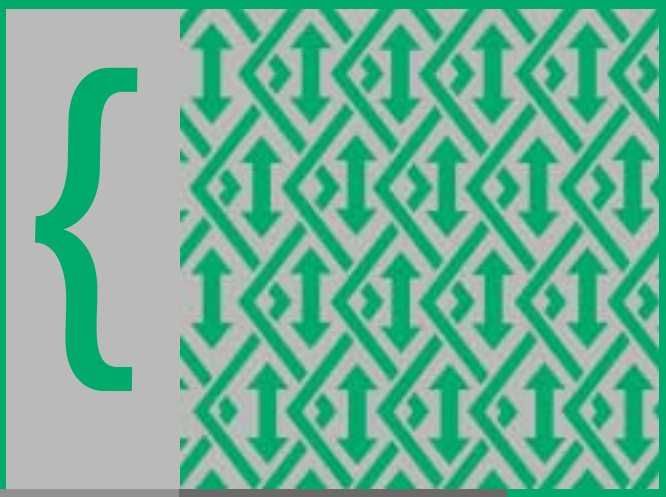
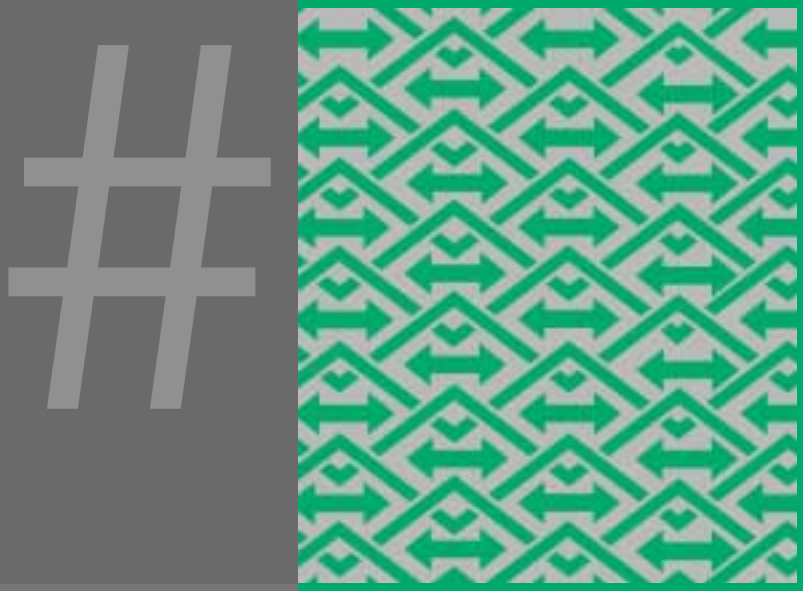
;login:

THE MAGAZINE OF USENIX & SAGE
October 2000 • volume 25 • number 6



inside:

PROGRAMMING
Using Java



USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

using java

Life After Containers and Layout Managers: Part II

by Prithvi Rao

Prithvi Rao is the co-founder of KiwiLabs, which specializes in software engineering methodology and Java/CORBA training. He has also worked on the development of the MACH OS and a real-time version of MACH. He is an adjunct faculty at Carnegie Mellon and teaches in the Heinz School of Public Policy and Management.

<prithvi+@ux4.sp.cs.cmu.edu>

In Part I of this topic, I presented a brief background on "Containers" and "Layout Management." I also discussed the use of "Panels" and "Frames."

One motivation for this was the increased popularity of GUI-based client-side applications. For instance, in the electronic-commerce domain the canonical n-tiered architecture often relies on a "thin" client that can be a GUI-based Java program. This does not imply that the console-based paradigm is being abandoned; it suggests that using GUI applications permits a more suitable semantic format for expressing a product's capability. An example of this is the merging of two database queries from two different databases. In this example a user may prefer to click and drag icons to merge these queries as opposed to typing them at a console.

In this article, I will present the use of "Dialogs" and how to handle "Menus." This will help to consolidate the information in Part I.

Dialogs

A dialog is normally a short-term popup window with a titlebar, border, and perhaps some other pieces whose client area groups a set of User Interface (UI) controls soliciting input from a user and performing an action. The dialog is responsible for creating and positioning the controls and dealing with the notification they generate to provide a coherent dialog with the user.

In most windowing systems, because of the static nature of the control content in the dialog and the static nature of the size and position of the controls, some kind of resource (read-only) data item will describe the dialog and its content. The dialog resource can be used to easily create the dialog box and its contents and position or to resize the controls. The AWT (Abstract Windowing Toolkit) does not have resources, so dialog boxes have to be constructed dynamically in the application as required.

The AWT Dialog class is derived from the Window class. It is different from regular windows because they are dependent on a "Frame." When the Frame is destroyed, all the dialogs associated with that Frame are also destroyed. If the Frame is iconified, the Dialogs dependent upon it also disappear from the screen. When the Frame is deiconified, its dependent Dialogs return to the screen. This behavior is intrinsic to the AWT, and no specific programming is required within the application to support capability.

Recall that in the case of Applets, they are running in their own Frame. This means that they cannot use Dialogs directly. Applets must bring up dialogues in their "own" Frames.

Modal dialogs require the attention of users, and they prevent the user from doing anything else in the Dialog's applet application until it has been dealt with. By default, dialogs are nonmodal. A bug in the JDK1.1 release prevented the creation of dialog subclasses that were modal. That restriction is not there in JDK1.2.

File Dialogue

The class FileDialog is derived from the Dialog class and displays a dialog window form which the user can select a file.

The user sets up the dialog by calling methods such as:

```
FileDialog.setFile()
FileDialog.setDirectory();
FileDialog.setFilenameFilter(); (determines the initial search criteria).
```

This is an example of a "modal" dialogue. In other words, when its show() method is invoked, it blocks the rest of the application until the user has chosen a file.

To determine which file has been chosen, the application issues a call to:

```
FileDialog.getFile();
```

Menus

A menu is a hierarchical group of components. A MenuItem represents a command. A Menu (often can be a sub-menu) is a group of MenuItems or other Menus.

The top level grouping is a Menubar and contains only Menus. There is also a CheckboxMenuItem class that maintains a check of the menu item that is currently selected. None of these classes inherits from Component; they are instead a subclass of the MenuComponent class.

In order to be able to contain a MenuComponent an object must adhere to the MenuContainer interface. Recall that classes such as Frame, Menubar, and Menu are all implementations of the MenuContainer interface. Additionally these are the only classes that implement this interface. In theory, it should be possible to write a MenuContainer interface implementation to which we can attach a MenuBar, Menu, or MenuItem.

In most windowing systems, some kind of resource (read-only data item) will describe the initial state and contents; this is true even though the state of the menu and its contents may change dynamically.

The JDK 1.0 release does not support resources, so menus must be constructed "on the fly" in the code. In JDK 1.1 and subsequent releases resources are supported and this is done through the internationalizable class (which is beyond the scope of this article).

The following is an example of some code in a Frame-derived class to provide a menu:

```
MenuBar bar = new MenuBar();
Menu popup = new Menu("File", true);
MenuItem item = new MenuItem("Exit");
// set a menu bar to be added
setMenuBar(bar);
// now add component so that it can be viewed
bar.add(popup);
// now add the menuitem
popup.add(item);
popup = new Menu("Edit", true);
bar.add(popup);
popup.addSeparator();
item = new MenuItem("Cut");
popup.add(item);
item = new MenuItem("Copy");
popup.add(item);
item = new MenuItem("Paste");
popup.add(item);
item = new MenuItem("Delete");
popup.add(item);
item = new MenuItem("Select All");
popup.add(item);
```

In most windowing systems, some kind of resource will describe the initial state and contents.

For serious GUI developers, it is preferable to use the "Swing" classes because they obviate the need to implement many of the interfaces.

```
popup.addSeparator();
item = new MenuItem("Properties");
popup.add(item);
popup = new Menu("Help", true);
bar.add(popup);
item = new MenuItem("About....");
popup.add(item);
bar.setHelpMenu(popup);
```

The key to understanding the above code is that the MenuItem, Menu, CheckBoxMenuItem, and MenuBar classes all derive from the MenuComponent and *not* the Component class.

Also in order to support menus, it is necessary to implement the MenuContainer interface (Frame, MenuBar, Menu, . . .).

Finally, MenuItems generate events.

Conclusion

In Part I, I discussed the use of layout managers and their use in writing effective GUI applications in Java. I also presented an example of how to manipulate components in the absence of a layout manager at the expense of portability.

In Part II, I have presented an example of how to create a menu.

For serious GUI developers, it is preferable to use the "Swing" classes because they obviate the need to implement many of the interfaces; in other words, it is done for you. However, for those interested in understanding the details of how the AWT supports GUI development, the examples presented will facilitate that endeavor.