI",

��������� ...

������� }

������� table.putDefaults(uiDefaults);

������� return table;

��� }

}

**class MyAuxUIDefaults extends UIDefaults {**

**��� protected void getUIError(String msg) {**

**������� //System.err.println**

**//   ("An annoying message!");**

**��� }**

**}**

In the preceding example, an auxiliary look and feel named MyAux creates a UIDefaults subclass that overrides the getUIError method.�The getUIError method is the method that is invoked when Swing cannot find a UI object in a look and feel.�By merely doing nothing in this method, you can avoid the error message.

**Examining Other UI Objects**

In rare instances, a UI object from an auxiliary look and feel may be interested in the default UI object used by the component.�In these cases, the UI object from auxiliary look and feel can obtain the UI from a component by calling its getUI method.�The returned UI is an instance of one of the multiplexing look and feel UI classes (for example, MultiButtonUI). The UI object from the auxiliary look and feel can call the getUIs method of the returned object to obtain an array containing a complete list of all UI objects handled by the multiplexing UI. The first element is guaranteed to be the UI created from the default look and feel.

**How the Multiplexing Look and Feel Is Implemented**

The Multiplexing look and feel (represented by javax.swing.plaf.multi.MultiLookAndFeel) is meant to be transparent to all developers and users.�It should "just work" -- and it is used only when the user tells Swing to use an auxiliary look and feel.

When the Multiplexing look and feel is in use, the type of the UI object associated with each component depends on whether any of the auxiliary look and feels currently in use support the component. If so, the component's UI object is an instance of a multiplexing UI. If only the default look and feel supports the component, then the component gets a UI object from the default look and feel, just as if no auxiliary look and feels were installed.

A multiplexing UI object obtains and maintains UI objects from the default and auxiliary look and feels, referring to these UIs in the following manner:

* The UI object from the default look and feel is always the first to be created. After that, a UI object is created from each auxiliary look and feel in the order they are specified in the swing.auxiliarylaf property.
* When a method that requests information from a UI object is invoked, the multiplexing UI object invokes the method on all the UI objects, but returns only the results from the UI for the default look and feel. For example, when the getPreferredSize method is invoked on a multiplexing UI, the UI returns only the results of invoking getPreferredSize on the UI obtained from the default look and feel. The getPreferredSize method is also invoked on the UI object for each auxiliary look and feel, but the return values are ignored.
* When a method that does not request information from the UI object is invoked, the multiplexing UI object invokes that method on all UIs -- on the UI object obtained from the default look and feel and on all the UIs obtained from the auxiliary look and feels, as well. For example, invoking the installUI method on a multiplexing UI causes the multiplexing UI to invoke installUI on the UI obtained from the default look and feel and the UIs obtained from the auxiliary factories.

In all cases, the UI object obtained from the default look and feel is acted upon first, and then the auxiliary look and feels are acted upon in the order they are specified in the swing.auxiliarylaf property.

**How to Provide a Custom Multiplexing Look and Feel**

While we hope the behavior of the Multiplexing look and feel is flexible enough not to require an alternative multiplexing look and feel, Swing allows the user to specify another multiplexing look and feel to use.

To do that, all the user has to do is modify the $JDKHOME/lib/swing.properties file to include a definition of the swing.plaf.multiplexinglaf property.�Swing then treats the swing.plaf.multiplexinglaf property as a LookAndFeel subclass that supports multiplexing.

For example, if a user has a multiplexing look and feel represented by com.myco.SuperMultiLookAndFeel that is a better match for their needs than the Multiplexing look and feel (javax.swing.plaf.multi.MultiLookAndFeel), the user could include the following line in $JDKHOME/lib/swing.properties:

swing.plaf.multiplexinglaf = com.myco.SuperMultiLookAndFeel

This statement instructs Swing to use com.myco.SuperMultiLookAndFeel instead of javax.swing.plaf.multi.MultiLookAndFeel. But if you use this kind of statement, be careful, because the suppliers of auxiliary look and feels will most likely have developed and tested against our Multiplexing look and feel.