Part II:

Writing Scripts for Tk

Chapter 14
An Introduction to Tk

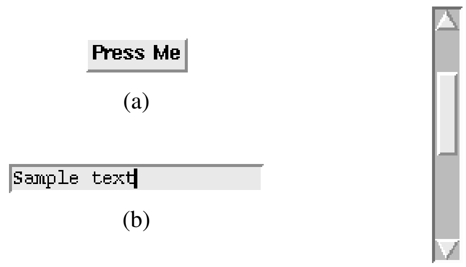
Tk is a toolkit that allows you to create graphical user interfaces for the X11 window system by writing Tcl scripts. Like Tcl, Tk is a C library package that can be included in C applications. Tk extends the built-in Tcl command set described in Part I with several dozen additional commands that you can use to create user interface elements called *widgets*, arrange them into interesting layouts on the screen using *geometry managers*, and connect them with each other, with the enclosing application, and with other applications. This part of the book describes Tk's Tcl commands.

In addition to its Tcl commands, Tk also provides a collection of C library functions that can be invoked from C code in a Tk-based application. The library functions allow you to implement new widgets and geometry managers in C. They are discussed in Part IV of the book.

This chapter introduces the basic structures used for creating user interfaces with Tk, including the hierarchical arrangements of widgets that make up interfaces and the main groups of Tcl commands provided by Tk. Later chapters will go over the individual facilities in more detail.

Note: *I've taken the liberty of describing things in the way I expect them to be when the book is finally published, so the descriptions in this draft do not always correspond to the current version of Tk (3.2). The following discrepancies exist between this draft and Tk 3.2: (a) the* `pack` *command syntax as described here is different than what exists in 3.2, although it provides almost exactly the same set of features; (b) Tk 3.2 doesn't contain all of the built-in bitmaps listed here (c)* `groove` *and* `ridge` *reliefs are not supported in Tk 3.2, and (d) embedded widgets are not yet supported in text widgets. As new versions of Tk are released the discrepancies should gradually disappear.*

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.



(a)

(b)

(c)

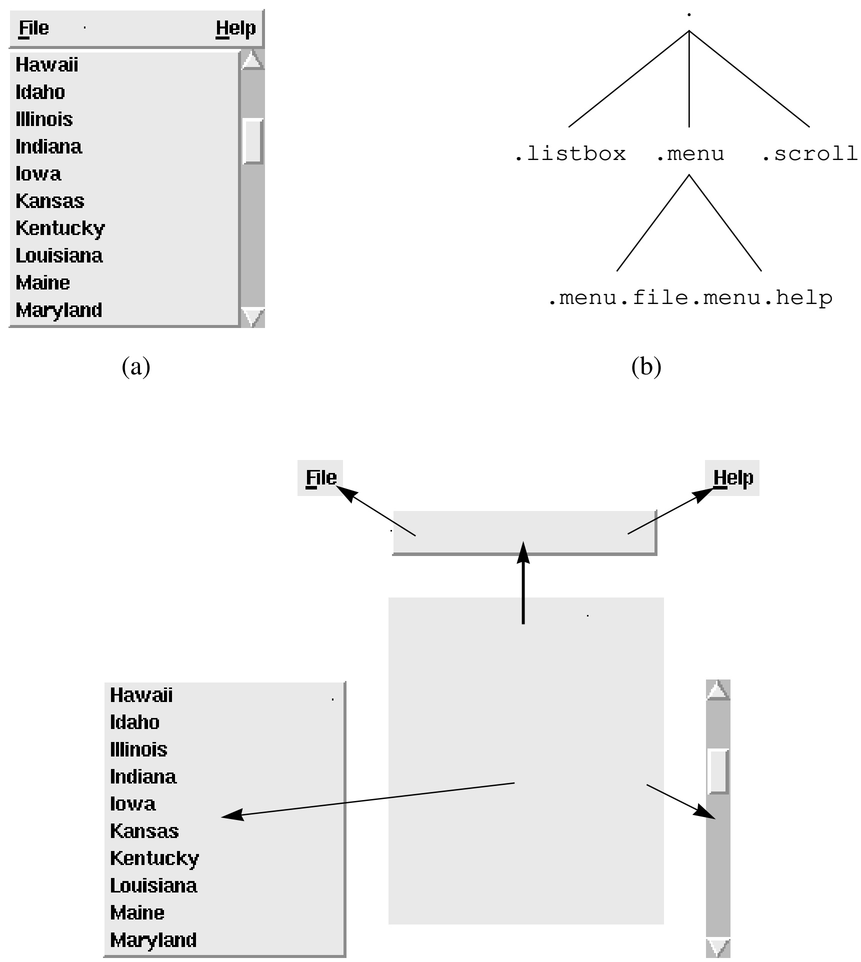
Figure 14.1. Examples of widgets in Tk: (a) a button widget displays a text string and invokes a given Tcl command when a mouse button is clicked over it; (b) an entry widget displays a one-line text string and allows the text to be edited with the mouse and keyboard; (c) a scrollbar widget displays a slider and two arrows, which can be manipulated with the mouse to adjust the view in some other widget.

14.1 Widgets and windows

The basic user interface elements in Tk are called *widgets*. Examples of widgets are labels, buttons, pull-down menus, scrollbars, and text entries (see Figure 14.1). Widgets are grouped into *classes*, where all of the widgets in a class have a similar appearance on the screen and similar behavior when manipulated with the mouse and keyboard. For example, widgets in the button class display a text string or bitmap as shown in Figure 14.1(a). Different buttons may display their strings or bitmaps in different ways (e.g. in different fonts and colors), but each one displays a single string or bitmap. Each button also has a Tcl script associated with it, which is invoked whenever mouse button 1 is pressed with the mouse cursor over the widget. Different button widgets may have different commands associated with them but each one has an associated command. When you create a widget you select its class and provide additional class-specific *options*, such as a string or bitmap to display or a command to invoke.

Tk's built-in widget classes implement the Motif™ look-and-feel standard specified by the Open Software Foundation. The Motif standard determines the three-dimensional look that you'll see in the Tk widgets and many aspects of their behavior.

Each widget is implemented using one window in the X window system, and the terms "window" and "widget" are used interchangeably in this book. Widgets may be nested in hierarchical arrangements with widgets containing other widgets that contain still other widgets. The result is a tree-like structure such as the one shown in Figure 14.2. Each widget can contain any number of children and the widget tree can have any depth. The widgets with behavior that is meaningful to the user are usually at the leaves of the widget tree; the higher-level widgets are usually just containers for organizing and arranging the leaf widgets.



(a) (b)

(c)

Figure 14.2. Widgets are arranged hierarchically. A collection of widgets is shown in (a) as it appears on the screen, and the hierarchical structure of the collection is shown in (b). An exploded view of the screen is shown in (c) to clarify the widget structure. The topmost widget in the hierarchy (".") contains three children: a menu bar across the top, a scrollbar along the right side, and a listbox filling the remainder. The menu bar contains two children of its own, a `File` menu button on the left and a `Help` menu button on the right. Each widget has a name that reflects its position in the hierarchy, such as `.menu.help` for the `Help` menu button.

Each widget/window has a textual name that is used to refer to it in Tcl commands. Window names are similar to the hierarchical path names used to name files in Unix, except that "`.`" is used as the separator character instead of "`/`". The name "." refers to the topmost window in the hierarchy, which is called the *main window*. The name `.a.b.c` refers to a window `c` that is a child of window `.a.b`, which in turn is a child of `.a`, which is a child of the main window.

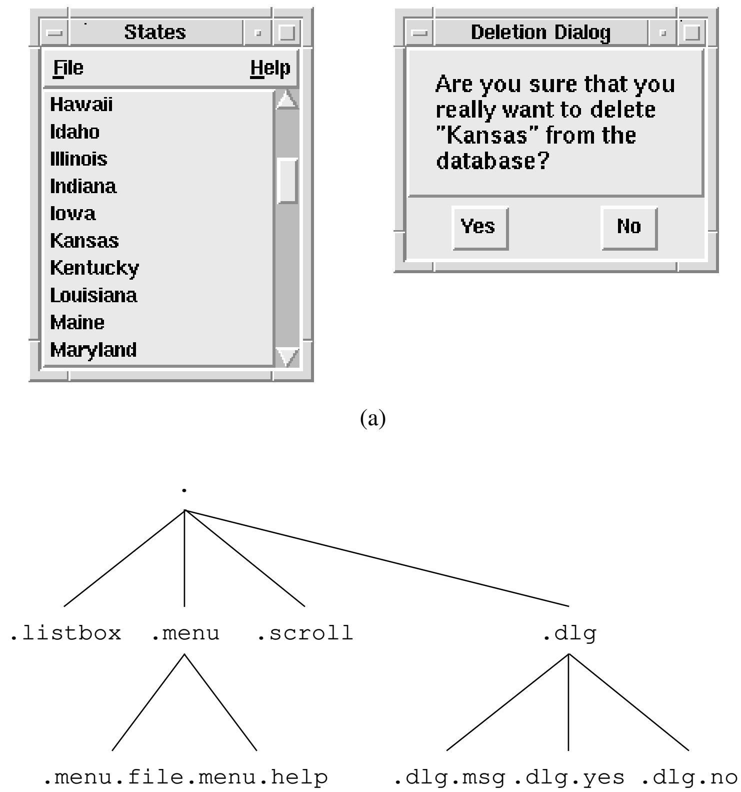
14.2 Screens, decorations, and toplevel windows

Tk creates the main window of an application as a child of the root window of a particular screen. This causes the main window to appear on that screen. Your window manager will then create a decorative frame around the main window, which usually displays a title and provides controls that you can use to move and resize the window. A given window manager will decorate all applications in the same way, but different window managers may use different styles of decoration. Figure 14.2 showed a main window without any window manager decoration; other figures will show decorations as provided by the `mwm` window manager (e.g. see Figure 14.3).

X clips each window to the area of its parent: it will not display any part of a window that lies outside the area of its parent. The descendants of the main window are called *internal windows* to reflect the fact that they appear inside the area of the main window. However, applications often need to create widgets that are don't lie inside the main window. For example, it might be useful to position a dialog box in the center of the screen regardless of the position of the main window, or an application might wish to post several panels that the user can move around on the screen independently. For situations like this Tk provides a third kind of window called a *top-level window*. A top-level window appears like an internal window in the application's widget hierarchy (e.g. it might have a name like `.a.b`) but its X window is created as a child of the screen's root rather than its parent in the Tk widget hierarchy. The window manager will treat top-level windows just like main windows, so the user will be able to move and resize and iconify each top-level window separately from the main window and other top-level windows. Top-level windows are typically used for panels and dialog boxes. See Figure 14.3 for an example.

It is not necessary for all of the widgets of an application to appear on the same screen or even the same display. When you create a top-level widget you can specify a screen for it. The screen defaults to the screen of the widget's parent in the Tk hierarchy, but you can specify any screen whose X server will accept a connection from the application. For example, it's possible to create a Tk application that broadcasts an announcement to a number of wokstations by opening a top-level window on each of their screens.

Once a widget is created on a particular screen, it cannot be moved to another screen. This is a limitation imposed by the X window system. However, you can achieve the same effect as moving the widget by deleting it and recreating it on a different screen.



(a)

(b)

Figure 14.3. Top-level widgets appear in the Tk widget hierarchy just like internal widgets, but they are positioned on the screen independently from their parents in the hierarchy. In this example the dialog box `.dlg` is a top-level window. Figure (a) shows how the windows appear on the screen (with decorations provided by the `mwm` window manager) and Figure (b) shows Tk's widget hierarchy for the application.

14.3 Applications and processes

In Tk the term *application* refers to a single widget hierarchy (one main window and any number of internal and top-level windows descended from it), a single Tcl interpreter associated with the widget hierarchy, plus all the commands provided by that interpreter. Each application is usually a separate process, but Tk also allows a single process to manage several applications, each with its own widget hierarchy and Tcl interpreter. Tk does not provide any particular support for multi-threading (using a collection of processes to manage a single application); it is conceivable that Tk could be used in a multi-threaded environment but it would not be trivial and I know of no working examples.

14.4 Scripts and events

Tk applications are controlled by two kinds of Tcl scripts: an *initialization script* and *event handlers*. The initialization script is executed when the application starts up. It creates the application's user interface, loads the application's data structures, and performs any other initialization needed by the application. Once initialization is complete the application enters an *event loop* to wait for user interactions. Whenever an interesting event occurs, such as the user invoking a menu entry or moving the mouse, a Tcl script is invoked to process that event. These scripts are called event handlers; they can invoke application-specific Tcl commands (e.g. enter an item into a database), modify the user interface (e.g. post a dialog box), or do many other things. Some event handlers are created by the initialization script, but event handlers can also be created and modified by other event handlers.

Most of the Tcl code for a Tk application is in the event handlers and the procedures that they invoke. Complex applications may contain hundreds of event handlers, and the handlers may create other panels and dialogs that have additional event handlers. Tk applications are thus *event-driven*. There is no well-defined flow of control within the application's scripts, since there is no clear task for the application to carry out. The application presents a user interface with many features and the user decides what to do next. All the application does is to respond to the events corresponding to the user's actions. The event handlers implement the responses; they tend to be short scripts, and they are mostly independent of each other.

14.5 Wish: a windowing shell

While you're reading this book you may find it useful to experiment with a program called `wish` (for "windowing shell"). `Wish` is the simplest possible Tk application. The only Tcl commands it contains are the Tcl built-ins and the additional commands provided by Tk. If you invoke `wish` with no arguments then it creates a main window and acts like a shell, reading Tcl commands from its standard input and executing them. For example, try typing the following commands to `wish`:

```
button .b -text "Hello, world!" -command "destroy ."
pack .b
```

This creates the application shown in Figure 14.4, consisting of a single button that displays the text "`Hello, world`". It also creates one event handler: if the user clicks mouse button 1 over the widget then Tk will invoke the command "`destroy .`", which

Figure 14.4. A simple Tk application created by typing commands to `wish`.

destroys the application's main window and all its descendants and thereby causes `wish` to exit. `Wish` responds to events for the application's windows as well as to commands typed on its standard input.

You can also use `wish` to invoke scripts that have been saved in files. For example, you could create a file named `hello` that contains the above two commands. Then you could start up `wish` and type

```
source hello
```

to process the file. Or, you could invoke `wish` with the following shell command:

```
wish -f hello
```

In this case `wish` will not read commands from standard input. Instead, it will execute the script contained in the file `hello` and then enter an event loop where it responds only to events from the application's windows.

Wish scripts can also be invoked using the same mechanism that's used for shell scripts in UNIX. To do this, enter the following comment as the first line of `hello`:

```
#!/usr/local/bin/wish -f
```

Then mark the script file as executable. You can now invoke `hello` directly from the shell like any other executable program:

```
hello
```

This will run `wish` and cause it to process the script file just as if you'd typed "`wish -f hello`".

See the `wish` reference documentation for details on other features provided by `wish`, such as command-line arguments for `wish` scripts. If `wish` isn't installed in `/usr/local/bin` on your system then you'll need to use a different comment in your script files that reflects the location of `wish`.

14.6 Widget creation commands

Tk provides four main groups of Tcl commands; they create widgets, arrange widgets on the screen, communicate with existing widgets, and interconnect widgets within and between applications. This section and the three following sections introduce the groups of commands to give you a general feel for Tk's features. All of the commands are discussed in more detail in later chapters.

To create a widget, you invoke a command named after the widget's class: `button` for button widgets, `scrollbar` for scrollbar widgets, and so on.. For example, the following command creates a button that displays the text "`Press me`" in red:

```
button .b -text "Press me" -foreground red
```

All of the widget creation commands have a form similar to this. The command's name is the same as the name of the class of the new widget. The first argument is a name for the new widget in the widget hierarchy, `.b` in this case. This widget must not already exist but its parent must exist. The command will create the widget and its corresponding X window.

The widget name is followed by any number of pairs of arguments, where the first argument of each pair specifies the name of a *configuration option* for the widget (e.g. `-text` or `-foreground`) and the second argument specifies a value for that option (e.g. "`Press me`" or `red`). Each widget class supports a different set of configuration options but many options, such as `-foreground`, are used in the same way by different classes. You need not specify a value for every option supported by a widget; defaults will be chosen for the options you don't specify. For example, buttons support about twenty different options but only two were specified in the example above. Chapter 16 discusses configuration options in more detail.

14.7 Geometry managers

Widgets don't determine their own sizes and locations on the screen. This function is carried out by *geometry managers*. Each geometry manager implements a particular style of layout. Given a collection of widgets to manage and some controlling information about how to arrange them, a geometry manager assigns a size and location to each widget. For example, you might tell a geometry manager to arrange a set of widgets in a vertical column. It would then position the widgets so that they are adjacent but don't overlap. If one widget should suddenly need more space (e.g. its font is changed to a larger one) it will notify the geometry manager and the geometry manager will move other widgets down to preserve the proper column structure.

The second main group of Tk commands consists of those for communicating with geometry managers. Tk currently contains four geometry managers. The *placer* is a simple fixed-placement geometry manager. You give it instructions like "place window `.x` at location (10,100) in its parent and make it 2 cm wide and 1 cm high." The placer is simple to understand but limited in applicability because it doesn't consider interactions between widgets. Chapter 17 describes the placer in detail.

```
button .top -text "Top button"
pack .top
button .bottom -text "Bottom button"
pack .bottom
```

(a)

(b)

Figure 14.5. The script in (a) creates two button widgets and arranges them in a vertical column with the first widget above the second. The application's appearance on the screen is shown in (b).

The second geometry manager is called the *packer*. It is constraint-based and allows you to implement arrangements like the column example from above. It is more complex than the placer but much more powerful and hence more widely used. The packer is the subject of Chapter 18.

Two other geometry managers are implemented as part of the canvas and text widgets. The canvas geometry manager allows you to mix widgets with structured graphics, and the text geometry manager mixes widgets with text. See the reference documentation for canvas and text widgets for descriptions of these geometry managers.

When you invoke a widget creation command like `button` the new widget will not immediately appear on the screen. It will only be displayed after you have asked a geometry manager to manage it. If you want to experiment with widgets before reading the full discussion of geometry managers, you can make a widget appear by invoking the `pack` command with the widget's name as argument. For example, the following script creates a button widget and displays it on the screen:

```
button .b -text "Hello, world!"
pack .b
```

This will size the main window so that it is just large enough to hold the button and it will arrange the button so that it fills the space of the main window. If you create other widgets and pack them in a similar fashion, the packer will arrange them in a column inside the main window, making the main window just large enough to accommodate them all. See Figure 14.5 for an example.

14.8 Widget commands

Whenever a new widget is created Tk also creates a new Tcl command whose name is the same as the widget's name. This command is called a *widget command*, and the set of all widget commands (one for each widget in the application) constitutes the third major group of Tk's commands. Thus after the above `button` command was executed above, a widget command whose name is `.b` appeared in the application's interpreter. This command will exist as long as the widget exists; if the widget is deleted then the command will be deleted too.

Widget commands are used to communicate with existing widgets. Here are some commands that could be invoked after the `button` command from Section 14.6:

```
.b configure -foreground blue
.b flash
.b invoke
```

The first command changes the color of the button's text to blue, the second command causes the button to flash briefly, and the third command invokes the button just as if the user had clicked mouse button 1 on it. In widget commands the command name is the name of the widget and the first argument specifies an operation to invoke on the widget, such as `configure`. Some widget commands, like `configure`, take additional arguments; the nature of these arguments depends on the specific command.

The set of widget commands supported by a given widget is determined by its class. All widgets in the same class support the same set of commands, but different classes have different command sets. Some common commands are supported by multiple classes. For example, every widget class supports a `configure` widget command, which can be used to query and change any of the configuration options associated with the widget.

14.9 Commands for interconnection

The fourth group of Tk commands is used for interconnection. These commands are used to make widgets work together, to make them work cooperatively with the objects defined in the application, and to allow different applications sharing the same display to work together in interesting ways.

Some of the interconnection commands are implemented as event handlers. For example, each button has a `-command` option that specifies a Tcl script to invoke whenever mouse button 1 is clicked over the widget. This option was used in Section 14.5 to terminate the application. Scrollbars provide another example of interconnection via event handlers. Each scrollbar is used to control the view in some other widget: when you click in the scrollbar or drag its slider, the view in the associated widget should change. This connection between widgets is implemented by specifying a Tcl command for the scrollbar to invoke whenever the slider is dragged. The command invokes a widget command for the asscociated widget to change its view. In addition to event handlers that are defined by widgets, you can create custom event handlers using the `bind` command described in Chapter 19.

Tk supports five other forms of interconnection in addition to event handlers: the selection, the input focus, the window manager, the `send` command, and grabs. The selection is a distinguished piece of information on the screen, such as a range of text or a graphic. The X window system provides a protocol for applications to claim ownership of the selection and retrieve the contents of the selection from whichever application owns it. Chapter 20 discusses the selection in more detail and describes Tk's `select` command, which is used to manipulate it.

At any given time, keystrokes typed for an application are directed to a particular widget, regardless of the mouse cursor's location. This widget is referred to as the *focus widget* or *input focus*. Chapter 21 describes the `focus` command, which is used to move the focus among the widgets of an application.

Chapter 22 describes Tk's `wm` command, which is used for communicating with the window manager. The window manager acts as a geometry manager for main windows and top-level windows, and the `wm` command can be used to make specific geometry requests from the window manager, such as "don't let the user make this window smaller than 20 pixels across." In addition, `wm` can be used to specify a title to appear in the window's decorative border, a title and/or icon to display when the window is iconified, and many other things.

Chapter 23 describes the `send` command, which provides a general-purpose means of communication between applications. With `send`, you can issue an arbitrary Tcl command to any Tk application on the display; the command will be transmitted to the target application, executed there, and the result will be returned to the original application. `Send` allows one application to control another application in intimate and powerful ways. For example, a debugger can send commands to an editor to highlight the current line of execution, or a spreadsheet can send commands to a database application to retrieve new values for cells in the spreadsheet, or a mail reader can send commands to a video application to play a video clip identifying the sender of a message.

The last form of interconnection is *grabs*, which are described in Chapter 24. A grab restricts keyboard and mouse events so that they are only processed in a subtree of the widget hierarchy; windows outside the grab subtree become lifeless until the grab is released. Grabs are used to disable parts of an application and force the user to deal immediately with a high-priority window such as a dialog box.

Chapter 15
Tour Of The Tk Widgets

This chapter introduces the fifteen widget classes that are currently implemented by Tk. The descriptions are not intended to explain every feature of every class; for that you should refer to the reference documentation for the individual widget classes. In fact, no specific Tk commands will be mentioned in this chapter. This chapter gives an overview of the behavior of the widgets as seen by users and the features provided by the widgets to interface designers. The purpose of this chapter is to provide you with general information about the capabilities of Tk's widgets so that it will be easier to understand the specific commands described in later chapters.

The widget behavior described in this chapter is not hard-coded into the widgets. Instead, Tk contains a startup script that generates default behaviors for the widgets using the binding mechanism described in Chapter 19. The descriptions in this chapter correspond to the default behaviors, and most widgets in most applications will use the default behaviors. However, it is possible to extend or override the defaults, so some Tk applications may contain widgets that behave differently than described here.

If you have access to the `wish` program and the Tk demonstration scripts (both of which are included in the Tk distributions) then you can experiment with real widgets as you read through the chapter. To do this, execute the `widget` demonstration script and use the menus to bring up various examples.

15.1 Frames and toplevels

Frames and toplevels are the simplest widgets. They have almost no interesting properties. A frame appears as a rectangular region with a color and possibly a border that gives the

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

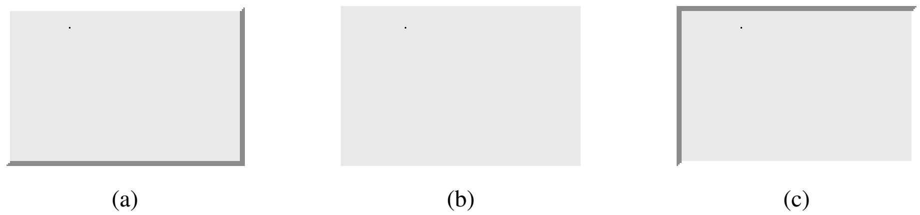


Figure 15.1. Frame and toplevel widgets have no visual characteristics except for a color and an optional three-dimensional border that can give the widget one of several appearances, such as raised as in (a), flat as in (b), or sunken as in (c).

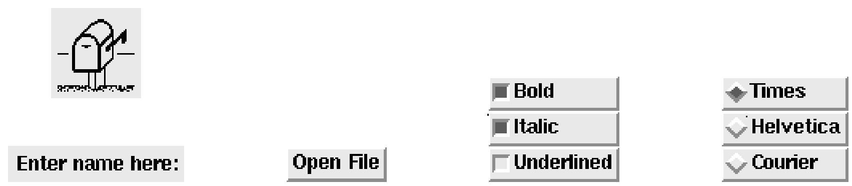
frame a raised or sunken appearance as shown in Figure 15.1. Frames serve two purposes. First, they can be used to generate decorations such as a block of color or a raised or sunken border around a group of widgets. Second, they serve as containers for grouping other widgets; most of the non-leaf widgets in the widget hierarchy are frames, and you'll see in Chapter 18 that frames are particularly important for building up nested layouts with geometry managers. When used in this way, frames are often invisible to the user. Frames do not normally respond to mouse or keyboard actions.

Toplevel widgets are identical to frames except that, as the name implies, they are top-level widgets whereas frames (and almost all other widgets) are internal widgets. This means that a toplevel widget can be positioned anywhere on its screen, independent of its parent in the widget hierarchy, and it need not even appear on the same screen as its parent. Toplevels are typically used as the outermost containers for panels and dialog boxes. When you create a toplevel you can specify a screen for it to be displayed on.

15.2 Labels, buttons, checkbuttons, and radiobuttons

Labels, buttons, checkbuttons, and radiobuttons make up a family of widget classes with similar characteristics. Each member of the family builds on the behavior of earlier members. Labels are the simplest member of the family. They are similar to frames except that each one can display a text string or a bitmap (see Figure 15.2). Like frames, labels do not normally respond to the mouse or keyboard; they simply provide decoration in the form of a text string or bitmap.

Buttons are similar to labels except that they also respond to the mouse. When the mouse cursor moves over a button, the button lights up. This indicates that pressing a mouse button will cause something to happen. It is a general property of Tk widgets that they light up if the mouse cursor passes over them when they are prepared to respond to



(a) (b) (c) (d)

Figure 15.2. Members of the label/button family of widgets. Two labels are shown in (a); the top one displays a bitmap and the bottom one displays a text string. Figure (b) shows a button widget. Three checkbuttons appear in (c); any combination of the checkbuttons may be selected at once. A group of three radiobuttons appears in (d); only one of the radiobuttons may be selected at anygiven time. Although a bitmap only appears in (a), any of the classes can display a bitmap as well as a string.

button presses. A button or other widget lit up in this way it is said to be *active*. Buttons become inactive again when the mouse cursor leaves them.

If mouse button 1 is pressed when a button is active then the button's appearance changes to make it look sunken, as if a real button had been pressed. When the mouse button is released, the widget's original appearance is restored. Furthermore, when the mouse button is released a Tcl script associated with the button is automatically executed. The script is a configuration option for the button.

Checkbuttons allow users to make binary choices such as enabling or disabling underlining or grid-alignment. They are similar to regular buttons except for two things. First, whenever mouse button 1 is clicked over a checkbutton a Tcl variable toggles between two values, one representing an "on" state and the other representing an "off" state. The name of the variable and the values corresponding to the "on" and "off" states are configuration options for the widget. Second, the checkbutton displays a small rectangular *selector* to the left of its text or bitmap. If the variable has the "on" value then the selector is displayed in a bright color and the button is said to be *selected*. If the variable has the "off" value then the selector box appears empty. Each checkbutton monitors the value of its associated variable and if the variable's value changes (e.g. because of a `set` command) the checkbutton updates the selector display.

The last member of the label/button family is the radiobutton class. Radiobuttons are typically arranged in groups and used to select one from among several mutually-exclusive choices, such as one of several colors or one of several styles of dashed lines. Radiobuttons are named after the radio selector buttons on older cars, where pressing the button for one station caused all the other buttons to be released. When mouse button 1 is clicked over a radiobutton, the widget sets the variable to the “on” value associated with that radiobutton. All of the radiobuttons in a group will share the same variable but each will have a different “on” value. A radiobutton displays a diamond-shaped selector to the left of its text or bitmap and lights up the selector when the widget is selected. Each radiobutton monitors its variable so if some other radiobutton resets the variable to select itself the previously-selected widget can turn off its selector diamond. If you change the value of the variable using the Tcl `set` command then all of the associated radiobuttons will redisplay their selectors to match the new value of the variable.

The members of the label/button family also have two additional features. First, you can specify that the string to be displayed in the widget should be taken from a Tcl variable. The widget will monitor the variable and update its display to reflect the current contents of the variable. Second, you can *disable* the widget. While a widget is disabled it is displayed in dimmer colors, it doesn’t activate when the mouse cursor passes over it, and it doesn’t respond to button presses.

15.3 Menus and menubuttons

Tk’s menu widget provides a general-purpose facility for implementing pull-down menus, pop-up menus, cascading menus, and many other things. A menu is a top-level widget that contains a collection of *entries* arranged in a column (see Figure 15.3(a)). Menu entries are not distinct widgets but they behave much like the members of the label/button family described in Section 15.2 above. The following types of entries may be used in menus:

Command: similar to a button widget. Displays a textual string or bitmap and invokes a Tcl script when mouse button 1 is released over it.

Checkbutton: similar to a checkbutton widget. Displays a string or bitmap and toggles a variable between “on” and “off” values when button 1 is released over the entry. Also displays a square selector indicating whether the variable is currently in its “on” or “off” state.

Radiobutton: similar to a radiobutton widget. Displays a string or bitmap and sets a variable to an “on” value associated with the enry when button 1 is released over it. Also displays a diamond-shaped selector indicating whether or not the variable has the value for this entry.

Cascade: similar to a menubutton widget. Posts a cascaded sub-menu when the mouse passes over it. See below for more details.

Separator: Displays a horizontal line for decoration. Does not respond to the mouse.

Unlike most other widgets, menus do not normally appear on the screen. They spend most of their time in an invisible state called *unposted*. When a user wants to invoke a menu entry, he or she *post*s the menu, which makes it appear on the screen. Then the user moves the mouse over the desired entry and releases button 1 to invoke that entry. Once

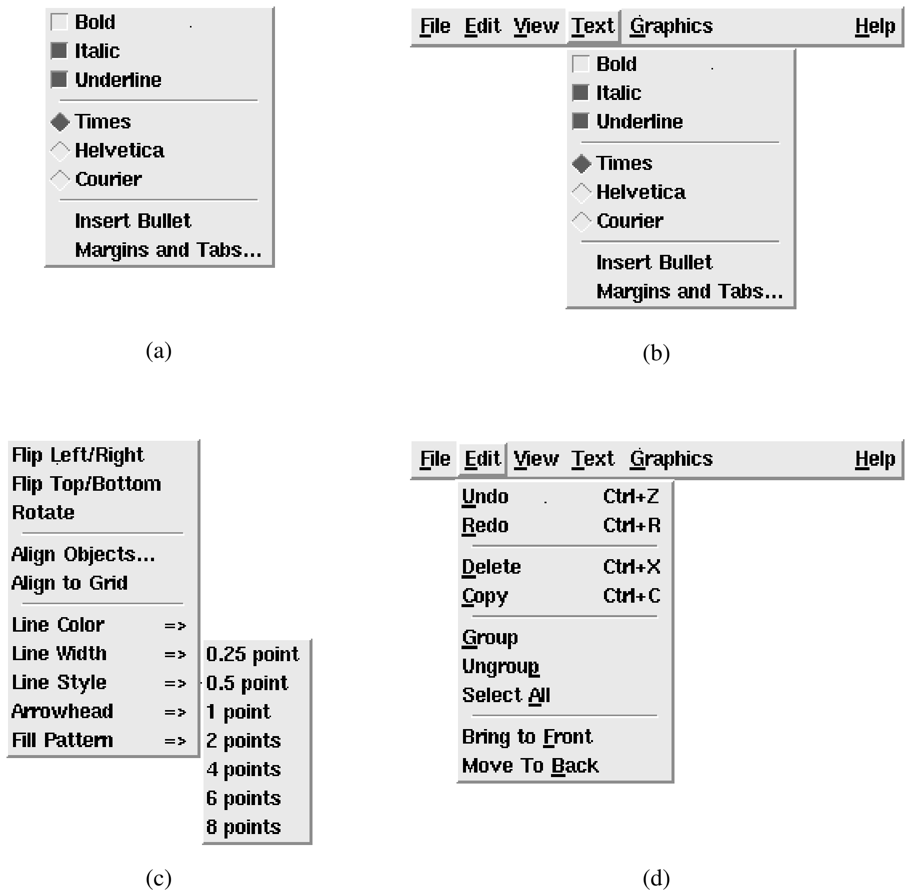


Figure 15.3. Examples of menus. Figure (a) shows a single menu with three checkbutton entries, three radiobutton entries, and two command entries. The groups of entries are separated by separator entries. Figure (b) shows the menu being used in pull-down fashion with a menu bar and several menubutton widgets. Figure (c) shows a cascaded series of menus; cascade entries in the parent (leftmost) menu display => at their right edges, and the `Line Width` entry is currently active. Figure (d) contains a menu that supports keyboard traversal and shortcuts. The underlined characters in the menubuttons and menu entries can be used to invoke them from the keyboard, and the key sequences at the right sides of some of the menu entries (such as `Ctrl+X`) can be used to invoke the same functions as menu entries without even posting the menu.

the menu has been invoked it is usually unposted until it is needed again. Menus are posted or unposted by invoking their widget commands, which gives the interface designer a lot of flexibility in deciding when to post and unpost them. The subsections below describe four of the most common approaches.

15.3.1 Pull-down menus

Menus are most commonly used in a *pull-down* style. In this style the application displays a *menu bar* near the top of its main window. A menu bar is a frame widget that contains several menubutton widgets as shown in Figure 15.3(b). Menubuttons are similar to button widgets except that instead of executing Tcl scripts when they are invoked they post menu widgets. When a user presses mouse button 1 over a menubutton it posts its associated menu underneath the menubutton widget. Then the user can slide the mouse down over the menu with the button still down and release the mouse button over the desired entry. When the button is released the menu entry is invoked and the menu is unposted. The user can release the mouse button outside the menu to unpost it without invoking any entry.

If the user releases the mouse button over the menubutton then the menu stays posted and the user will not be able to do anything else with the application until the menu is unposted either by clicking on one of its entries (which invokes that entry and unposts the menu) or clicking outside of the menu (which unposts the menu without invoking any entry). Situations like this where a user must respond to a particular part of an application and cannot do anything with the rest of the application until responding are called *modal* user interface elements. Menus and dialog boxes are examples of modal interface elements. Modal interface elements are implemented using the grab mechanism described in Chapter 24.

15.3.2 Pop-up menus

The second common style of menu usage is called *pop-up* menus. In this approach, pressing one of the mouse buttons in a particular widget causes a menu to post next to the mouse cursor and the user can slide the mouse over the desired entry and release it there to invoke the entry and unpost the menu. As with pull-down menus, releasing the mouse button outside the menu causes it to unpost without invoking any of its entries.

15.3.3 Cascaded menus

The third commonly used approach to posting menus is called *cascaded menus*. Cascaded menus are implemented using cascade menu entries in other menus, such as pull-down and pop-up menus. Each cascade menu entry is similar to a menubutton in that it is associated with a menu widget. When the mouse cursor passes over the cascade entry, its associated menu is posted just to the right of the cascade entry, as shown in Figure 15.3(c). The user can then slide the mouse to the right onto the cascaded menu and select an entry in the cascaded menu. Menus can be cascaded to any depth.

15.3.4 Keyboard traversal and accelerators

Pull-down menus can also be posted from the keyboard using a technique called *keyboard traversal*. One of the letters in each menubutton is underlined to indicate that it is the traversal character for that menubutton. If that letter is typed while holding the `Alt` key down then the menubutton's menu will be posted. Once a menu has been posted the arrow keys can be used to move among the menus and their entries. The left and right arrow keys move left or right among the menubuttons, unposting the menu for the previous menubutton and posting the menu for the new one. The up and down keys move among the entries in a menu, activating the next higher or lower entry. The `Return` key can be used to invoke the active menu entry. In addition, the labels in menu entries are typically drawn with one character underlined; if this character is typed when the menu is posted then the entry is invoked immediately.

Lastly, in many cases it is possible to invoke the function of a menu entry without even posting the menu by typing *keyboard shortcuts*. If there is a shortcut for a menu entry then the keystroke for the shortcut will be displayed at the right side of the menu entry (e.g. `Ctrl+X` is displayed in the `Delete` menu entry in Figure 15.3(d)). This key combination may be typed in the application to invoke the same function as the menu entry (e.g. type `x` while holding the `Control` key down to invoke the `Delete` operation without going through the menu).

15.4 Listboxes

A listbox is a widget that allows the user to select one or more possibilities from a range of alternatives, such as a file name from those in the current directory or a color from a database of defined colors. A listbox contains one or more entries, each of which displays a one-line string as shown in Figure 15.4. The widget commands for listboxes allow entries to be created, destroyed, and queried.

If there are more entries than there are lines in the listbox's window then only a few of them are displayed at a time; the user can control which portion is displayed by using a separate scrollbar widget associated with the listbox (see Section 15.6). The view in a listbox can also be controlled by pressing mouse button 2 in the widget and dragging up or down. This is called *scanning*: it has the effect of dragging the listbox contents past the window at high speed. Most Tk widgets that support scrollbars also support scanning. If the strings in the listbox are too long to fit in the window then the listbox can also be scrolled and scanned in the horizontal direction.

Typically listboxes are configured so that the user can select an entry by clicking on it with mouse button 1. In some cases the user can also select a range of entries by pressing and dragging with button 1. Selected entries appear in a different color and usually have a raised 3-D effect. Once the desired entries have been selected, the user will typically use those entries by invoking another widget, such as a button widget or menu entry. For

Figure 15.4. An example of a listbox widget displaying the names of all the states in the U.S.A. Only a few of the entries are visible in the window at one time. The `Ohio` entry is selected.

Figure 15.5. An example of an entry widget. The vertical bar is the insertion cursor ,which identifies the point at which new text will be inserted.

example, the user might select one or more file names from a listbox and then click on a button widget to delete the selected files; the Tcl command associated with the button widget can read out the strings from the selected listbox entries. It's also common for listboxes to support double-clicking, which both selects an entry and invokes some operation on it. For example, in a file-open dialog box, double-clicking on a file name might cause that file to be opened by the application.

15.5 Entries

An entry is a widget that allows the user to type in and edit a one-line text string. For example, if a document is being saved to disk for the first time then the user will have to provide a file name to use. The user might type the file name in an entry widget, then click on a button widget whose Tcl command retrieves the file name from the entry and saves the document in that file. Figure 15.5 shows an example of an entry widget.

To enter text into an entry the user clicks mouse button 1 in the entry. This makes a blinking vertical bar appear, called the *insertion cursor*. The user can then type characters

Figure 15.6. A horizontal scrollbar widget. The rectangular slider indicates how much of the document in an associated widget is visible in its window (in this case the rightmost 20% is visible). The user can adjust the view in the associated widget by dragging the slider with mouse button 1 or by clicking on the arrows or the slider region.

and they will be inserted into the entry at the point of the insertion cursor. The insertion cursor can be moved by clicking anywhere in the entry's text. Text in an entry can be selected by pressing and dragging with mouse button 1, and it can be edited with a variety of keyboard actions; see the reference documentation for details.

If the text for an entry is too long to fit in its window then only a portion of it is displayed and the view can be adjusted using an associated scrollbar widget or by scanning with mouse button 2. Entries can be disabled so that no insertion cursor will appear and the text in the entry cannot be modified. The text in an entry can be associated with a Tcl variable so that changes to the variable are reflected in the entry and changes made in the entry are reflected in the variable.

15.6 Scrollbars

Scrollbar widgets are used to control what is displayed in other widgets. Each scrollbar is associated with some other widget such as a listbox or entry. The scrollbar is typically displayed next to the other widget and when the user clicks and drags on the scrollbar the view in the associated widget will change. A scrollbar appears as shown in Figure 15.6 with an arrow at each end and a slider in the middle. The size and position of the slider correspond to the portion of the associated widget's document that is currently visible in its window. For example, if the slider covers the rightmost 20% of the region between the two arrows as in Figure 15.6 it means that the rightmost 20% of the document is visible in the window. Scrollbars can be oriented either vertically or horizontally.

Users can adjust the view by clicking mouse button 1 on the arrows, which moves the view a small amount in the direction of the arrow, or by clicking in the empty space on either side of the slider, which moves the view by one screenful in that direction. The view can also be changed by pressing on the slider and dragging it.

A scrollbar interacts with its associated widget using Tcl scripts. One of a scrollbar's configuration options is a Tcl script to invoke to change the view; typically this script invokes the widget command for the associated widget. When the user manipulates the

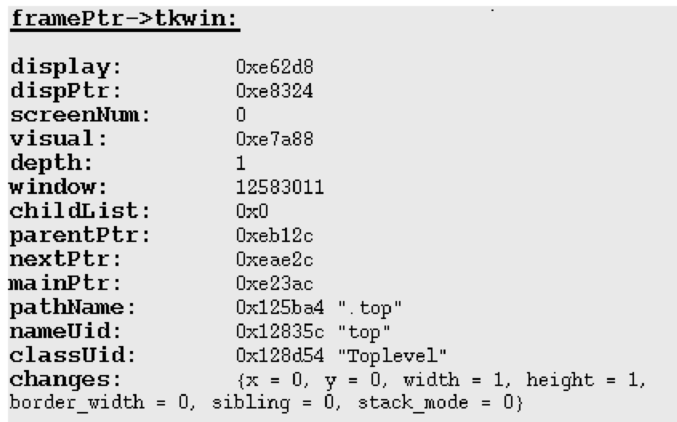


Figure 15.7. An example of a text widget. This widget displays the contents of a structure as part of a symbolic debugger. Tags are used to display field names in bold and to underline the name of the structure.

scrollbar, the scrollbar invokes the script, including additional information about the new view that the user requested. The associated widget changes its view and then invokes another Tcl script (one of its configuration options) that tells the scrollbar exactly what information is now displayed in the window, so the scrollbar can display the slider correctly. The scrollbar doesn't update its slider until told to do so by the associated widget; this makes it possible for the associated widget to reject or modify the user's request (e.g. to prevent the user from scrolling past the ends of the information in the widget).

15.7 Text

A text widget is similar to an entry except that it allows the text to span more than one line (see Figure 15.7 for an example). Text widgets are optimized to handle large amounts of text, such as files containing thousands of lines. As with entries, the user can click mouse button 1 to set the insertion cursor and then type new information into a text. Information in a text widget can be selected with the mouse just as for entries, and a number of mouse and keyboard actions are defined to assist in editing (see the reference documentation for details). Text widgets support scrolling and scanning, and they can be disabled to temporarily prevent edits.

In addition to the basic features described above, text widgets support three kinds of *annotations* on the text: marks, tags and embedded widgets. A mark associates a name with a particular position in the text (the gap between two adjacent characters). Marks are used to keep track of interesting locations in the text as characters are added and deleted.

A *tag* is a string that is associated with ranges of characters in a text widget. Each tag may be associated with any number of ranges of characters in the text, and the ranges of different tags may overlap. Tags are different from marks in that they are associated with particular characters, so they disappear when the characters are deleted. Tags are used for two purposes in texts: formatting and binding.

Each tag may contain formatting information such as background and foreground colors, font, and stippling and underlining information. If a character has been tagged then the formatting information in the tag overrides the default formatting information for the widget as a whole. This makes it possible to display text with multiple fonts and colors. In addition, the formatting information for a tag can be changed at any time. For example, you can apply a tag to all instances of a particular word in the text, then modify the tag's formatting information to make the words blink on and off.

The second use of tags is for *bindings*. A binding specifies a Tcl script to be invoked when certain events occur; each tag may have one or more bindings associated with it. For example, you can arrange for a script to be invoked whenever the mouse cursor passes over text with a particular tag, or whenever a mouse button is clicked over a particular item (see Chapter 19 for more information on bindings). This can be used to produce hypertext effects such as displaying a figure whenever the user clicks on the name of the figure in a text widget.

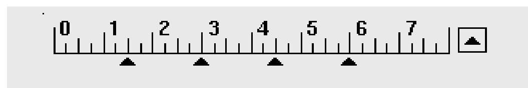
The third form of annotation in texts consists of embedded widgets. It is possible to embed other widgets in a text so that the other widgets are displayed at particular positions in the text. For example, you can arrange for a button widget to appear in a text widget as another way of getting hypertext-like capabilities, or you can embed canvas widgets to include figures inside texts, and so on.

Note: *Embedded widgets are not supported in Tk version 3.2.*

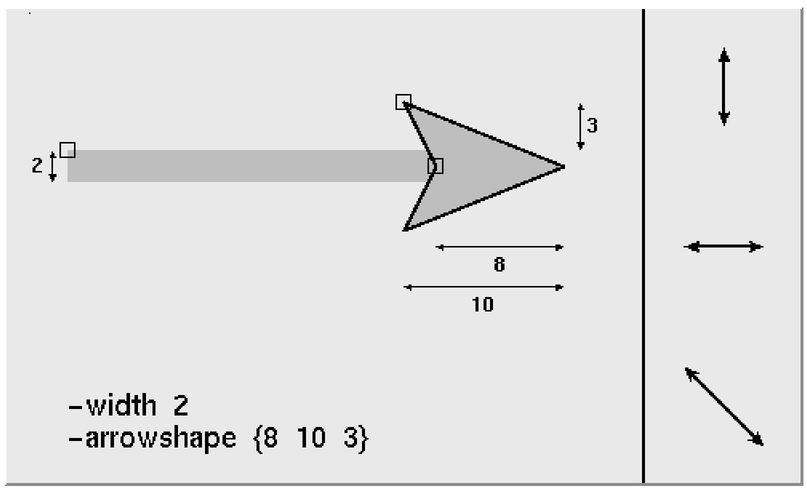
Text annotations allow you to configure a given text widget in a variety of interesting ways, so different text widgets may have very different behavior. For example, a file editor might use a text widget to display an entire file in a single font with no special formatting or bindings. In contrast, a debugger might use a text widget to display a structure as shown in Figure 15.7, where the names of the structure's fields are formatted differently than their values and bindings are set up so that the user can click on fields to open new windows on the structures pointed to by the fields.

15.8 Canvases

A canvas is a widget that displays a drawing surface and any number of graphical and textual *items*. The items can include rectangles, ellipses, arcs, lines, curves, polygons, curvagons, editable text, bitmaps, and embedded widgets. See Figure 15.8 for examples.



(a)



(b)

Figure 15.8. Canvas widget examples. Figure (a) shows a ruler with a tab well to the right. The user can create new tab stops by pressing mouse button 1 in the tab well and dragging out a new tab stop. Four existing tab stops appear underneath the ruler; they can be repositioned by dragging them with the mouse. Figure (b) shows an editor for arrowhead shapes. The user can edit the arrowhead shape and line width by dragging the three small squares attached to the oversized arrow. Changes to this shape are reflected in the normal-size arrows on the right side of the canvas, in the dimensions displayed next to the oversize arrow, and in the configuration option strings in the bottom left corner.

Items can be created and deleted at any time, and their display attributes (such as line width and color) can also be modified dynamically. Items can be moved and scaled but rotations are not currently supported.

Canvases also provide a tagging mechanism similar to the tags in text widgets. Each item may have any number of textual tags associated with it. Tags serve two purposes in canvases. First, they make it easy to operate on groups of items all at once; for example, in a single command you can move or delete or recolor all items with a given tag. Second, tags can have bindings associated with them just as in texts. This allows you to achieve

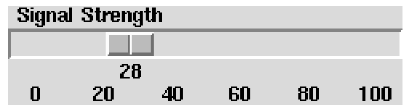


Figure 15.9. A scale widget. The scale's value can be adjusted by dragging the slider with the mouse.

hypergraphic effects such as invoking some operation whenever a mouse button is clicked over an item, or allowing some items to be dragged with the mouse.

As with texts, the features provided by canvases are flexible enough to achieve many different effects, so different canvases may appear and behave very differently. Canvases can be used to provide non-interactive graphical displays, such as pie-charts or figures, or they can be used to create new kinds of editors and interactive widgets.

15.9 Scales

A scale is a widget that displays a numerical value and allows the user to edit the value (see Figure 15.9). A scale widget appears as a linear scale with optional numerical labels and a slider that shows the current value. The user can adjust the value by clicking mouse button 1 in the scale or by dragging the slider with mouse button 1. Each scale can be configured with a Tcl script to invoke whenever its value changes; the script can propagate the new value to other parts of the application. For example, three scales might be used to edit the hue, saturation, and intensity values for a color; as the user modifies the scale values, the new values can be used to update the color for an item in a canvas so that the item is always displayed in the color selected by the scales.

15.10 Messages

A message widget displays a multi-line string of text like the one shown in Figure 15.10. Messages are less powerful than texts (e.g. they don't allow their text to be selected or edited, they don't provide annotations, they don't support scrolling, and they don't handle large amounts of text efficiently), but they are simpler to create and configure. Messages are typically used for simple things like multi-line messages in dialog boxes.

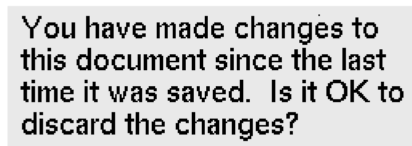


Figure 15.10. A message widget displays a string, breaking it into multiple lines if necessary. Messages provide little other functionality (e.g. no edit capability).

Chapter 16
Configuration Options

Most of the state of a widget exists as a set of *configuration options* for the widget. For example, the colors and font and text for a button widget are configuration options, as is the Tcl script to invoke when the user clicks on the button. Each configuration option has a name (e.g., `-relief`) and a value (e.g. `raised`). Widgets typically have 15-30 configuration options. For widgets such as texts and canvases that have complex internal structures, the configuration options don't provide complete access to the internal structures; special widget commands exist for this purpose. However, state that is shared among all the objects in the internal structures (such as a default font for text widgets) is still represented as configuration options.

This chapter describes Tk's mechanisms for dealing with configuration options. Section 16.1 gives an overview of how the values of options are set. Sections 16.2-16.11 describe some of the common configuration options that are used in the Tk widget set. Finally, Sections 16.12 and 16.13 explain the `configure` widget command and the option database in more detail. Table 16.1 summarizes the commands for manipulating configuration options. For a complete list of the options available for a given class, see the reference documentation for the command that creates widgets of that class (e.g. the `button` command)

16.1 How options are set

Configuration options may be specified in four ways. First, you can specify configuration options in the command that creates a widget. For example, the command

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

`class window ?optionName value optionName value ...?`	Create a new widget with class `class` and path name `window`, and set options for the new widget as given by `optionName`-`value` pairs. Unspecified options are filled in using the option database or widget defaults. Returns `window` as result.
`window config`	Returns a list whose elements are sublists describing all of the options for `window`. Each sublist describes one option in the form described below.
`window config optionName`	Returns a list describing option `optionName` for `window`. The list will normally contain five values: `optionName`, the option's name in the option database, its class, its default value, and its current value. If the option is a synonym for another option, then the list contains two values: the option name and the database name for the synonym.
`window config optionName value`	Set the value for option `optionName` of `window` to value.
`option add pattern value ?priority?`	Add a new option to the option database as specified by `pattern` and `value`. `Priority` must be either a number between `0` and `100` or a symbolic name (see the reference documentation for details on symbolic names).
`option clear`	Remove all entries from the option database.
`option get window name class`	If the option database contains a pattern that matches `window`, `name`, and `class`, return the value for the highest priority matching pattern. Otherwise return an empty string.
`option readfile fileName ?priority?`	Read `fileName`, which must have the standard format for a `.Xdefaults` file, and add all the options specified in that file to the option database at priority level `priority`.

Table 16.1. The commands for manipulating widget configuration options.

```
button .help -text Help -foreground red
```

creates a new button widget and specifies the `-text` and `-foreground` options for it. Every widget creation command has this form, where the command name is the name of the widget class, the first argument is the name of the new widget in the Tk widget hierarchy, and additional arguments (if any) are name-value pairs specifying options.

The second way to specify configuration options is through the *option database*. If no value is given for a configuration option on the command line that creates a widget, then Tk checks the option database to see if a value has been specified for the option. The option database is similar to the resource database in other X toolkits. It allows users to specify values for options in the `RESOURCE_MANAGER` property on the root window or in a `.Xdefaults` file. Entries in the database can contain wildcard characters so that, for example, a single entry in the option database can set the background color for all buttons to blue. See Section 16.13 for more information on the option database.

The third way that configuration options are specified is through default values for each widget class. Class defaults are used for options that aren't specified in the widget creation command and aren't defined in the option database. The class defaults are intended to produce a reasonable effect so that you don't need to specify most options either on the command line or in the option database. The class defaults are compiled into the Tk library so you can't change them without recompiling Tk, but you can always override them with values in the option database.

The final way to specify configuration options for a widget is with its `configure` widget command. Every widget class supports a `configure` widget command. For example, the following command changes the text in the button widget created above and also specifies a Tcl script to invoke when the user clicks on the widget:

```
.help configure -text Quit -command exit
```

The `configure` widget command allows you to change the configuration options for a widget at any time and it also allows you to query the current state of the configuration options (see Section 16.12 for details on this).

16.2 Colors

Although each widget class defines its own set of configuration options, the options tend to be used in a consistent fashion by different classes. This section and the ones that follow provide an overview of the most common options. These options have the same names and legal values in many different widget classes.

The most common options are those for specifying colors. Every widget class supports a `-background` option, which determines the background color of the widget and is also used to compute the light and dark shadows if there is a 3D border drawn around the widget. Nearly every widget class also supports a `-foreground` option, which is used when displaying text and graphics in the widget. Table 16.2 lists all of the common color options.

Color values may be specified either symbolically or numerically. A symbolic color value is a name such as `white` or `red` or `SeaGreen2`. The valid color names are defined in a file named `rgb.txt` in your X library directory. Common names such as `black` and `white` and `red` should be defined in every X environment, but names like `SeaGreen2` might not be available everywhere. Color names are not case-sensitive: `black` is the same as `Black` or `bLaCk`.

Colors can also be specified numerically in terms of their red, green, and blue components. Four forms are available, in which the components are specified with 4-bit, 8-bit, 12-bit, or 16-bit vales:

Name on Command Line	Usage
`-background`	Background areas of widgets.
`-foreground`	Text and graphics.
`-activebackground`	Background color when widget is active (mouse cursor is over widget and pressing a mouse button will invoke some action).
`-activeforeground`	Foreground color when widget is active.
`-selectbackground`	Background color for areas occupied by selected information within widget.
`-selectforeground`	Foreground color for selected text and graphics.
`-insertbackground`	Color for insertion cursor.
`-disabledforeground`	Foreground color when widget has been disabled.

Table 16.2. Commonly-used color options. The left column gives the name of the option as specified in widget creation commands and `configure` widget commands. The right column describes how the option is used.

```
#RGB
#RRGGBB
#RRRGGGBBB
#RRRRGGGGBBBB
```

Each `R`, `G`, or `B` in the above examples represents one hexadecimal digit of red, green, or blue intensity, respectively. The first character of the specification must be #, and the same number of digits must be provided for each component. If fewer than 16 bits are given for the color components, they represent the most significant bits of the values. For example, `#3a7` is equivalent to `#3000a0007000`. A value of all ones represents “full on” for that color, and a value of zero represents “off.” Thus `#000` is black, `#f00` is red, `#ff0` is yellow, and `#fff` is white.

If you specify a color other than black or white for a monochrome display, then Tk will use black or white instead, depending on the overall intensity of the color you requested. Furthermore, if you are using a color display and all of the entries in its color map are in use (e.g. because you’re displaying a complex image on the screen) then Tk will treat the display as if it were monochrome.

Name on Command Line	Usage
`-borderwidth`	Width of 3D border drawn around widget.
`-activeborderwidth`	Width of 3D border drawn around active elements within widget.
`-selectborderwidth`	Width of 3D border drawn around selected text.
`-insertwidth`	Total width of insertion cursor including its border, if any.
`-insertborderwidth`	Width of 3D border for insertion cursor.
`-padx`	Additional space to leave on left and right sides of information displayed in widget.
`-pady`	Additional space to leave above and below information displayed in widget.

Table 16.3. Common options for specifying distances. The left column gives the name of the option as specified in widget creation commands and `configure` widget commands. The right column describes how the option is used.

16.3 Screen distances

Several options are used to specify distances on the screen. The most common of these options is `-borderwidth`, which determines the width of the 3D border drawn around a widget. Every widget class supports the `-borderwidth` option. Table 16.3 lists several other common distance options.

Ultimately, each distance option must reduce to a distance in screen pixels. However, Tk allows distances to be specified either in pixels or in absolute units that are independent of the screen resolution. A distance is specified as an integer or floating-point value followed optionally by a single character giving the units. If no unit specifier is given then the units are pixels. Otherwise the unit specifier must be one of the following characters:

`c`	centimeters
`i`	inches
`m`	millimeters
`p`	printer's points (1/72 inch)

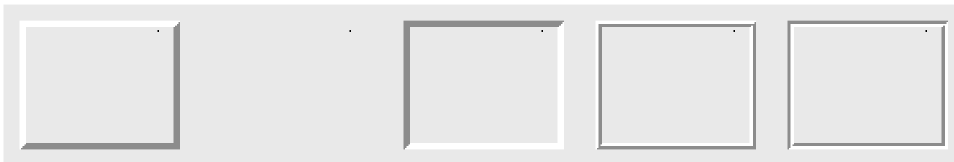


Figure 16.1. The three-dimensional effects produced by different values for the `-relief` option.

For example, a distance specified as `2.2c` will be rounded to the number of pixels that most closely approximates 2.2 centimeters; this may be a different number of pixels on different screens.

16.4 Reliefs

Every widget class supports an option named `-relief`, which determines the three-dimensional appearance of the widget. The option must have one of the values `raised`, `flat`, `sunken`, `ridge`, or `groove`. Figure 16.1 illustrates the effect produced by each value. Tk draws widget borders with combinations of light and dark shadows to produce the different effects. For example, if a widget's relief is `raised` then Tk draws the top and left borders in a lighter color than the widget's background and it drawns the lower and right borders in a darker color. This makes the widget appear to protrude from the screen.

The width of a widget's 3D border is determined by its `-borderwidth` option. If the border width is 0 then the widget will appear flat regardless of its `-relief` option.

16.5 Fonts

The `-font` option is used to specify a font for widgets that display text, such as buttons, listboxes, entries, and texts. Tk uses standard X font names, which are illustrated in Figure 16.2 The name of a font consists of twelve fields separated by hyphens. The fields have the following meanings:

foundry	The type foundry that supplied the font data.
family	Identifies a group of fonts with a similar typeface design.

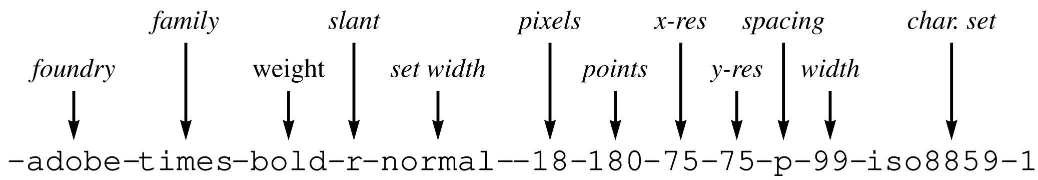


Figure 16.2. The fields of an X font name.

weight	Typographic weight of font, such as `medium`, `normal`, or `bold`.
slant	Posture of font, such as `r` for roman or upright, `i` for italic, or `o` for oblique.
set width	Proportionate width of font, such as `normal` or `con-densed` or `narrow`.
pixels	Size of font in pixels.
points	Size of font in tenths of points, assuming screen has x-res and y-res specified for font.
x-res	Horizontal resolution of screen for which font was designed, in dots per inch.
y-res	Vertical resolution of screen for which font was designed, in dots per inch.
spacing	Escapement class of font, such as `m` for monospace (fixed-width) or `p` for proportional (variable-width).
width	Average width of characters in font, in tenths of pixels.
char. set	Character set that identifies the encoding of characters in the font.

When `-font` values you can use `*` and `?` wildcards: `?` matches any single character in a font name, and `*` matches any group of characters. For example, the font name

```
*-times-medium-r-normal--*-100-*
```

requests a 10-point Times Roman font in a medium (normal) weight and normal width. It specifies “don’t care” for the foundry, the pixel size, and all fields after the point size. If multiple fonts match this pattern then the X server will pick one of them. I recommend specifying the point size for fonts but not the pixel size, so that characters will be the same size regardless of the display resolution.

Figure 16.3. Bitmaps defined internally by Tk.

16.6 Bitmaps

Many widgets, such as labels and menubuttons, can display *bitmaps*. A bitmap is an image with two colors, foreground and background. Bitmaps are specified using the `-bitmap` option, whose values may have two forms. If the first character of the value is `@` then the remainder of the value is the name of a file containing a bitmap in the standard X11 bitmap file format. Such files are generated by the `bitmap` program, among others. Thus "`-bitmap @face.bit`" specifies a bitmap contained in the file `face.bit`.

If the first character of the value isn't `@` then the value must be the name of a bitmap defined internally. Tk defines several internal bitmaps itself (see Figure 16.3) and individual applications may define additional ones.

The `-bitmap` option only determines the pattern of 1's and 0's that make up the bitmap. The foreground and background colors used to display the bitmap are determined by other options (typically `-foreground` and `-background`). This means that the same bitmap can appear in different colors at different places in an application, or the colors of a given bitmap may be changed by modifying the options that determine them.

16.7 Cursors

Every widget class in Tk supports a `-cursor` option, which determines the image to display in the mouse cursor when it is over that widget. If the `-cursor` option isn't specified or if its value is an empty string then the widget will use its parent's cursor. Otherwise the value of the `-cursor` option must be a proper Tcl list with one of the following forms:

```
name fgColor bgColor
name fgColor
```

```
name
@sourceFile maskFile fgColor bgColor
@sourceFile fgColor
```

In the first three forms *`name`* refers to one of the cursors in the standard X cursor font. You can find a complete list of all the legal names in the X include file `cursorfont.h`. The names in that file all start with `XC_`, such as `XC_arrow` or `XC_hand2`; when using one of these names in a `-cursor` option, omit the `XC_`, e.g. `arrow` or `hand2`. Most of the Xlib reference manuals also include a table showing the names and images of all the cursors in the X cursor font; for example, see Appendix B of *X Window System: The Complete Reference to Xlib, X Protocol, ICCM, and XLFD*, by Scheifler and Gettys, Second Edition. If *`name`* is followed by two additional list elements as in the following widget command:

```
.f config -cursor {arrow red white}
```

then the second and third elements give the foreground and background colors to use for the cursor; as with all color values, they may have any of the forms described in Section 16.2. If only one color value is supplied then it gives the foreground color for the cursor; the background will be transparent. If no color values are given then black will be used for the foreground and white for the background.

If the first character in the `-cursor` value is `@` then the image(s) for the cursor are taken from files in bitmap format rather than the X cursor font. If two file names and two colors are specified for the value, as in the following widget command:

```
.f config -cursor {@cursors/bits cursors/mask red white}
```

then the first file is a bitmap that contains the cursor's pattern (1's represent foreground and 0's background) and the second file is a mask bitmap. The cursor will be transparent everywhere that the mask bitmap has a 0 value; it will display the foreground or background wherever the mask is 1. If only one file name and one color are specified then the cursor will have a transparent background.

16.8 Anchors

An *anchor position* indicates how to attach one object to another. For example, if the window for a button widget is larger than needed for the widget's text, a `-anchor` option may be specified to indicate where the text should be positioned in the window. Anchor positions are also used for other purposes, such as telling a canvas widget where to position a bitmap relative to a point or telling the packer geometry manager where to position a window in its frame.

Anchor positions are specified using one of the following points of the compass:

`n`	Center of object's top side.
`ne`	Top right corner of object.

`e`	Center of object's right side.
`se`	Lower right corner of object.
`s`	Center of object's bottom side.
`sw`	Lower left corner of object.
`w`	Center of object's left side.
`nw`	Top left corner of object.
`center`	Center of object.

The anchor position specifies *the point on the object by which it is to be attached*, as if a push-pin were stuck through the object at that point and then used to pin the object someplace. For example, if a `-anchor` option of `w` is specified for a button, it means that the button's text or bitmap is to be attached by the center of its left side, and that point will be positioned over the corresponding point in the window. Thus `w` means that the text or bitmap will be centered vertically and aligned with the left edge of the window. For bitmap items in canvas widgets, the `-anchor` option indicates where the bitmap should be positioned relative to a point associated with the item; in this case, `w` means that the center of the bitmap's left side should be positioned over the point, so that the bitmap actually lies to the east of the point. Figure 16.4 illustrates these uses of anchor positions.

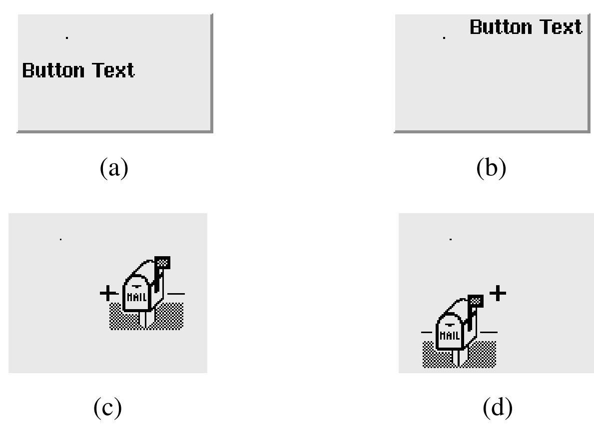


Figure 16.4. Examples of anchor positions used for button widgets and for bitmap items within canvases. Figure (a) shows a button widget with text anchored `w`, and (b) shows the same widget with an anchor position of `ne`. Figure(c) shows a canvas containing a bitmap with an anchor position of `w` relative to its point (the point appears as a cross, even though it wouldn't appear in an actual canvas). Figure (d) shows the same bitmap item with an anchor point of `ne`.

16.9 Script options and scrolling

Script options are used in many places in Tk widgets. The most common usage is for widgets like buttons and menus that are supposed to take action when invoked by the user. This is handled by specifying a Tcl script as a configuration option for the widget. For example, button widgets support a `-command` option, which should contain a Tcl script. When the user invokes the widget by clicking over it with the mouse button, the widget causes the script to be executed. Similarly, each entry in a menu widget has a script associated with it, which is executed when the user invokes the menu entry.

Script options are also used for communicating between widgets. Typically, one widget will be configured with *part* of a Tcl command (e.g. the name of another widget's widget command and the first argument to that command). At appropriate times, the widget will invoke the command. Before invoking the command the widget will augment it with additional information that is relevant to the specific invocation. The best example of this is the communication between scrollbars and other widgets, which is described in the rest of this section.

When a scrollbar is associated with another widget and used to change its view, the communication between the scrollbar and the associated widget is controlled by two options, one for the associated widget and one for the scrollbar. In normal usage, each of these options invokes a widget command for the other widget.

The associated widget must inform the scrollbar about what it is currently displaying, so that the scrollbar can display the slider in the correct position. To do this, the scrollbar provides a widget command of the following form:

```
window set totalUnits windowUnits first last
```

`Window` is the name of the scrollbar widget (i.e. the name of the widget command for the scrollbar). *`TotalUnits`* indicates the total size of the information being displayed in the associated widget in the dimension being scrolled, such as the number of lines in a listbox or the number of characters in a text entry. *`WindowUnits`* indicates how much of the information can be displayed in the widget at one time given the current size of its window, and *`first`* and *`last`* give the indices of the top and bottom elements currently visible in the widget's window (for horizontal scrollbars *`first`* and *`last`* refer to the leftmost and rightmost visible elements).

The associated widget invokes the scrollbar's `set` command whenever information of interest to the scrollbar changes in the widget. To do this, scrollable widgets provide a `-xScrollCommand` option if they support horizontal scrolling and a `-yScrollCommand` option if they support vertical scrolling. For example, a listbox might be created with a vertical scrollbar using the following commands:

```
listbox .l -yscrollcommand {.vscroll set}
scrollbar .vscroll -orient vertical
pack .l -side left
pack .vscroll -side right
```

The value of the `-yscrollcommand` option is a Tcl command prefix. When the view in the listbox changes (e.g. because elements were deleted), the listbox takes the value of the `-yscrollcommand` option ("`.vscroll set`" in this case) and appends four integer values corresponding the the `totalUnits`, `windowUnits`, `first`, and `last` arguments described above. This will produce a Tcl command such as

```
.vscroll set 100 20 38 57
```

Then the listbox invokes the command, which causes the scrollbar to redraw its slider to reflect the new view. If horizontal scrolling is desired for the listbox as well, an additional scrollbar could be created and a `-xscrollcommand` option could be specified for the listbox.

A similar form of communication is used by the scrollbar to notify the associated widget when the user manipulates the scrollbar to request a new view. Each scrollbar provides a `-command` option, which specifies a Tcl command prefix for communicating new views to the associated widget. It can be set for the `.vscroll` widget above using the following command:

```
.vscroll config -command {.l yview}
```

Then when the user clicks in the scrollbar to change the view the scrollbar takes the `-command` option and appends the index of the element that should now appear at the top of the window. The result is a command like the following:

```
.l yview 39
```

The scrollbar widget then invokes this command. Listboxes and other widgets that support scrolling provide a `yview` widget command with exactly the above syntax that causes the widget to adjust its view. After adjusting its view, the listbox uses its `-yscrollcommand` option to notify the scrollbar of the new view so the scrollbar can redraw its slider.

This scheme has the advantage that neither widget needs any built-in information about the other; both the name of the other widget and the widget command to invoke are provided with options that can be configured by the application designer. In fact, the command options need not even correspond to widget commands. For example, a single scrollbar could be made to control two widgets simultaneously by using a Tcl procedure name as its `-command` option:

```
.vscroll config -command scrollProc
proc scrollProc index {
        .l yview $index
        .l2 yview $index
}
```

Then the commands invoked by the scrollbar will look like

```
scrollProc 39
```

and `scrollProc` will invoke `yview` widget commands in each of the two associated widgets.

16.10 Variables

Another common form for options is variable names. These options are used to associate one or more Tcl global variables with a widget so that the widget can set the variable under certain conditions or monitor its value and react to changes in the variable.

For example, many of the widgets that display text, such as labels and buttons and messages and entries, support a `-textvariable` option. The value of the option is the name of a global variable that contains the text to display in the widget. The widget monitors the value of the variable and updates the display whenever the variable changes value. In addition, for widgets like entries that can modify their text, the widget updates the variable to track changes made by the user.

Checkbuttons and radiobuttons also support a `-variable` option, which contains the name of a global variable. For checkbuttons there are two additional options (`-onvalue` and `-offvalue`) that specify values to store in the variable when the checkbutton is "on" and "off." As the user clicks on the checkbutton with the mouse, it updates the variable to reflect the checkbutton's state. The checkbutton also monitors the value of the variable and changes its on/off state if the variable's value is changed externally. Each checkbutton typically has its own variable.

With radiobuttons a group of widgets shares the same variable but each radiobutton has a distinct value that it stores into the variable (the `-value` option). When the user clicks on a radiobutton it sets the variable to its value and selects itself. The radiobutton monitors the variable so that it can deselect itself when some other radiobutton stores a different value into the variable. If the variable's value is changed externally then all of the radiobuttons associated with the variable update their selected/deselected state to reflect the variable's new value.

16.11 Time intervals

Several widget classes provide options that specify time intervals, such as the blink rate for the insertion cursor or the rate at which mouse buttons should auto-repeat. Table 16.4 summarizes the most commonly used options for specifying intervals. Time intervals are always specified as integer numbers of milliseconds: an interval of `100` means 100ms, `1000` means one second, and so on.

16.12 The configure widget command

Every widget class supports a `configure` widget command. This command comes in three forms, which can be used both to change the values of options and also to retrieve information about the widget's options. See Table 16.1 for a summary of these forms.

Name on Command Line	Usage
`-insertoffTime`	How long to leave insertion cursor turned off in each blink cycle. Zero means cursor doesn't blink.
`-insertOnTime`	How long to leave insertion cursor turned on in each blink cycle.
`-repeatDelay`	How long to wait before auto-repeating a button or keystroke.
`-repeatInterval`	Once auto-repeat starts, how long to wait from one auto-repeat to the next.

Table 16.4. Commonly-used time interval options. The left column gives the name of the option as specified in widget creation commands and `configure` widget commands. The right column describes how the option is used.

If `configure` is given two additional arguments then it changes the value of an option as in the following example:

```
.button configure -text Quit
```

If the `configure` widget command is given just one extra argument then it returns information about the named option. The return value is normally a list with five elements:

```
.button configure -text
-text text Text { } Quit
```

The first element of the list is the name of the option as you'd specify it on a Tcl command line when creating or configuring a widget. The second and third elements are a name and class to use for looking up the option in the option database (see Section 16.13 below). The fourth element is the default value provided by the widget class (a single space character in the above example), and the fifth element is the current value of the option.

Some widget options are just synonyms for other options (e.g. the `-bg` option for buttons is the same as the `-background` option). Configuration information for a synonym is returned as a list with two elements consisting of the option's command-line name and the option database name of its synonym:

```
.button configure -bg
-bg background
```

If the `configure` widget command is invoked with no additional arguments then it returns information about all of the widget's options as a list of lists with one sub-list for each option:

```
.button configure
{-activebackground activeBackground Foreground Black 
Black} {-activeforeground activeForeground Background 
White White} {-anchor anchor Anchor center center} 
{-background background Background White White} {-bd 
borderWidth} {-bg background} {-bitmap bitmap Bitmap {} 
{}} {-borderwidth borderWidth BorderWidth 2 2} {-command 
command Command {} {}} {-cursor cursor Cursor {} {}} 
{-disabledforeground disabledForeground 
DisabledForeground {} {}} {-fg foreground} {-font font 
Font -Adobe-Helvetica-Bold-R-Normal-*-120-* -Adobe-
Helvetica-Bold-R-Normal-*-120-*} {-foreground 
foreground Foreground Black Black} {-height height 
Height 0 0} {-padx padX Pad 1 1} {-pady padY Pad 1 1} 
{-relief relief Relief raised raised} {-state state 
State normal normal} {-text text Text { } Quit} 
{-textvariable textVariable Variable {} {}} {-width 
width Width 0 0}
```

16.13 The option database

The option database supplies values for configuration options that aren't specified explicitly by the application designer. The option database is consulted when widgets are created: for each option not specified on the command line, the widget queries the option database and uses the value found there, if any. If there is no value in the option database then the widget supplies a default value. Values in the option database are usually provided by the user to personalize applications, e.g. by using consistently larger fonts. Tk supports the `RESOURCE_MANAGER` property and `.Xdefaults` file in the same way as other X toolkits like Xt.

16.13.1 Patterns

The option database contains any number of entries, where each entry consists of two strings: a *pattern* and a *value*. The pattern determines whether the entry applies to a given option for a given widget, and the value is a string to use for options that match the pattern.

In its simplest form, a pattern consists of an application name, a window name, and an option name, all separated by dots. For example, here are two options in this form:

```
wish.a.b.foreground
wish.background
```

The first pattern applies to the `foreground` option in the window `.a.b` in the application `wish`, and the second pattern applies to the `background` option in the main window for `wish`. Each of these patterns applies to only a single option for a single widget.

Patterns may also contain classes or wildcards, which allow them to match many different options or widgets. Any of the window names in the pattern may be replaced by a class, in which case the pattern matches any widget that is an instance of that class. For example, the pattern below applies to all children of `.a` that are buttons:

```
wish.a.Button.foreground
```

Application and option names may also be replaced with classes. The class for an application is the class of its main window; names and classes for applications are discussed in more detail in Chapter 22. Individual options also have classes. For example, the class for the `foreground` option is `Foreground`. Several other options, such as `activeBackground` and `insertBackground`, also have the class `Foregound`, so the following pattern applies to any of these options for any button widget that is a child of `.a` in `wish`:

```
wish.a.Button.Foreground
```

Lastly, patterns may contain `*` wildcard characters. A `*` matches any number of window names or classes, as in the following examples:

```
*Foreground
wish*Button.foreground
```

The first pattern applies to any option in any widget of any application as long as the option's class is `Foreground`. The second pattern applies to the `foreground` option of any button widget in the `wish` application. The `*` wildcard may only be used for window or application names; it cannot be used for the option name (it wouldn't make much sense to specify the same value for all options of a widget).

This syntax for patterns is the same as that supported by the standard X resource database mechanisms in the X11R3 and X11R4 releases. The `?` wildcard, which was added in the X11R5 release, is not yet supported by Tk's option database.

In order to support the above matching rules, each option has three names:

1. the name that can be typed on a command line, which always starts with a − and has no upper-case letters, as in `-activeborderwidth`;
2. the name of the option in the database, which is typically the same as the command-line name except that it contains no − and uses capital letters to mark internal word boundaries, as in `activeBorderWidth`;
3. the class of the option, which always starts with a capital letter and may contain additional capital letters to mark internal boundaries, as in `BorderWidth`.

When you query an option with the `configure` widget command all three of these names are returned. It's important to remember that in Tk classes *always* start with an initial capital letter, and any name starting with an initial capital letter is assumed to be a class.

16.13.2 RESOURCE_MANAGER property and .Xdefaults file

When a Tk application starts up, Tk automatically initializes the option database. If there is a RESOURCE_MANAGER property on the root window, then the database is initialized from it. Otherwise Tk checks the user's home directory for a .Xdefaults file and uses it if it exists. The initialization information has the same form whether it comes from the RESOURCE_MANAGER property or the .Xdefaults file. The syntax described below is the same as that supported by other toolkits such as Xt.

Each line of the initialization data specifies one entry in the resource database in a form like the following:

```
*Foreground: blue
```

The line consists of a pattern (*Foreground in the example) followed by a colon followed by whitespace and then a value to associate with that pattern (blue in the example). If the value is too long to fit on one line then it can be placed on multiple lines with each line but the last ending in a backslash-newline sequence:

```
*Gizmo.text: This is a very long initial \
value to use for the text option in all \
"Gizmo" widgets.
```

The backslashes and newlines will not be part of the value.

Blank lines are ignored, as are lines whose first non-blank character is # or !.

16.13.3 Priorities

It is possible for several patterns in the option database to match a particular option. When this happens Tk uses a two-part priority scheme to determine which pattern applies. Tk's mechanism for resolving conflicts is different than the standard mechanism supported by the Tk toolkit, but I think it's simpler and easier to work with.

For the most part the priority of an option in the database is determined by the order in which it was entered into the database: newer options take priority over older ones. When specifying options (e.g. by typing them into your .Xdefaults file) you should specify the more general options first, with more specific overrides following later. For example, if you want button widgets to have a background color of Bisque1 and all other widgets to have white backgrounds, then put the following lines in your .Xdefaults file:

```
*background: white
*Button.background: Bisque1
```

The *background pattern will match any option that the *Button.background pattern matches, but the *Button.background pattern has higher priority since it was specified last. If the order of the patterns had been reversed then all widgets (including buttons) would have white backgrounds and the *Button.background pattern would have no effect.

In some cases it may not be possible to specify general patterns before specific ones (e.g. you might add a more general pattern to the option database after it has already been initialized with a number of specific patterns from the RESOURCE_MANAGER property). To accommodate these situations, each entry also has an integer priority level between 0 and 100, inclusive. An entry with a higher priority level takes precedence over entries with lower priority levels, regardless of the order in which they were inserted into the option database. Priority levels are not used very often in Tk; for complete details on how they work, please refer to the reference documentation.

Tk's priority scheme is different that the scheme used by other X toolkits such as Xt. Xt gives higher priority to the most specific pattern, e.g. `.a.b.foreground` is more specific than `*foreground` so it receives higher priority regardless of the order in which the patterns appear. In most cases this won't be a problem: specify options for Xt applications using the Xt rules, and for Tk applications using the Tk rules. In cases where you want to specify options that apply both to Tk applications and Xt applications, use the Xt rules but also make sure that the patterns considered higher-priority by Xt also appear later in your `.Xdefaults` file. In general, you shouldn't need to specify very many options to Tk applications (if you do, it suggests that the applications haven't been designed well), so the issue of pattern priority shouldn't come up often.

It's important to remember that the option database is only queried for options not specified explicitly in the widget creation command. This means that the user will not be able to override any option that was specified on the command line. If you want to specify a value for an option but allow the user to override that value through the RESOURCE_MANAGER property, you should specify the value for the option using the `option` command described below.

16.13.4 The option command

The `option` command allows you to manipulate the option database while an application is running. The command `option add` will create a new entry in the database. It takes two or three arguments. The first two arguments are the pattern and value for the new entry and the third argument, if specified, is a priority level for the new entry. For example,

```
option add *Button.background Bisque1
```

adds an entry that sets the background color for all button widgets to `Bisque1`.

The command

```
option clear
```

will remove all entries from the option database. The `option readfile` command will read a file in the format described above for the RESOURCE_MANAGER property and make entries in the option database for each line. For example, the following script discards any existing options (including those loaded automatically from the RESOURCE_MANAGER property) and reloads the database from file `newOptions`:

```
option clear
option readfile newOptions
```

The `option readfile` command can also be given a priority level as an extra argument after the file name.

To query whether there is an entry in the option database that applies to a particular option, use the `option get` command:

```
option get .a.b background Background
```

This command takes three arguments, which are the path name of a widget (`.a.b`), the database name for an option (`background`) and the class for that option (`Background`). The command will search the option database to see if any entries match the given window, option, and class. If so, the value of the highest-priority matching option is returned. If no entry matches then an empty string is returned.

Chapter 17
Geometry Managers: The Placer

Geometry managers are the entities that determine the dimensions and locations of widgets. Tk is similar to other X11 toolkits in that it doesn't allow individual widgets to determine their own geometry. A widget will not even appear on the screen unless it is managed by a geometry manager. This separation of geometry management from internal widget behavior allows multiple geometry managers to exist simultaneously and it allows any widget to be used with any geometry manager. If widgets selected their own geometry then this flexibility would be lost: every existing widget would have to be modified to introduce a new style of layout.

This chapter describes the overall structure for geometry management and then presents the placer, which is Tk's simplest geometry manager . The placer manages windows independently without considering other related windows, so it isn't very flexible in the layouts it produces. Because of this, the placer tends to be used only in special situations. Chapter 18 describes a more powerful geometry manager called the packer. The packer lays out groups of windows together, considering the needs of each of the windows when laying out the group. This produces more flexible layouts but also makes the packer harder to understand.

17.1 An overview of geometry management

A geometry manager's job is to arrange one or more *slave* windows relative to a *master* window. For example, it might arrange three slaves in a row from left to right across the area of the master, or it might arrange two slaves so that they split the space of the master with one slave occupying the top half and the other occupying the bottom half. Different

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

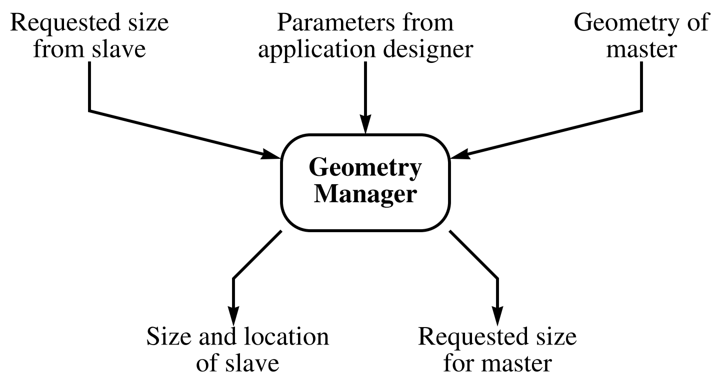


Figure 17.1. A geometry manager receives three kinds of inputs: a requested size for each slave (which usually reflects the information to be displayed in the slave), commands from the application designer (such as "arrange these three windows in a row"), and the actual geometry of the master window. The geometry manager then assigns a size and location to each slave. It may also set the requested size for the master window, which can be used by a higher-level geometry manager to manager the master.

geometry managers embody different styles of layout. The master is often the parent of the slave but there are times when it's convenient to use other windows as masters (you'll see examples of this later).

A geometry manager receives three sorts of information for its use in computing a layout (see Figure 17.1). First, each slave widget requests a particular width and height. These are usually the minimum dimensions needed by the widget to display its information. For example, a button widget requests a size just large enough to display its text or bitmap along with the border specified for the widget. Although geometry managers aren't obliged to satisfy the requests made by their slave widgets, they usually do.

The second kind of input for a geometry manager comes from the application designer and is used to control the layout algorithm. The nature of this information varies from geometry manager to geometry manager. In some cases the information is very specific. For example, with the placer an application designer can specify the precise location and dimensions for a given slave; all the placer does is to apply the given geomety to the slave window. In other cases the information is more abstract. For example, with the packer an application designer can name three slaves and request that they be arranged in a row from left to right within the master; the packer will then check the requested sizes of the slaves and position them so that they abut in a row, with each slave given just as much space as it needs.

The third kind of information used by geometry managers is the geometry of the master window. For example, the geometry manager might position a slave at the lower left corner of its master, or it might divide the space of the master among one or more slaves, or it might refuse to display a slave altogether if it doesn't fit within the area of its master.

Once it has received all of the above information, the geometry manager executes a layout algorithm to determine the dimensions and position of each of its slaves. If the size of a widget isn't what it requested then the widget must make do in the best way it can. Geometry managers usually try to give widgets the space they requested, but they may produce better layouts by giving widgets extra space in some situations. If there isn't enough space in a master for all of its slaves, then some of the slaves may get less space than they asked for. In extreme cases the geometry manager may choose not to display some slaves at all.

The controlling information for geometry management may change while an application runs. For example, a button might be reconfigured with a different font or bitmap, in which case it will change its requested dimensions. Or, the geometry manager might be told to use a different approach (e.g., arrange a collection of windows from top to bottom instead of left to right) or some of the slave windows might be deleted, or the user might interactively resize the master window. When any of these things happens the geometry manager recomputes the layout.

Some geometry managers (e.g. the packer) will set the requested size for the master window. For example, the packer computes how much space is needed in the master to accommodate all of its slaves in the fashion requested by the application designer. It then sets the requested size for the master to these dimensions, overriding any request made by the master widget itself. This approach allows for hierarchical geometry management, where each master is itself the slave of another higher-level master. Size requests pass up through the hierarchy from each slave to its master, resulting ultimately in a size request for a top-level window, which is passed to the window manager. Then actual geometry information passes down through the hierarchy, with the geometry manager at each level accepting the geometry of a master and using it to compute the geometry of one or more slaves. As a result, the entire hierarchy sizes itself to just meet the needs of the lowest-level slaves (the master windows "shrink-wrap" around their slaves).

Each widget can be managed by at most one geometry manager at a time, although it is possible to switch geometry managers during the life of a slave. A widget can act as master to any number of slaves, and it is even possible for different geometry managers to control different groups of slaves associated with the same master. A single geometry manager can simultaneously manage different groups of slaves associated with different masters.

Only internal windows may be slaves for geometry management. The techniques described here do not apply to top-level or main windows. These windows are managed by the window manager for the display; see Chapter 22 for information on how to control their geometry.

`place` *`window option value`* ?*`option value`* ...?	Same as `place configure` command described below.
`place configure` *`window option value`* ?*`option value`* ...?	Arranges for the placer to manage the geometry of *`window`*. The *`option`* and *`value`* arguments determine the dimensions and position of *`window`*.
`place dependents` *`window`*	Returns a list whose elements are the slave windows managed by the placer for which *`window`* is the master.
`place forget` *`window`*	Causes the placer to stop managing *`window`* and unmap it from the screen. Has no effect if *`window`* isn't currently managed by the placer.
`place info` *`window`*	Returns a list giving the current configuration of *`window`*. The list consists of *`option`*-*`value`* pairs in exactly the same form as might be specified to the `place configure` command. Returns an empty string if *`window`* isn't currently managed by the placer.

Table 17.1. A summary of the `place` command.

17.2 Controlling positions with the placer

The placer is a simple geometry manager that implements fixed placements. The application designer specifies the position and size of each slave relative to its master, and the placer simply implements the requested placement. The placer treats each slave independently, so changes in the placement of one slave have no effect on any other slave.

The `place` command is used to communicate with the placer; see Table 17.1 for a summary of its features. In its simplest form its arguments consist of a window name and one or more configuration options specified as name-value pairs:

```
place .x -x 0 -y 0
```

This command positions window `.x` so that its upper-left corner appears at the upper-left corner of its master, which defaults to its parent. The placer supports about a dozen configuration options in all; Table 17.2 summarizes the options and Figure 17.2 shows some examples of using the placer.

The placer determines the position of a slave window in two steps. First, it uses the `-x`, `-y`, `-relx`, and `-rely` options to choose an anchor point, then it positions the slave relative to that anchor point using the `-anchor` option. The anchor point is specified relative to the upper left corner of the master window. If the `-x` and `-y` options are used then the position is given with absolute distances in any of the forms described in Section 16.3. If the `-relx` and `-rely` options are used then the position is specified as a fraction of the size of the master; for example, "`-relx .75`" specifies that the anchor point should lie

`-x` *distance*	Specifies the horizontal distance of the slave's anchor point from the left edge of its master.
`-y` *distance*	Specifies the vertical distance of the slave's anchor point from the top edge of its master.
`-relx` *fraction*	Specifies the horizontal position of the slave's anchor point in a relative fashion as a floating-point number. If *fraction* is `0.0` it refers to the master's left edge, and `1.0` refers to the right edge. *Fraction* need not lie between 0.0 and 1.0.
`-rely` *fraction*	Specifies the vertical position of the slave's anchor point in a relative fashion as a floating-point number. If *fraction* is `0.0` it refers to the master's top edge, and `1.0` refers to the bottom edge. *Fraction* need not lie between 0.0 and 1.0.
`-anchor` *anchor*	Specifies which point on the slave window is to be positioned over the anchor point.
`-width` *distance*	Specifies the width of the slave.
`-height` *distance*	Specifies the height of the slave.
`-relwidth` *fraction*	Specifies the slave's width as a fraction of the width of its master.
`-relheight` *fraction*	Specifies the slave's height as a fraction of the height of its master.
`-in` *window*	Specifies the master window for the slave. Must be the slave's parent or a descendant of the parent.
`-bordermode` *mode*	Specifies how the master's borders are to be used in placing the slave. *Mode* must be `inside`, `outside`, or `ignore`.

Table 17.2. A summary of the configuration options supported by the placer.

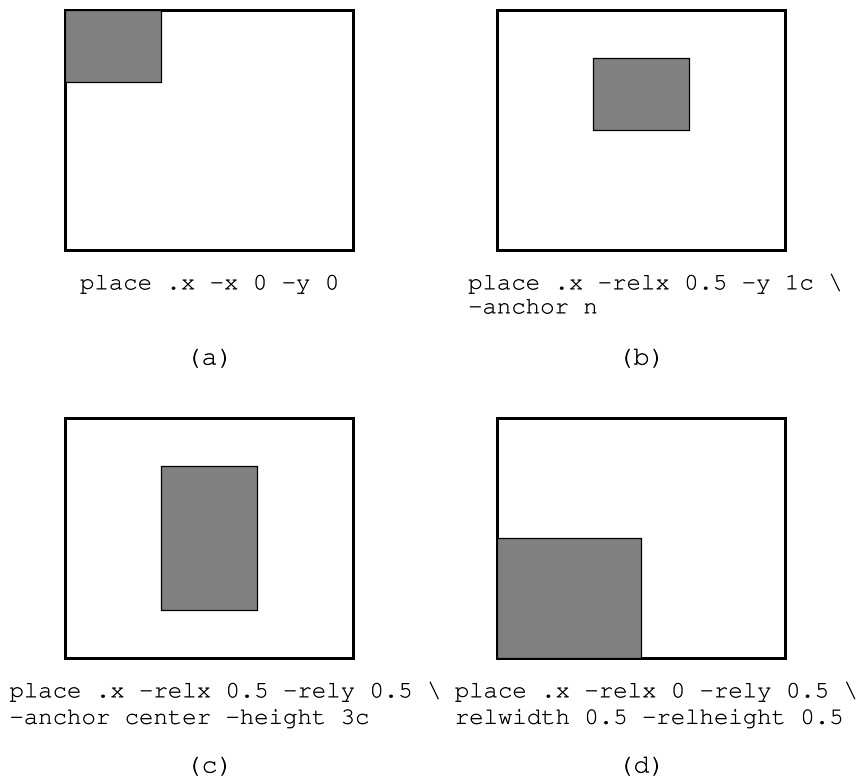


Figure 17.2. Examples of using the placer to manage a window. Each figure shows a `place` command and the layout that results. The larger window is the master and the smaller shaded window is `.x`, the slave being managed. In (a) and (b) the slave is given the size it requested. In (c) the height of the slave is specified in the `place` command, and in (d) both the width and height of the slave are specified in the `place` command.

three-fourths of the way from the left edge of the master to its right edge. These forms can be mixed for a given slave, as in Figure 17.2(b).

The `-anchor` option indicates which point on the slave window should be positioned over the anchor point. It can have any of the anchor names described in Section 16.8. For example, an anchor position of `s` positions the slave so that the center of its bottom edge lies over the anchor point.

It is possible to position a slave outside the area of its master, for example by giving a negative `-x` option or a `-rely` option greater than 1.0. However, X clips each window to the dimensions of its parent, so the portions of the slave that lie outside its parent will not appear on the screen. In the normal case where the parent is the master it probably isn't very useful to position the slave outside its master. However, if the master is a sibling or nephew of the slave then the slave can be positioned outside its master and still be visible on the screen. See Section 17.4 for information on changing the master window.

17.3 Controlling the size of a slave

By default, a slave window managed by the placer is given the size it requests. However, the `-width`, `-height`, `-relwidth`, and `-relheight` options may be used to override either or both of the slave's requested dimensions. The `-width` and `-height` options specify the dimensions in absolute terms, and `-relwidth` and `-relheight` specify the dimensions as a fraction of the size of the master. For example, the following command sets the width of `.x` to 50 pixels and the height to half the height of its master:

```
place .x -width 50 -relheight 0.5
```

17.4 Selecting the master window

In most cases the master window for a given slave will be its parent in the window hierarchy. If no master is specified, the placer uses the parent by default. However, it is sometimes useful to use a different window as the master for a slave. For example, it might be useful to attach one window to a sibling so that whenever the sibling is moved the window will follow. This can be accomplished using the `-in` configuration option. For example, the following command arranges for `.x` always to be displayed with its upper-left corner "glued" to the upper right corner of `.y`:

```
place .x -in .y -relx 1.0 -rely 0
```

In this example, `.x` won't actually be "in" `.y`; `.y` will be `.x`'s master and `.x` will be displayed outside `.y` but adjacent to it.

Note: *The master for a slave must be either the parent of the slave or a descendant of the parent. The reason for this restriction has to do with X's clipping rules. Each window is clipped to the boundaries of its parent; no portion of a child that lies outside of its parent will be displayed. Tk's restriction on master windows gurantees that the slave will be visible and unclipped if its master is visible and unclipped. Suppose that the restriction were not enforced, so that window* `.x.y` *could have* `.a` *as its master. Suppose also that* `.a` *and* `.x` *do not overlap at all. If you asked the placer to position* `.x.y` *at the center of* `.a`*, the placer would set* `.x.y`*'s position as requested, but this would cause* `.x.y` *to be outside the area of* `.x` *so X would not display it, even though* `.a` *is fully visible. This behavior would be confusing to application designers so Tk restricts mastership to keep it from occurring. The restriction applies to all of Tk's geometry managers.*

17.5 Border modes

The last configuration option for the placer is `-bordermode`; it determines how the masters borders are used in placing the slave, and it must have one of the values `inside`, `outside`, or `ignore`. A border mode of `inside` is typically used when placing the slave inside the master, and it is the default. In this case, the placer considers the area of the master to be its innermost area, inside any borders. The anchor point is specified relative to the upper-left corner of this area, and the `-relx`, `-rely`, `-relwidth`, and `-relheight` options use the dimensions of this inner area.

A border mode of `outside` is typically used when positioning the slave outside the area of its master. In this case the placer considers the area of the master to be its outermost area including all borders.

The final border mode, `ignore`, causes the placer to completely ignore any borders and use the master's official X area. This area includes the 3D borders drawn by widgets, which are drawn inside a window's X area, but excludes any external borders. The `ignore` option is provided for completeness but probably isn't very useful.

17.6 More on the place command

So far the `place` command has been discussed in its simplest form, where its first argument is the name of a slave window to manage. `Place` also has several other forms, where the first argument selects a particular command option. `Place configure` has the same effect as the short form that's been used so far. For example, the following two commands have the same effect:

```
place .x -x 0 -y 0
place configure .x -x 0 -y 0
```

`Place configure` (or place without a specific option) can be invoked at any time to change the configuration of a slave window. When invoked on a window already managed by the placer, unspecified options retain their previous values.

The command `place dependents` returns a list of all the slave windows managed by the placer for a given master window:

```
place dependents .
.x .y .z
```

`Place info` returns information about the current configuration of a slave window managed by the placer:

```
place info .x
-x 0 -y 0 -anchor nw
```

The return value is a list containing name-value pairs in exactly the same form that you would specify them to `place configure`. It can be used to record the placement of a window so that it can be restored later.

Lastly, `place forget` causes the placer to stop managing a given slave window:

```
place forget .x
```

As a side effect, it unmaps the window so that it no longer appears on the screen. `Place forget` is useful if you decide that a window should be managed by a different geometry manager: you can tell the placer to forget it, then ask a different geometry manager to take over. You don't need to invoke `place forget` before deleting a widget: the placer (like all geometry managers) automatically forgets about widgets when they are deleted.

17.7 Controlling the size of the master

Although it is possible for a geometry manager to set the requested size for the master windows it manages, the placer does not do this. It simply uses whatever size is provided for a given master, without attempting to influence that size at all. Thus you'll need to use some other mechanism to specify the master's size (e.g. if the master is a frame widget you can request particular dimensions with the `-width` and `-height` configuration options).

Chapter 18
The Packer

The packer is the second geometry manager provided by Tk. Although it is slightly more complicated than the placer described in Chapter 17, it is more powerful because it arranges groups of slaves together, taking into account the needs of one slave when choosing the geometry for the others. With the packer it is easy to achieve effects such as “arrange the following three windows in a row” or “put the menu bar across the top of the window, then the scrollbar across the right side, then fill the remaining space with a text widget.” Because of this, the packer is much more commonly used than the placer, and the placer tends to be used only for special purposes. The `pack` command, summarized in Table 18.1, is used to communicate with the packer.

Note: The `pack` *command syntax described in this chapter is what will eventually exist in a future release of Tk. No existing release supports this syntax. The current Tk release provides essentially all of the features described in this chapter but with a clumsier syntax. The only difference in features has to do with padding. Please refer to the manual entry for the* `pack` *command before writing any scripts that use it.*

18.1 Packer basics

The packer maintains a list of all the slaves for a given master window, called the *packing list*. The packer arranges the slaves by processing the packing list in order, packing one slave in each step. At the time a particular slave is processed, part of the area of the master window has already been allocated to earlier slaves on the list, leaving a rectangular unallocated area left for this and all remaining slaves, as shown in Figure 18.1(a). The slave is positioned in three steps: allocate a frame, stretch the slave, and position it in the frame.

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

`pack` *`window`* ?*`window`* ...? *`option value`* ?*`option value`* ...?	
	Same as `pack configure` command described below.
`pack configure` *`window`* ?*`window`* ...? *`option value`* ?*`option value`* ...?	
	Arrange for the packer to manage the geometry of the *`windows`*. The *`option`* and *`value`* arguments provide information that determines the dimensions and position of the *`windows`*.
`pack forget` *`window`*	
	Causes the packer to stop managing *`window`* and unmap it from the screen. Has no effect if *`window`* isn't currently managed by the packer. Returns an empty string.
`pack info` *`window`*	
	Returns a list giving the current configuration of *`window`*. The list consists of *`option-value`* pairs in exactly the same form as might be specified to the `pack configure` command. Returns an empty string if *`window`* isn't currently managed by the packer.
`pack slaves` *`window`*	
	Returns a list of the slaves on window's packing list, in order.

Table 18.1. A summary of the `pack` command.

In the first step a rectangular region called a *frame* is allocated from the available space. This is done by "slicing" off a piece along one side of the available space. For example, in Figure 18.1(b) the frame has been sliced from the right side of the available space. The packer allows you to control the width of the frame (if it is on the left or right) or the height of the frame (if it is on the top or bottom) and which side to slice it from. By default, the controllable dimension of the frame is taken from the window's requested size in that dimension.

In the second step the packer chooses the dimensions of the slave. By default the slave will get the size it requested, but you can specify instead that it should be stretched in one or both dimensions to fill the space of the frame. If the slave's requested size is larger than the frame then it is reduced to fit the size of the frame. In Figure 18.1(c) the slave has been stretched horizontally but not vertically.

The third step is to position the slave inside its frame. If the slave is smaller than the frame then you can specify an anchor position for the slave such as `n`, `s`, or `center`. In Figure 18.1(c) the slave has been positioned in the center of the frame, which is the default.

Once the slave has been positioned, a smaller rectangular region is left for the next slave to use, as shown in Figure 18.1(d). If a slave doesn't use all of the space in its frame, as in Figure 18.1, the leftover space is unused; it won't be used for later slaves. Thus each step in the packing starts with a rectangular region of available space and ends up with a smaller rectangular region.

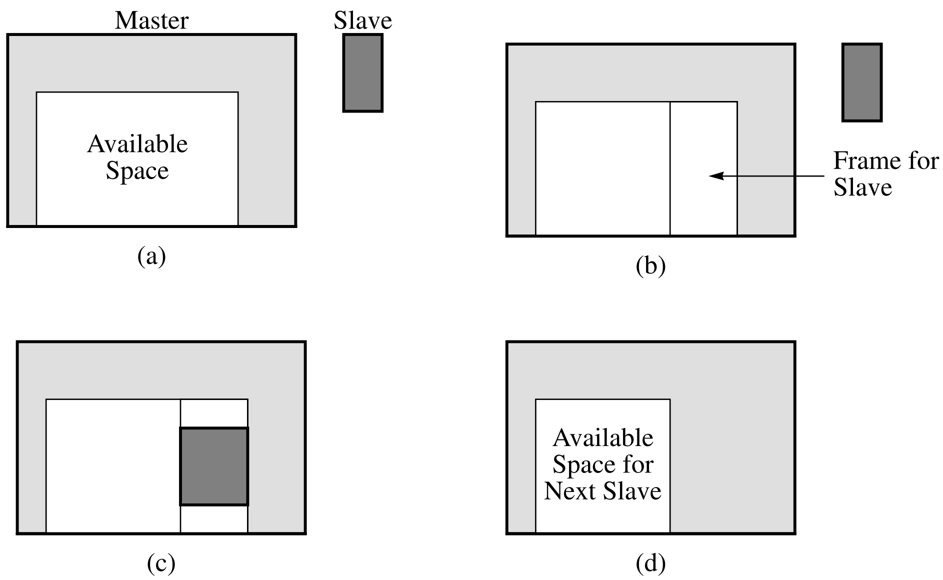


Figure 18.1. The steps taken to pack a single slave. Figure (a) shows the situation before packing a slave. Part of the master's area has already been allocated for previous slaves, and a rectangular region is left for the remaining slaves. The current slave is shown in its requested size. The packer allocates a frame for the slave along one side of the available space, as shown in (b). The packer may stretch the slave to partially or completely fill the frame, then it positions the slave over the frame as in (c). This leaves a smaller rectangular region for the next slave to use, as shown in (d).

The `pack` command is used to communicate with the packer. In its simplest form, a `pack` command takes one or more window names as arguments, followed by one or more pairs of additional arguments that indicate how to manage the windows. For example, consider the following command:

```
pack .ok .cancel .help -side left
```

This command asks the packer to manage `.ok`, `.cancel`, and `.help` as slaves and to pack them in that order. The master for the slaves defaults to their parent. The "`-side left`" option indicates that the frame for each slave should be allocated on the left side of the available space. By default, the frame for each slave is allocated just wide enough for the slave's requested width, and the slave is centered in its frame without any stretching. The result is that the slaves will be arranged in a row from left to right across the master, as shown in Figure 18.2 (b).

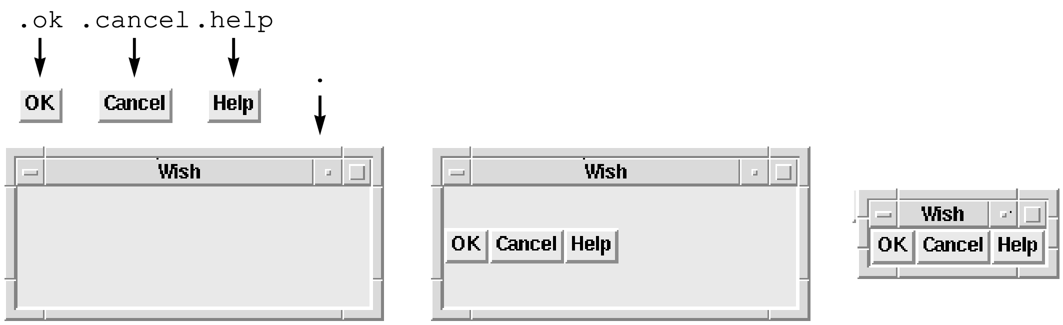


Figure 18.2. A simple example of packing. Figure (a) shows a master window and the requested sizes for three slaves. Figure (b) shows the arrangement that is produced by the command "`pack .ok .cancel .help -side left`" if the master's size is fixed. In most cases, however, the master will resize so that it just meets the needs of its slaves, producing the result in (c).

The result in Figure 18.2(b) assumes that the master window is fixed in size. However, this isn't usually the case. As part of its layout computation the packer computes the minimum dimensions the master would need so that all of its slaves just barely fit, and it sets the requested size of the master to those dimensions. In most cases the geometry manager for the master will set the master's size from those dimensions, so that the master "shrink wraps" around the slaves. For example, top-level windows resize themsleves to their requested dimensions unless other directions have been given with the `wm` command described in Chapter 22. Thus the result from the `pack` command above is more likely to be as shown in Figure 18.2(c). You can choose between the scenarios in Figure 18.2(b) and Figure 18.2(c) with the way you manage the master's geometry.

Figure 18.3 shows another simple packer example, which uses the following script to arrange three windows:

```
pack .label -side top -fill x
pack .scrollbar -side right -fill y
pack .listbox
```

The three windows are configured differently so a separate `pack` command is used for each one. The order of the `pack` commands determines the order of the windows in the packing list. The `.menubar` widget is packed first, and it occupies the top part of the master window. The "`-fill x`" option specifies that the window should be stretched horizontally so that it fills its frame. The scrollbar widget is packed next, in a similar fashion except that it is arranged against the right side of the window and stretched vertically. The widget `.listbox` is packed last. No options need to be specified for `.listbox`: it gets all the remaining space regardless of which side it is packed against.

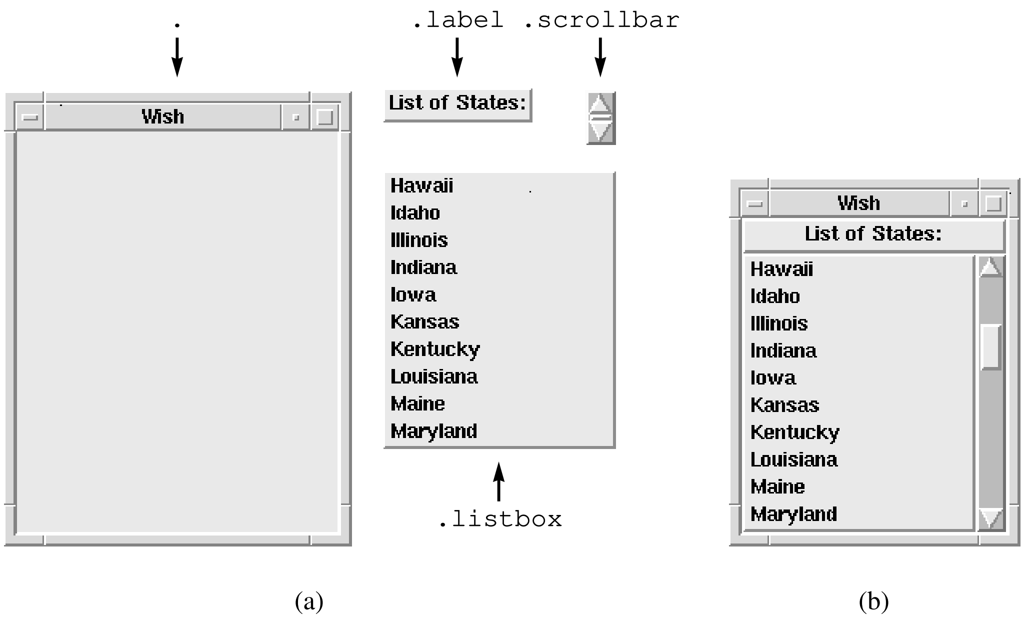


Figure 18.3. Another packer example. Figure (a) shows a master window (`.`) and the requested sizes for three slaves. Figure (b) shows the result of packing the slaves with the script

```
pack .label -side top -fill x
pack .scrollbar -side right -fill y
pack .listbox
```

under the assumption that the master window resizes to just meet the needs of its slaves.

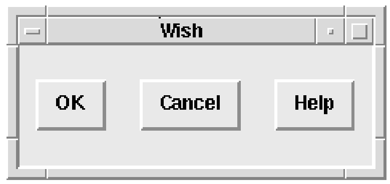
18.2 Packer configuration options

The examples in the previous section illustrated a few of the configuration options provided by the packer; Table 18.2 contains a complete listing. The options fall into three groups: those that determine the location and size of a slave's frame; those that determine the size and position of the slave within its frame; and those that select a master for the slave and determine the slave's position in the master's packing list.

The location of a slave's frame is determined by the `-side` option as already discussed. For slaves packed on the top or bottom, the width of the frame is always the width of the available space left in the master. The height of the frame is usually the requested height of the slave; however, the options `-padx`, `-ipadx`, `-pady`, and `-ipady` cause the packer to pretend that the slave's requested size is larger than what the slave specified. Slaves packed on the left and right sides are handled in an analogous fashion.

-after *window*	Use *window*'s master as the master for the slave and insert the slave into the packing list just after *window*.
-anchor *position*	If the frame is larger than the slave's final size, this option determines where in the frame the slave will be positioned.
-before *window*	Use *window*'s master as the master for the slave and insert the slave into the packing list just before *window*.
-expand *boolean*	If *boolean* is a true value then the slave's frame will be grown to absorb any extra space left over in the master.
-fill *style*	Specifies whether (and how) to grow the slave if its frame is larger than the slave's requested size. *Style* must be either none, x, y, or both.
-in *window*	Use *window* as the master for slave. *Window* must be the slave's parent or a descendant of the slave's parent. If no master is specified then it defaults to the slave's parent.
-ipadx *distance*	*Distance* specifies internal padding for the slave, which is extra horizontal space to allow inside the slave on each side, in addition to what the slave requests.
-ipady *distance*	*Distance* specifies internal padding for the slave, which is extra vertical space to allow inside the slave on each side, in addition to what the slave requests.
-padx *distance*	*Distance* specifies external padding for the slave, which is extra horizontal space to allow outside the slave but inside its frame on each side.
-pady *distance*	*Distance* specifies external padding for the slave, which is extra vertical space to allow outside the slave but inside its frame on each side.
-side *side*	*Side* specifies which side of the master the slave should be packed against. Must be top, bottom, left, or right.

Table 18.2. A summary of the configuration options supported by the packer.




```
pack .ok .cancel .help -side left -ipadx 3m -ipady 2m -expand 1
```

Figure 18.4. An example of the padding and `-expand` options. When the `pack` command in the figure is applied to the windows shown in Figure 18.2(a), the resulting layout is as shown in the figure, assuming that the master's size is fixed. Internal padding causes each window's size to be increased beyond what it requested, and the `-expand` option causes the extra space in the master to be distributed among the slaves' frames.

The `-expand` option allows a frame to absorb leftover space in the master. If the master ends up with more space than its slaves need (e.g. because the user has interactively stretched a top-level window), and if the `-expand` option has been set to true for one of the slaves, then that slave's frame will be expanded to use up all the extra horizontal or vertical space (for left/right and top/bottom slaves, respectively). If multiple slaves have the `-expand` option set, then the extra space is divided evenly among them. See Figure 18.4 for an example that uses `-expand` and the padding options.

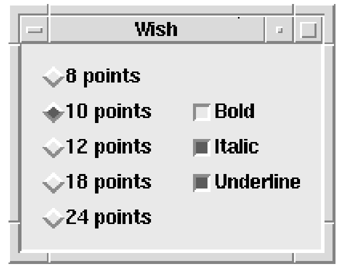
The size and location of a slave within its frame are determined by the `-fill` and `-anchor` options in conjunction with the padding options. The `-fill` option can select no filling, filling in a single direction, or filling in both directions. If internal padding has been specified for a slave (`-ipadx` or `-ipady`) then the slave will be stretched by the amount of the internal padding even if no filling has been requested in that dimension. If external padding has been specified for a slave (`-padx` or `-pady`), then the packer will leave the specified amount of space between the window and the edge of the frame even if filling is requested.

If the final size of the slave is smaller than the frame, then the `-anchor` option controls where to place the slave in the frame. This option may have any of the values described in Section 16.8, such as `nw` to indicate that the northwest (upper-left) corner of the slave should be positioned at the northwest corner of the frame. If external padding has been specified with `-padx` or `-pady`, then `nw` really refers to a point inset from the corner of the frame by the pad amounts.

The third group of options, `-in`, `-before`, and `-after`, controls the master for a slave and the position of the slave in the packing list. By default the master for a slave is its parent and the order of slaves in the packing list is determined by the order of their `pack` commands. However, the `-in` option may be used to specify a different master. As

```
pack .left -side left -padx 3m -pady 3m
pack .right -side right -padx 3m -pady 3m
pack .pts8 .pts10 .pts12 .pts18 .pts24 \
    -in .left -side top -anchor w
pack .bold .italic .underline \
    -in .right -side top -anchor w
```

(a)



(b)

Figure 18.5. Hierarchical packing. The pack commands in (a) produce the layout shown in (b). Two invisible frame widgets, `.left` and `.right`, are used to achieve the column effect.

with the placer, the master must be either the slave's parent or a descendant of the slave's parent (see page 185 for an explanation of this restriction). The `-before` and `-after` options allow you to control the order in which slaves are packed. When one of these options is used, the master for the slave is automatically set to the master for the window named in the option.

18.3 Hierarchical packing

The packer is often used in hierarchical arrangements where slave windows are also masters for other slaves. Figure 18.5 shows an example of hierarchical packing. The resulting layout has a column of radio buttons on the left and a column of check buttons on the right, with each group of buttons centered vertically in its column. To achieve this effect two extra frame widgets, `.left` and `.right`, are packed side by side in the main window, then the buttons are packed inside them. The packer sets the requested sizes for `.left` and `.right` to provide enough space for the buttons, then uses this information to set the requested size for the main window. The main window's geometry will be set to the requested size, then the packer will arrange `.left` and `.right` inside the it, and finally it will arrange the buttons inside `.left` and `.right`.

Figure 18.5 also illustrates why it is sometimes useful for a window's master to be different from its parent. It would have been possible to create the button windows as children of `.left` and `.right` (e.g. `.left.pts8` instead of `.pts8`) but it is better to create them as children of `.` and then pack them inside `.left` and `.right`.The windows `.left` and `.right` serve no purpose in the application except to help in geometry management. They are not even visible on the screen. If the buttons were children of their geometry masters then changes to the geometry management (such as adding more levels in the packing hierarchy) might require the button windows to be renamed and would break any code that used the old names (such as entries in users' .Xdefaults files). It is better to give windows names that reflect their logical purpose in the application, build separate frame hierarchies where needed for geometry management, and then pack the functional windows into the frames.

18.4 Other options to the pack command

So far the pack command has been discussed in its most common form, where the first argument is the name of a slave window and the other arguments specify configuration options. Table 18.1 shows several other forms for the pack command, where the first argument selects a particular command option. Pack configure has the same effect as the short form that's been used up until now: the remaining arguments specify windows and configuration options. If pack configure (or the short form with no command option) is applied to a window that is already managed by the packer, then the slave's configuration is modified; configuration options not specified in the pack command retain their old values.

The command pack slaves returns a list of all of the slaves managed by the packer for a given master window. The order of the slaves in the list reflects their order in the packing list:

```
pack slaves .left
.pts8 .pts10 .pts12 .pts18 .pts24
```

Pack info returns all of the configuration options for a given slave:

```
pack info .pts8
-in .left -side top -anchor w
```

The return value is a list consisting of names and values for configuration options in exactly the form you would specify them to pack configure. This command can be used to save the state of a slave so that it can be restored later.

Lastly, pack forget causes the packer to stop managing one or more slaves and forget all of its configuration state for them. It also unmaps the windows so that they no longer appear on the screen. This command can be used to transfer control of a window from one geometry manager to another, or simply to remove a window from the screen for a while. If a forgotten window is itself a master for other slaves, the information about those slaves is retained but the slaves won't be displayed on the screen until the master window becomes managed again.

Chapter 19
Bindings

You have already seen that Tcl scripts can be associated with certain widgets such as buttons or menus so that the scripts are invoked whenever certain events occur, such as clicking a mouse button over a button widget. These mechanisms are provided as specific features of specific widget classes. Tk also contains a general-purpose *binding* mechanism that can be used to create additional event handlers for widgets. A binding "binds" a Tcl script to an X event or sequence of X events in one or more windows; the script will be invoked automatically by Tk whenever the given event sequence occurs in any of the windows. You can create new bindings to extend the basic functions of a widget (e.g. with keyboard accelerators for common actions), or you can override or modify the default behaviors of widgets, since they are implemented with bindings.

This chapter assumes that you already know at least the basics about X event types, keysyms, modifiers, and the fields in event structures. More information on these topics can be found in any of several books that describe the Xlib programming interface.

19.1 An overview of the bind command

The `bind` command is used to create, modify, query, and remove bindings; Table 19.1 summarizes its syntax. This section illustrates the basic features of `bind`, and later sections go over the features in more detail.

Bindings are created with commands like the one below:

```
bind .entry <Control-d> {.entry delete insert}
```


Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

`bind` *`windowSpec sequence script`*	Arranges for *`script`* to be executed each time the event sequence given by *`sequence`* occurs in the window(s) given by *`windowSpec`*. If a binding already exists for *`windowSpec`* and *`sequence`* then it is replaced. If *`script`* is an empty string then the binding for *`windowSpec`* and *`sequence`* is removed, if there is one.
`bind` *`windowSpec sequence +script`*	If there is already a binding for *`windowSpec`* and *`sequence`* then appends *`script`* to the script for the current binding; otherwise creates a new binding.
`bind` *`windowSpec sequence`*	If there is a binding for *`windowSpec`* and *`sequence`* then returns its script. Otherwise returns an empty string.
`bind` *`windowSpec`*	Returns a list whose entries are all of the sequences for which *`windowSpec`* has bindings.
`tkerror` *`message`*	Invoked by Tk when it encounters a Tcl error in an event handler such as a binding. *`Message`* is the error message returned by Tcl. Any result returned by `tkerror` is ignored.

Table 19.1. A summary of the `bind` and `tkerror` commands.

The first argument to the command specifies the path name of the window that the binding applies to. It can also be a widget class name, in which case the binding applies to all widgets of that class (such bindings are called *class bindings*), or it can be `all`, in which case the binding applies to all widgets. The second argument specifies a sequence of one or more X events. In this example the sequence specifies a single event, which is a key-press of the `d` character while the `Control` key is down. The third argument may be any Tcl script. The script in the example invokes `.entry`'s widget command to delete the character just after the insertion cursor.

After the command completes, the script will be invoked whenever Control-d is typed in `.entry`. The binding can trigger any number of times. It remains in effect until `.entry` is deleted or the binding is explicitly removed by invoking `bind` with an empty script:

```
bind .entry <Control-d> {}
```

Note: A binding for a keystroke will only trigger if the input focus is set to the window for the binding. See Chapter 21 for more information on the input focus.

The `bind` command can also be used to retrieve information about bindings. If `bind` is invoked with an event sequence but no script then it returns the script for the given event sequence:

```
bind .entry <Control-d>
```

```
.entry delete insert
```

If `bind` is invoked with a single argument then it returns a list of all the bound event sequences for that window or class:

```
bind .entry
<Control-Key-d>
bind Button
<ButtonRelease-1> <Button-1> <Any-Leave> <Any-Enter>
```

The first example returned the bound sequences for `.entry`, and the second example returned information about all of the class bindings for button widgets.

19.2 Event patterns

Event sequences are constructed out of basic units called *event patterns*, which Tk matches against the stream of X events received by the application. An event sequence can contain any number of patterns, but in practice most sequences only contain a single pattern.

The simplest form for an event pattern consists of a printing character such as `a` or `@`. This form of pattern matches a key-press event for that character as long as there are no modifier keys pressed. For example,

```
bind .entry a {.entry insert insert a}
```

arranges for the character `a` to be inserted into `.entry` at the point of the insertion cursor whenever it is typed.

The second form for an event pattern is longer but more flexible. It consists of one or more fields between angle brackets, with the following syntax:

```
<modifier-modifier-...-modifier-type-detail>
```

White space may be used instead of dashes to separate the various fields, and most of the fields are optional. The *type* field identifies the particular X event type, such as `KeyPress` or `Enter` (see Table 19.2 for a list of all the available types). For example, the command

```
bind .x <Enter> {puts Hello!}
```

causes "`Hello!`" to be printed on standard output whenever the mouse cursor moves into widget `.x`.

For key and button events, the event type may be followed by a *detail* field that specifies a particular button or key. For buttons, the detail is the number of the button (1-5). For keys, the detail is an X *keysym*. A keysym is a textual name that describes a particular key on the keyboard, such as `BackSpace` or `Escape` or `comma`. The keysym for alphanumeric ASCII characters such as "a" or "A" or "2" is just the character itself. Refer to your X documentation for a complete list of keysyms.

Button, ButtonPress	Expose	Leave
ButtonRelease	FocusIn	Map
Circulate	FocusOut	Property
CirculateRequest	Gravity	Reparent
Colormap	Keymap	ResizeRequest
Configure	Key, KeyPress	Unmap
ConfigureRequest	KeyRelease	Visibility
Destroy	MapRequest	
Enter	Motion	

Table 19.2. Names for event types. Some event types have multiple names, e.g. Key and KeyPress.

If no detail field is provided, as in <KeyPress>, then the pattern matches any event of the given type. If a detail field is provided, as in <KeyPress-Escape>, then the pattern only matches events for the specific key or button. If a detail is specified then you can omit the event type: <Escape> is equivalent to <KeyPress-Escape>.

Note: The pattern <1> *is equivalent to* <Button-1>*, not* <KeyPress-1>.

The event type may be preceded by any number of *modifiers*, each of which must be one of the values in Table 19.3. Most of the modifiers are X modifier names, such as Control or Shift. If one or more of these modifiers are specified then the pattern only matches events that occur when the specified modifiers are present. For example, the pattern <Meta-Control-d> requires that both the Meta and Control keys be held down when d is typed, and <B1-Button-2> requires that button 1 already be down when button 2 is pressed. If no modifiers are specified then none must be present: <KeyPress-a> will not match an event if the Control key is down.

If the Any modifier is specified, it means that the state of unspecified modifiers should be ignored. For example, <Any-a> will match a press of the "a" key even if button 1 is down or the Meta key is pressed. <Any-B1-Motion> will match any mouse motion event as long as button 1 is pressed; other modifiers are ignored.

The last two modifiers, Double and Triple, are used primarily for specifying double and triple mouse clicks. They match a sequence of two or three events, each of which matches the remainder of the pattern. For example, <Double-1> matches a double-click of mouse button 1 with no modifiers down, and <Any-Triple-2> matches any triple click of button 2 regardless of modifiers. For a Double or Triple pattern to match, all of the events must occur close together in time and without substantial mouse motion between them.

`Control`	`Button4, B4`	`Mod1, M2, Alt`
`Shift`	`Button5, B5`	`Mod3, M3`
`Lock`	`Any`	`Mod4, M4`
`Button1, B1`	`Double`	`Mod5, M5`
`Button2, B2`	`Triple`	
`Button3, B3`	`Mod1, M1, Meta, M`	

Table 19.3. Modifier names for event patterns. Multiple names are available for some modifiers; for example, `Mod1`, `M1`, `Meta`, and `M` are all synonyms for the same modifier.

19.3 Sequences of events

An event sequence consists of one or more event patterns optionally separated by white space. For example, the sequence `<Escape>a` contains two patterns. It triggers when the `a` key is pressed immediately after the `Escape` key.

A sequence need not necessarily match consecutive events. For example, the sequence `<Escape>a` will match an event sequence consisting of a key-press on `Escape`, a release of `Escape`, and then a press of `a`; the release of `Escape` will be ignored in determining the match. Tk ignores conflicting events in the input event stream unless they are of type `KeyPress` or `ButtonPress`. Thus if some other key is pressed between the `Escape` and the `a` then the sequence won't match. These same rules apply to double events such as `<Double-1>`.

19.4 Conflict resolution

At most one binding will trigger for any given X event. If several bindings match the event then the most specific binding is chosen and only its script is invoked. For example, suppose there are bindings for `<Button-1>` and `<Double-Button-1>` and button 1 is clicked three times. The first button-press event will match only the `<Button-1>` binding, but the second and third presses will match both bindings. Since `<Double-Button-1>` is more specific than `<Button-1>`, its script is executed on the second and third presses. Similarly, `<Escape>a` is more specific than `<a>`, `<Control-d>` is more specific than `<Any-d>` or `<d>`, and `<d>` is more specific than `<KeyPress>`.

There may also be a conflict among bindings with different window specifications. For example, there might be a binding for a specific window, plus another binding for its class, plus another for `all`. When this occurs, any window-specific binding receives preference over any class binding and any class binding receives preference over any `all` binding. For example, if there is an `<Any-KeyPress>` binding for a window and a `<Return>` binding for its class, pressing the return key will trigger the window-specific binding, not the class binding.

Note: The default behaviors for widgets are established with class bindings created by Tk during initialization. You can modify the behavior of an individual widget by creating window-specific bindings that override the class bindings. However, you have to be careful in doing this that you don't accidentally override more behavior than you intended. For example, if you specify an `<Any-KeyPress>` *binding for a widget, it will override a* `<Return>` *binding for the class, even though the* `<Return>` *binding appears to be more specific. The solution is to duplicate the* `<Return>` *class binding for the widget.*

19.5 Substitutions in scripts

If the script for a binding contains `%` characters then it is not executed directly. Instead, a new script is generated by replacing each `%` character and the one that follows it with information about the X event. The character following the `%` selects a specific substitution to make. About 30 different substitutions are defined; see the reference documentation for complete details. The following substitutions are the most commonly used ones:

`%x`	Substitute the x-coordinate from the event.
`%y`	Substitute the y-coordinate from the event.
`%W`	Substitute the path name of the event window.
`%A`	Substitute the 8-bit ISO character value that corresponds to a `KeyPress` or `KeyRelease` event, or an empty string if the event is for a key like Shift that doesn't have an ISO equivalent.
`%%`	Substitute the character `%`.

For example, the following bindings implement a simple mouse tracker:

```
bind all <Enter> {puts "Entering %W"}
bind all <Leave> {puts "Leaving %W"}
bind all <Motion> {puts "Mouse at (%x,%y)"}
```

Note: When Tk makes % substitutions it treats the script as an ordinary string without any special properties. The normal quoting rules for Tcl commands are not considered, so `%` *sequences will be substituted even if embedded in braces or preceded by backslashes. The only way to prevent a* `%` *substitution is to double the* `%` *character. The easiest way to avoid problems with complex scripts and % substitutions is to keep the binding simple, for example by putting the script in a procedure and having the binding invoke the procedure with arguments created via % substitution.*

19.6 When are events processed?

Tk only processes events at a few well-defined times. After a Tk application completes its initialization it enters an *event loop* to wait for X events and other events such as timer and file events. When an event occurs the event loop executes C or Tcl code to respond to that event. Once the response has completed, control returns to the event loop to wait for the next interesting event. Almost all events are processed from the top-level event loop. New events will not be considered while responding to the current event, so there is no danger of one binding triggering in the middle of the script for another binding. This approach applies to all event handlers, including those for bindings, those for the script options associated with widgets, and others yet to be discussed, such as window manager protocol handlers.

A few special commands such as `tkwait` and `update` reinvoke the event loop recursively, so bindings may trigger during the execution of these commands. You should only invoke these commands at times when it is safe for bindings to trigger. Commands that invoke the event loop are specially noted in their reference documentation; all other commands complete immediately without re-entering the event loop.

Note: *Event handlers are always invoked at global level (as if the command "*`uplevel #0`*" were used), even if the event loop was invoked from a* `tkwait` *or* `update` *command inside a procedure. This means that global variables are always accessible in event handlers without invoking the* `global` *command.*

19.7 Background errors: tkerror

It is possible for a Tcl error to occur while executing the script for a binding. These errors are called *background errors*; when one occurs, the default action is for Tk to print the associated error message on standard output. However, this probably isn't very useful in most cases. It is usually better to display the error message in a message window or dialog box on the screen where the user can see it. The `tkerror` command permits each application to handle background errors in the best way for that application. When a background error occurs, Tk invokes `tkerror` with a single argument consisting of the error message. The `tkerror` command is not defined by Tk; presumably each application will define its own `tkerror` procedure to report errors in a way that makes sense for that application. If `tkerror` returns normally then Tk will assume it has dealt with the error and it won't do anything else itself. If `tkerror` returns an error (e.g. because there is no `tkerror` command defined) then Tk falls back on the default approach of printing the message on standard output.

The `tkerror` procedure is invoked not just for errors in bindings, but for all other errors that are returned to Tk at times when it has no-one else to return the errors to. For example, menus and buttons call `tkerror` if an error is returned by the script for a menu entry or button; scrollbars call `tkerror` if a Tcl error occurs while communicating with the associated widget; and the window-manager interface calls `tkerror` if an error is returned by the script associated with a window manager protocol.

19.8 Other uses of bindings

The binding mechanism described in this chapter applies to widgets. However, similar mechanisms are available internally within some widgets. For example, canvas widgets allow bindings to be associated with graphical items such as rectangles or polygons, and text widgets allow bindings to be associated with ranges of characters. These bindings are created using the same syntax for event sequences and `%`-substitutions, but they are created with the widget command for the widget and refer to the widget's internal objects instead of windows. For example, the following command arranges for a message to be printed whenever mouse button 1 is clicked over item 2 in a canvas `.c`:

```
.c bind 2 <ButtonPress-1> {puts Hello!}
```

Chapter 20
The Selection

The *selection* is a mechanism for passing information between widgets and applications. The user first selects one or more objects in a widget, for example by dragging the mouse across a range of text or clicking on a graphical object. Once a selection has been made, the user can invoke commands in other widgets that cause them to retrieve information about the selection, such as the characters in the selected range or the name of the file containing the selection. The widget containing the selection and the widget requesting it can be in the same or different applications. The selection is most commonly used to copy information from one place to another, but it can be used for other purposes as well, such as setting a breakpoint at a selected line or opening a new window on a selected file.

X defines a standard mechanism for supplying and retrieving the selection and Tk provides access to this mechanism with the `selection` command. Table 20.1 summarizes the `selection` command. The rest of this chapter describes its features in more detail. For complete information on the X selection protocol, refer to the Inter-Client Communications Convention Manual (ICCCM).

20.1 Selections, retrievals, and targets

X's selection mechanism allows for multiple selections to exist at once, with names like "primary selection", "secondary selection", and so on. However, Tk supports only the primary selection; Tk applications cannot retrieve or supply selections other than the primary one and the term "selection" always refers to the primary selection in this book. At most one widget has a primary selection at any given time on a given display. When a user selects information in one widget, any selected information in any other widget is auto-

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

`selection clear` *`window`*	If there is a selection anywhere on *`window`*'s display, deselect it so that no window owns the selection anymore.
`selection get ?`*`target`*`?`	Retrieve the value of the primary selection using *`target`* as the form in which to retrieve it, and return the selection's value as result. *`Target`* defaults to `STRING`.
`selection handle` *`window script`* `?`*`target`*`? ?`*`format`*`?`	Creates a handler for selection requests such that *`script`* will be executed whenever the primary selection is owned by *`window`* and someone attempts to retrieve it in the form given by *`target`*. *`Target`* defaults to `STRING`. *`Format`* specifies a representation for transmitting the selection to the requester; it defaults to `STRING`. When *`script`* is invoked, two additional numbers are appended to it, consisting of the starting offset and maximum number of bytes to retrieve. *`Script`* should return the requested range of the selection; if it returns an error then the selection retrieval will be rejected.
`selection own ?`*`window`*`? ?`*`script`*`?`	Claims ownership of the selection for *`window`*; if some other window previously owned the selection, deselects the old selection. If *`script`* is specified then it will be executed when *`window`* is deselected. If neither *`window`* nor *`script`* is specified, then the command returns the path name of the window that currently owns the selection, or an empty string if no window in this application owns the selection.

Table 20.1. A summary of the `selection` command.

matically deselected. It is possible for multiple disjoint objects to be selected simultaneously within a widget (e.g. three different items in a listbox or several different polygons in a drawing window), but usually the selection consists of a single object or a range of adjacent objects.

When you retrieve information about the selection, you can ask for any of several different kinds of information. The different kinds of information are referred to as retrieval *targets*. The most common target is `STRING`. In this case the contents of the selection are returned as a string. For example, if text is selected then a retrieval with target `STRING` will return the contents of the selected text; if graphics are selected then a retrieval with target `STRING` will return some string representation for the selected graphics. If the selection is retrieved with target `FILE_NAME` then the return value will be the name of the file associated with the selection. If target `LINE` is used then the return value will be the number of the selected line within its file. There are many targets with well-defined meanings; refer to the X ICCCM for more information.

The command `selection get` retrieves the selection. The target may be specified explicitly or it may be left unspecified, in which case it defaults to `STRING`. For example, the following commands might be invoked when the selection consists of a few words on one line of a file containing the text of Shakespeare's *Romeo and Juliet*:

```
selection get
star-crossed lovers
selection get FILE_NAME
romeoJuliet
selection get LINE
6
```

These commands could be issued in any Tk application on the display containing the selection; they need not be issued in the application containing the selection.

Not every widget supports every possible selection target. For example, if the information in a widget isn't associated with a file then the `FILE_NAME` target will not be supported. If you try to retrieve the selection with an unsupported target then an error will be returned. Fortunately, every widget is supposed to support retrievals with target `TARGETS`; such retrievals return a list of all the target forms supported by the current selection owner. You can use the result of a `TARGETS` retrieval to pick the most convenient available target. For example, the following procedure retrieves the selection as Postscript as possible, otherwise as an unformatted string:

```
proc getSelection {} {
        set targets [selection get TARGETS]
        if {[lsearch $targets POSTSCRIPT] >= 0} {
            return [selection get POSTSCRIPT]
        }
        selection get STRING
}
```

20.2 Locating and clearing the selection

Tk provides two mechanisms for retrieving information about who owns the selection. The command `selection own` (with no additional arguments) will check to see if the selection is owned by a widget in the invoking application. If so it will return the path name of that widget; if there is no selection or it is owned by some other application then `selection own` will return an empty string.

The second way to locate the selection is with the retrieval targets `APPLICATION` and `WINDOW_NAME`. These targets are both implemented by Tk and are automatically available whenever the selection is in a Tk application. The command

```
selection get APPLICATION
```

returns the name of the Tk application that owns the selection (in a form suitable for use with the `send` command, for example) and

```
selection get WINDOW_NAME
```

returns the path name of the window that owns the selection. These commands will work only if the owning application is based on Tk. If the application that owns the selection isn't based on Tk then it probably does not support the APPLICATION and WINDOW_NAME targets and the selection get command will return an error. These commands will also return errors if there is no selection.

The command

```
selection clear
```

will clear out any selection on the display of the invoking application. It works regardless of whether the selection is in the invoking application or some other application on the same display. The following script will clear out the selection only if it is in the invoking application:

```
if {[selection own] != ""} {
        selection clear
}
```

20.3 Supplying the selection with Tcl scripts

The sections above described Tk's facilities for retrieving the selection; this section describes how to supply the selection. The standard widgets like entries and texts already contain C code that supplies the selection, so you don't usually have to worry about it when writing Tcl scripts. However, it is possible to write Tcl scripts that implement new targets or that provide the complete supply-side protocol, and this section describes how to do it. This feature of Tk is seldom used so you may wish to skip over this material until you need it.

The protocol for supplying the selection has three parts:

1. A widget must claim ownership of the selection. This deselects any previous selection and typically redisplays the selected material in a highlighted fashion.
2. The selection owner must respond to retrieval requests by other widgets and applications.
3. The owner may request that it be notified when it is deselected. Widgets typically respond to deselection by eliminating the highlights on the display.

The paragraphs below describe two scenarios. The first scenario just adds a new target to a widget that already has selection support, so it only deals with the second part of the protocol. The second scenario implements complete selection support for a group of widgets that didn't previously have any; it deals with all three parts of the protocol.

Suppose that you wish to add a new target to those supported for a particular widget. For example, text widgets contain built-in support for the STRING target but they don't automatically support the FILE_NAME target. You could add support for FILE_NAME retrievals with the following script:

```
selection handle .t getFile FILE_NAME
proc getFile {offset maxBytes} {
        global fileName
        set last [expr $offset+$maxBytes-1]
        string range $fileName $offset $last
}
```

This code assumes that the text widget is named `.t` and that the name of its associated file is stored in a global variable named `fileName`. The `selection handle` command tells Tk to invoke `getFile` whenever `.t` owns the selection and someone attempts to retrieve it with target `FILE_NAME`. When such a retrieval occurs, Tk takes the specified command (`getFile` in this case) appends two additional numerical arguments, and invokes the resulting string as a Tcl command. In this example a command like

```
getFile 0 4000
```

will result. The additional arguments identify a sub-range of the selection by its first byte and maximum length, and the command must return this portion of the selection. If the requested range extends beyond the end of the selection, then the command should return everything from the given starting point up to the end of the selection. Tk takes care of returning the information to the application that requested it. In most cases the entire selection will be retrieved in one invocation of the command, but for very large selections Tk will make several separate invocations so that it can transmit the selection back to the requester in manageable pieces.

The above example simply added a new target to a widget that already provided some built-in selection support. If selection support is being added to a widget that has no built-in support at all, then additional Tcl code is needed to claim ownership of the selection and to respond to deselections. For example, suppose that there is a group of three radio buttons named `.a`, `.b`, and `.c` and that the buttons have already been configured with their `-variable` and `-value` options to store information about the selected button in a global variable named `state`. Now suppose that you want to tie the radio buttons to the selection, so that (a) whenever a button becomes selected it claims the X selection, (b) selection retrievals return the contents of `state`, and (c) when some other widget claims the selection away from the buttons then `state` is cleared and all the buttons become deselected. The following code implements these features:

```
selection handle .a getValue STRING
proc getValue {offset maxBytes} {
        global state
        set last [expr $offset+$maxBytes-1]
        string range $state $offset $last
}
foreach w {.a .b .c} {
        $w config -command {selection own .a selGone}
}
proc selGone {} {
```

```
            global state
            set state {}
    }
```

The `selection handle` command and the `getValue` procedure are similar to the previous example: they respond to `STRING` selection requests for `.a` by returning the contents of the `state` variable. The `foreach` loop specifies a `-command` option for each of the widgets. This causes the `selection own` command to be invoked whenever the user clicks on any of the radio buttons, and the `selection own` command claims ownership of the selection for widget `.a` (`.a` will own the selection regardless of which radio button gets selected and it will return `state` in response to selection requests). The `selection own` command also specifies that procedure `selGone` should be invoked whenever the selection is claimed away by some other widget. `SelGone` sets `state` to an empty string. All of the radio buttons monitor `state` for changes, so when it gets cleared the radio buttons will all deselect themselves.

Chapter 21
The Input Focus

At any given time one window of an application is designated as the *input focus window*, or *focus window* for short. All keystrokes received by the application are directed to the focus window and they are processed according to its event bindings. This chapter describes Tk's `focus` command, which is used to control the input focus. Table 21.1 summarizes the syntax of the `focus` command. The focus window only determines what happens once a keystroke event arrives at a particular application; it does not determine which of the applications on the display receives keystrokes. The selection of a focus application is made by the window manager.

21.1 Focus model: explicit vs. implicit

There are two possible ways of handling the input focus, which are known as the *implicit* and *explicit* models. In the implicit model the focus follows the mouse: keystrokes are directed to the window under the mouse pointer and the focus window changes implicitly when the mouse moves from one window to another. In the explicit model the focus window is set explicitly and doesn't change until it is explicitly reset; mouse motions do not change the focus.

Tk implements the explicit focus model, for several reasons. First, the explicit model allows you to move the mouse cursor out of the way when you're typing in a window; with the implicit model you'd have to keep the mouse in the window you're typing to. Second, and more important, the explicit model allows an application to change the focus window without the user moving the mouse. For example, when an application pops up a dialog box that requires type-in (e.g. one that prompts for a file name) it can set the input

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

`focus`	Returns the path name of the application's focus window, or an empty string if there is no focus window.
`focus` *`window`*	Sets the application's focus window to *`window`*.
`focus default ?`*`window`*`?`	If *`window`* is specified then it becomes the default focus window, which will receive the input focus whenever the focus window is deleted. In this case the command returns an empty string. If *`window`* is specified as `none`, then there will be no default focus window. If *`window`* is omitted then the command returns the current default focus window, or `none` if there is no default.
`focus none`	Clears the focus window.

Table 21.1. A summary of the `focus` command.

focus to the appropriate window in the dialog without you having to move the mouse, and it can move the focus back to its original window when you're finished with the dialog box. This allows you to keep your hands on the keyboard. Similarly, when you're typing in a form the application can move the input focus to the next entry in the form each time you type a tab, so that you can keep your hands on the keyboard and work more efficiently. Lastly, if you want an implicit focus model then you can always achieve it with event bindings that change the focus each time the mouse cursor enters a new window.

Tk applications don't need to worry about the input focus very often because the default bindings for text-oriented widgets already take care of the most common situations. For example, when you click button 1 over an entry or text widget, the widget will automatically make itself the focus window. As application designer, you only need to set the focus in cases like those in the previous paragraph where you want to move the focus among the windows of your application to reflect the flow of work.

21.2 Setting the input focus

To set the input focus, invoke the `focus` command with a widget name as argument:

```
focus .dialog.entry
```

From this point on, all keystrokes received by the application will be directed to `.dialog.entry` and the previous focus window will no longer receive keystrokes. The new focus window will display some sort of highlight, such as a blinking insertion cursor, to indicate that it has the focus and the previous focus window will stop displaying its highlight.

Here is a script that implements tabbing among four entries in a form:

```
set tabList {.form.e1 .form.e2 .form.e3 .form.e4}
foreach w $tabList {
        bind $w <Tab> {tab $tabList}
}
proc tab list {
        set i [lsearch $list [focus]]
        incr i
        if {$i >= [llength $list]} {
            set i 0
        }
        focus [lindex $list $i]
}
```

This script assumes that the four entry windows have already been created. It uses the variable `tabList` to describe the order of traversal among the entries and arranges for the procedure `tab` to be invoked whenever a tab is typed in any of the entries. `Tab` invokes `focus` with no arguments to determine which window has the focus, finds where this window is in the list that gives the order of tabbing, and then sets the input focus to the next window in the list. The procedure `tab` could be used for many different forms just by passing it a different `list` argument for each form. The order of focussing can also be changed at any time by changing the value of the `tabList` variable.

21.3 Clearing the focus

The command `focus none` clears the input focus for the application. Once this command has been executed, keystrokes for the application will be discarded.

21.4 The default focus

When the focus window is deleted, Tk automatically sets the input focus for the application to a window called the *default focus window*. The default focus window is initially `none`, which means that there will be no focus window after the focus window is deleted and keystrokes will be discarded until the focus window is set again.

The `focus default` command can be used to specify a default focus window and to query the current default:

```
focus default
none
focus default .entry
focus default
```

```
.entry
```

Once this script has been completed, `.entry` will receive the input focus whenever the input focus window is deleted.

21.5 Keyboard accelerators

Applications with keyboard accelerators (e.g. they allow you to type `Control+s` to save the file or `Control+q` to quit the application) require special attention to bindings and the input focus. First, the accelerator bindings must be present in every window where you want them to apply. For example, suppose that an editor has a main text window plus several entry windows for searching and replacement. You will create bindings for accelerators like `Control+q` in the main text window, but you will probably want most or all of the bindings to apply in the auxiliary windows also, so you'll have to define the accelerator bindings in each of these windows too.

In addition, an application with keyboard accelerators should never let the focus become `none`, since that will prevent any of the accelerators from being processed. If no other focus window is available, I suggest setting the focus to the main window of the application; of course, you'll have to define accelerator bindings for `.` so that they are available in this mode. In addition, I recommend setting the default focus window to `.` or some other suitable window so that the focus isn't lost when dialog boxes and other windows are deleted.

Chapter 22
Window Managers

For each display running the X Window System there is a special process called the *window manager*. The window manager is separate from the X display server and from the application processes using the display. The main function of the window manager is to control the arrangement of all the top-level windows on each screen. In this respect it is similar to the geometry managers described in Chapters 17 and 18 except that instead of managing the internal windows within an application it manages the top-level windows of all applications. The window manager allows each application to request particular locations and sizes for its top-level windows, which can be overridden interactively by users. Window managers also serve several other purposes besides geometry management: they add decorative frames around top-level windows; they allow windows to be iconified and deiconified; and they notify applications of certain events, such as user requests to destroy the window.

X allows for the existence of many different window managers that implement different styles of layout, provide different kinds of decoration and icon management, and so on. Only a single window manager runs for a display at any given time, and the user gets to choose which one. In order to allow any application to work smoothly with any window manager, X defines a protocol for the interactions between applications and window managers. The protocol is defined as part of the Inter-Client Communication Conventions Manual (ICCCM). With Tk you use the `wm` command to communicate with the window manager; Tk implements the `wm` command using the ICCCM protocols so that any Tk-based application should work with any window manger. Tables 22.1 and 22.2 summarize the `wm` command.

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

`wm aspect` *window* ?*xThin yThin xFat yFat*?	Set or query *window*'s aspect ratio. If an aspect ratio is specified, it constrains interactive resizes so that *window*'s width/height will be at least as great as *xThin*/*yThin* and no greater than *xFat*/*yFat*.
`wm client` *window* ?*name*?	Set or query the `WM_CLIENT_MACHINE` property for *window*, which gives the name of the machine on which *window*'s application is running.
`wm command` *window* ?*value*?	Set or query the `WM_COMMAND` property for *window*, which should contain the command line used to initiate *window*'s application.
`wm deiconify` *window*	Arrange for window to be displayed in normal fashion.
`wm focusmodel` *window* ?*model*?	Set or query the focus model for *window* . *Model* must be `active` or *passive*.
`wm geometry` *window* ?*value*?	Set or query the requested geometry for window. *Value* must have the form =*width*x*height x y* (any of =, *width*x*height*, or *x y* can be omitted).
`wm group` *window* ?*leader*?	Set or query the window group that *window* belongs to. *Leader* must be the name of a top-level window, or an empty string to remove *window* from its current group.
`wm iconbitmap` *window* ?*bitmap*?	Set or query the bitmap for *window*'s icon.
`wm iconify` *window*	Arrange for *window* to be displayed in iconic form.
`wm iconmask` *window* ?*bitmap*?	Set or query the mask bitmap for *window*'s icon.
`wm iconname window ?string?`	Set or query the string to be displayed in *window*'s icon.
`wm iconposition` *window* ?*x y*?	Set or query the hints about where on the screen to display *window*'s icon.
`wm iconwindow` *window* ?*icon*?	Set or query the window to use as icon for *window*. *Icon* must be the path name of a top-level window.
`wm maxsize` *window* ?*width height*?	Set or query the maximum permissible dimensions for *window* during interactive resize operations.
`wm minsize` *window* ?*width height*?	Set or query the minimum permissible dimensions for *window* during interactive resize operations.

Table 22.1. A summary of the `wm` command. In all of these commands *window* must be the name of a top-level window. Many of the commands, such as `wm aspect` or `wm group`, are used to set and query various parameters related to window management. For these commands, if the parameters are specified as null strings then the parameters are removed completely, and if the parameters are omitted then the command returns the current settings for the parameters.

`wm overrideredirect` *`window`* ?*`boolean`*?	Set or query the override-redirect flag for *`window`*.
`wm positionfrom` *`window`* ?*`whom`*?	Set or query the source of the position specification for *`window`*. *`Whom`* must be `program` or `user`.
`wm protocol` *`window`* ?*`protocol`*? ?*`script`*?	Arrange for *`script`* to be executed whenever the window manager sends a message to *`window`* with the given *`protocol`*. *`Protocol`* must be the name of an atom for a window manager protocol, such as `WM_DELETE_WINDOW`, `WM_SAVE_YOURSELF`, or `WM_TAKE_FOCUS`. If *`script`* is an empty string then the current handler for protocol is deleted. If *`script`* is omitted then the current script for *`protocol`* is returned (or an empty string if there is no handler for *`protocol`*). If both *`protocol`* and *`script`* are omitted then the command returns a list of all protocols with handlers defined for *`window`*.
`wm sizefrom` *`window`* ?*`whom`*?	Set or query the source of the size specification for *`window`*. *`Whom`* must be `program` or `user`.
`wm state` *`window`*	Returns the current state of *`window`*: `normal`, `iconic`, or `withdrawn`.
`wm title` *`window`* ?*`string`*?	Set or query the title string to display in the decorative border for *`window`*.
`wm transient` *`window`* ?*`master`*?	Set or query the transient status of *`window`*. *`Master`* must be the name of a top-level window on whose behalf *`window`* is working as a transient.
`wm withdraw` *`window`*	Arrange for *`window`* not to appear on the screen at all, either in normal or iconic form.

22.1 Window sizes

If a Tk application doesn't use the `wm` command, Tk will communicate with the window manager automatically on the application's behalf so that its top-level windows appear on the screen. By default each top-level window will appear in its "natural" size, which is the size it requested using the normal Tk mechanisms for geometry management. Tk will forward the requested size on to the window manager and most window managers will honor the request. If the requested size of a top-level window should change then Tk will forward the new size on to the window manager and the window manager will resize the window to correspond to the latest request. By default the user will not be able to resize windows interactively: window sizes will be determined solely by their requested sizes as computed internally.

If you want to allow interactive resizing then you must invoke at least one of the `wm minsize` and `wm maxsize` commands, which specify a range of acceptable sizes. For example the commands

```
wm minsize .x 100 50
wm maxsize .x 400 150
```

will allow `.x` to be resized but constrain it to be 100 to 400 pixels wide and 50 to 150 pixels high. If the command

```
wm minsize .x 1 1
```

is invoked then there will effectively be no lower limit on the size of `.x`. If you set a minimum size without a maximum size (or vice versa) then the other limit will be unconstrained. You can disable interactive resizing again by clearing all of the size bounds:

```
wm minsize .x {} {}
wm maxsize .x {} {}
```

In addition to constraining the dimensions of a window you can also constrain its aspect ratio (width divided by height) using the `wm aspect` command. For example,

```
wm aspect .x 1 3 4 1
```

will tell the window manager not to let the user resize the window to an aspect ratio less than 1/3 (window three times as tall as it is wide) or greater than 4 (four times as wide as it is tall).

If the user interactively resizes a top-level window then the window's internally requested size will be ignored from that point on. Regardless of how the internal needs of the window change, its size will remain as set by the user. A similar effect occurs if you invoke the `wm geometry` command, as in the following example:

```
wm geometry .x 300x200
```

This command forces `.x` to be 300 pixels wide and 200 pixels high just as if the user had resized the window interactively. The internally requested size for `.x` will be ignored once the command has completed, and the size specified in the `wm geometry` command overrides any size that the user might have specified interactively (but the user can resize the window again to override the size in the `wm geometry` command). The only difference between the `wm geometry` command and an interactive resize is that `wm geometry` is not subject to the constraints specified by `wm minsize`, `wm maxsize`, and `wm aspect`.

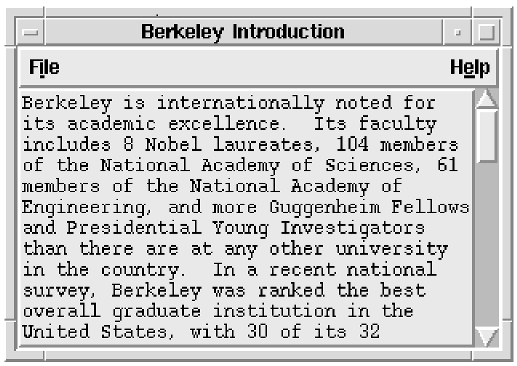
If you would like to restore a window to its natural size you can invoke `wm geometry` with an empty geometry string:

```
wm geometry .x {}
```

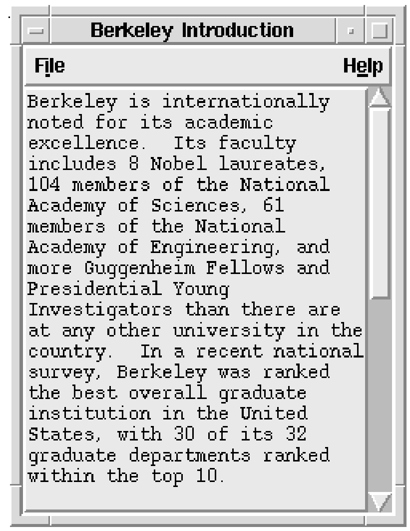
This causes Tk to forget any size specified by the user or by `wm geometry`, so the window will return to the size it requested internally.

22.2 Gridded windows

In some cases it doesn't make sense to resize a window to arbitrary pixel sizes. For example, consider the application in Figure 22.1. When the user resizes the top-level window



(a)



(b)

Figure 22.1. An example of gridded geometry management. If the user interactively resizes the window from the dimensions in (a) to those in (b), the window manager will round off the dimensions so that the text widget holds an even number of characters in each dimension. This figure shows decorative borders as provided by the `mwm` window manager.

the text widget changes size in response. Ideally the text widget should always contain an even number of characters in each dimension, and sizes that result in partial characters should be rounded off.

Gridded geometry management accomplishes this effect. When gridding is enabled for a top-level window its dimensions will be constrained to lie on an imaginary grid. The geometry of the grid is determined by one of the widgets contained in the top-level window (e.g. the text widget in Figure 22.1) so that the widget always holds an integral number of its internal objects. Usually the widget that controls the gridding is a text-oriented widget such as an entry or listbox or text.

To enable gridding, set the `-setgrid` option to 1 in the controlling widget. The following code was used in the example in Figure 22.1, where the text widget is `.t`:

```
.t configure -setgrid 1
```

This command has several effects. First, it automatically makes the main window resizable, even if no `wm minsize` or `wm maxsize` command has been invoked. Second, it constrains the size of the main window so that `.t` will always hold an even number of characters in its font. Third, it changes the meaning of dimensions used in Tk. These dimensions now represent grid units rather than pixels. For example, the command

```
wm geometry . 50x30
```

will set the size of the main window so that `.t` is 50 characters wide and 30 lines high, and dimensions in the `wm minsize` and `wm maxsize` commands will also be grid units. Many window managers display the dimensions of a window on the screen while it is being resized; these dimensions will given in grid units too.

Note: In order for gridding to work correctly you must have configured the internal geometry management of the application so that the controlling window stretches and shrinks in response to changes in the size of the top-level window , e.g. by packing it with the `-expand` *option set to* `1` *and* `-fill` *to* `both`.

22.3 Window positions

Controlling the position of a top-level window is simpler than controlling its size. Users can always move windows interactively, and an application can also move its own windows using the `wm geometry` command. For example, the command

```
wm geometry .x +100+200
```

will position `.x` so that its upper-left corner is at pixel (100,200) on the display. If either of the + characters is replaced with a − then the coordinates are measured from the right and bottom sides of the display. For example,

```
wm geometry .x -0-0
```

positions `.x` at the lower-right corner of the display.

22.4 Window states

At any given time each top-level window is in one of three states. In the *normal* or *de-iconified* state the window appears on the screen. In the *iconified* state the window does not appear on the screen, but a small icon is displayed instead. In the *withdrawn* state neither the window nor its icon appears on the screen and the window is ignored completely by the window manager.

New top-level windows start off in the normal state. You can use the facilities of your window manager to iconify a window interactively, or you can invoke the `wm iconify` command within the window's application, for example

```
wm iconify .x
```

If you invoke `wm iconify` immediately, before the window first appears on the screen, then it will start off in the iconic state. The command `wm deiconify` causes a window to revert to normal state again.

The command `wm withdraw` places a window in the withdrawn state. If invoked immediately, before a window has appeared on the screen, then the window will start off withdrawn. The most common use for this command is to prevent the main window of an application from ever appearing on the screen (in some applications the main window serves no purpose: the application presents a collection of windows any of which can be deleted independently from the others; if one of these windows were the main window, deleting it would delete all the other windows too). Once a window has been withdrawn, it can be returned to the screen with either `wm deiconify` or `wm iconify`.

The `wm state` command returns the current state for a window:

```
wm iconify .x
wm state .x
iconic
```

22.5 Decorations

When a window appears on the screen in the normal state, the window manager will usually add a decorative frame around the window. The frame typically displays a title for the window and contains interactive controls for resizing the window, moving it, and so on. For example, the window in Figure 22.1 was decorated by the `mwm` window manager.

The `wm title` command allows you to set the title that's displayed a window's decorative frame. For example, the command

```
wm title . "Berkeley Introduction"
```

was used to set the title for the window in Figure 22.1.

The `wm` command provides several options for controlling what is displayed when a window is iconified. First, you can use the `wm iconname` command to specify a title to display in the icon. Second, some window managers allow you to specify a bitmap to be displayed in the icon. The `wm iconbitmap` command allows you to set this bitmap, and `wm iconmask` allows you to create non-rectangular icons by specifying that certain bits of the icon are transparent. Third, some window managers allow you to use one window as the icon for another; `wm iconwindow` will set up such an arrangement if your window manager supports it. Finally, you can specify a position on the screen for the icon with the `wm iconposition` command.

Note: *Almost all window managers support* `wm iconname` *and* `wm iconposition` *but fewer support* `wm iconbitmap` *and almost no window managers support* `wm iconwindow` *very well. Don't assume that these features work until you've tried them with your own window manager.*

22.6 Window manager protocols

There are times when the window manager needs to inform an application that an important event has occurred or is about to occur so that the application can do something to deal with the event. In X terminology, these events are called *window manager protocols*.

The window manager passes an identifier for the event to the application and the application can do what it likes in response (including nothing). The two most useful protocols are `WM_DELETE_WINDOW` and `WM_SAVE_YOURSELF`. The window manager invokes the `WM_DELETE_WINDOW` protocol when it wants the application to destroy the window (e.g. because the user asked the window manager to kill the window). The `WM_SAVE_YOURSELF` protocol is invoked when the X server is about to be shut down or the window is about to be lost for some other reason. It gives the application a chance to save its state on disk before its X connection disappears. For information about other protocols, refer to ICCCM documentation.

The `wm protocol` command arranges for a script to be invoked whenever a particular protocol is triggered. For example, the command

```
wm protocol . WM_DELETE_WINDOW {
        puts stdout "I don’t wish to die"
}
```

will arrange for a message to be printed on standard output whenever the window manager asks the application to kill its main window. In this case, the window will not actually be destroyed. If you don’t specify a handler for `WM_DELETE_WINDOW` then Tk will destroy the window automatically. `WM_DELETE_WINDOW` is the only protocol where Tk takes default action on your behalf; for other protocols, like `WM_SAVE_YOURSELF`, nothing will happen unless you specify an explicit handler.

22.7 Special handling: transients, groups, and override-redirect

The window manager protocols allow you to request three kinds of special treatment for windows. First, you can mark a top-level window as *transient* with a command like the following:

```
wm transient .x .
```

This indicates to the window manager that `.x` is a short-lived window, such as a dialog box, working on behalf of the application’s main window. The last argument to `wm transient` (“`.`” in the example) is referred to as the *master* for the transient window. The window manager may treat transient windows differently e.g. by providing less decoration or by iconifying and deiconifying them whenever their master is iconified or deiconified.

In situations where a group of long-lived windows works together you can use the `wm group` command to tell the window manager about the group. The following script tells the window manager that the windows `.top1`, `.top2`, `.top3`, and `.top4` are working together as a group, and `.top1` is the group *leader*:

```
foreach i {.top2 .top3 .top4} {
        wm group $i .top1
}
```

The window manager can then treat the group as a unit, and it may give special treatment to the leader. For example, when the group leader is iconified, all the other windows in the group might be removed from the display without displaying icons for them: the leader's icon would represent the whole group. When the leader's icon is deiconfied again, all the windows in the group might return to the display also. The exact treatment of groups is up to the window manager, and different window managers may handle them differently. The leader for a group need not actually appear on the screen (e.g. it could be withdrawn).

In some extreme cases it is important for a top-level window to be completely ignored by the window manager: no decorations, no interactive manipulation of the window via the window manager, no iconifying, and so on. The best example of such a window is a pop-up menu. In these cases, the windows should be marked as *override-redirect* using a command like the following:

```
wm overrideredirect .popup
```

This command must be invoked before the window has actually appeared on the screen.

22.8 Session management

The `wm` command provides two options for communicating with session managers: `wm client` and `wm command`. These commands pass information to the session manager about the application running in the window; they are typically used by the session manager to display information to the user and to save the state of the session so that it can be recreated in the future. `Wm client` identifies the machine on which the application is running, and `wm command` identifies the shell command used to invoke the application. For example,

```
wm client . sprite.berkeley.edu
wm application . {browse /usr/local/bin}
```

indicates that the application is running on the machine `sprite.berkeley.edu` and was invoked with the shell command "`browse /usr/local/bin`".

22.9 A warning about window managers

Although the desired behavior of window managers is supposedly described in the X ICCCM document, the ICCCM is not always clear and no window manager that I am aware of implements everything exactly as described in the ICCCM. For example, the `mwm` window manager doesn't always deal properly with changes in the minimum and maximum sizes for windows after they've appeared on the screen, and the `twm` window manager treats the aspect ratio backwards; neither window manager positions windows on the screen in exactly the places they request. Tk tries to compensate for some of the deficiencies of window managers (e.g. it checks to see where the window manager puts a window and if it's the wrong place then Tk repositions it again to compensate for the window manager's error), but it can't compensate for all the problems.

One of the main sources of trouble is Tk's dynamic nature, which allows you to change anything anytime. Almost all applications (except those based on Tk) set all the information about a window before it appears on the screen and they never change it after that. Because of this, window manager code to handle dynamic changes hasn't been debugged very well. You can avoid problems by setting as much of the information as possible before the window first appears on the screen and avoiding changes.

Chapter 23
The Send Command

The selection mechanism described in Chapter 20 provides a simple way for one application to retrieve data from another application. This chapter describes the `send` command, which provides a more powerful form of communication between applications. With `send`, any Tk application can invoke arbitrary Tcl scripts in any other Tk application on the display; these commands can not only retrieve information but also take actions that modify the state of the target application. Table 23.1 summarizes `send` and a few other commands that are useful in conjunction with it.

23.1 Basics

To use `send`, all you have to do is give the name of an application and a Tcl script to execute in the application. For example, consider the following command:

```
send tgdb {break tkButton.c 200}
```

The first argument to `send` is the name of the target application (see Section 23.3 below for more on application names) and the second argument is a Tcl script to execute in that application. Tk locates the named application (an imaginary Tcl-based version of the `gdb` debugger in this case), forwards the script to that application, and arranges for the script to be executed in the application's interpreter. In this example the script sets a breakpoint at a particular line in a particular file. The result or error generated by the script is passed back to the originating application and returned by the `send` command.

`Send` is synchronous: it doesn't complete until the script has been executed in the remote application and the result has been returned. While waiting for the remote applica-

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

`send appName arg ?arg ...?`	Concatenates all the *arg*'s with spaces as separators, then executes the resulting script in the interpreter of the application given by *appName*. The result of that execution is returned as the result of the `send` command.
`winfo interps`	Returns a list whose elements are the names of all the applications available on the display containing the application's main window.
`winfo name .`	Returns the name of the current application, suitable for use in `send` commands issued by other applications.

Table 23.1. A summary of `send` and related commands.

tion to respond, `send` will defer the processing of X events, so the application will not respond to its user interface during this time. Once the `send` command completes and the application returns to normal event processing, any deferred events will be processed. A sending application *will* respond to `send` requests from other applications while waiting for its own `send` to complete. This means, for example, that the target of the `send` can send a command back to the initiator while processing the script, if that is useful.

23.2 Hypertools

I hope that `send` will enable a new kind of small re-usable application that I call *hypertools*. Many of today's windowing applications are monoliths that bundle several different packages into a single program. For example, debuggers often contain editors to display the source files being debugged, and spreadsheets often contain charting packages or communication packages or even databases. Unfortunately, each of these packages can only be used from within the monolithic program that contains it.

With `send` each of these packages can be built as a separate stand-alone program. Related programs can communicate by sending commands to each other. For example, a debugger can send a command to an editor to highlight the current line of execution, or a spreadsheet can send a script to a charting package to chart a dataset derived from the spreadsheet, or a mail reader can send a command to a multi-media application to play a video clip associated with the mail. With this approach it should be possible to re-use existing programs in many unforeseen ways. For example, once a Tk-based audio-video application becomes available, any existing Tk application can become a multi-media application just by extending with scripts that send commands to the audio-video application. The term "hypertools" reflects this ability to connect applications together in interesting ways and to re-use them in ways not foreseen by their original designers.

When designing Tk applications, I encourage you to focus on doing one or a few things well; don't try to bundle everything in one program. Instead, provide different functions in different hypertools that can be controlled via `send` and re-used independently.

23.3 Application names

In order to send to an application you have to know its name. Each application on the display has a unique name, which it can choose in any way it pleases as long as it is unique. In many cases the application name is just the name of the program that created the application. For example, `wish` will use the application name `wish` by default; or, if it is running under the control of a script file then it will use the name of the script file as its application name. In programs like editors that are typically associated with a disk file, the application name typically has two parts: the name of the application and the name of the file or object on which it is operating. For example, if an editor named `mx` is displaying a file named `tk.h`, then the application's name is likely to be "`mx tk.h`".

If an application requests a name that is already in use then Tk adds an extra number to the end of the new name to keep it from conflicting with the existing name. For example, if you start up `wish` twice on the same display the first instance will have the name `wish` and the second instance will have the name "`wish #2`". Similarly, if you open a second editor window on the same file it will end up with a name like "`mx tk.h #2`".

Tk provides two commands that return information about the names of applications. First, the command

```
winfo name .
wish #2
```

will return the name of the invoking application (this command is admittedly obscure; implement "`tk appname`" before the book is published!!). Second, the command

```
winfo interps
wish {wish #2} {mx tk.h}
```

will return a list whose elements are the names of all the applications defined on the display.

23.4 Security issues

The `send` command is potentially a major security loophole. Any application that uses your display can `send` scripts to any Tk application on that display, and the scripts can use the full power of Tcl to read and write your files or invoke subprocesses with the authority of your account. Ultimately this security problem must be solved in the X display server, since even applications that don't use Tk can be tricked into abusing your account by sufficiently sophisticated applications on the same display. However without Tk it is relatively difficult to create invasive applications; with Tk and `send` it is trivial.

You can protect yourself fairly well if you employ a key-based protection scheme for your display like `xauth` instead of a host-based scheme like `xhost`. Unfortunately, many people use the `xhost` program for protection: it specifies a set of machine names to the server and any process running on any of those machines can establish connections with the server. Anyone with an account on any of the listed machines can connect to your server, send to your Tk applications, and misuse your account.

If you currently use `xhost` for protection, you should learn about `xauth` and switch to it as soon as possible. `Xauth` generates an obscure authorization string and tells the server not to allow an application to use the display unless it can produce the string. Typically the string is stored in a file that can only be read by a particular user, so this restricts use of the display to the one user. If you want to allow other users to access your display then you can give them a copy of your authorization file, or you can change the protection on your authorization file so that it is group-readable. Of course, you should be aware that in doing so you are effectively giving these other users full use of your account.

Chapter 24
Modal Interactions

Usually the user of a Tk application has complete flexibility to determine what to do next. The application offers a variety of panels and controls and the user selects between them. However, there are times when it's useful to restrict the user's range of choices or force the user to do things in a certain order; these are called *modal interactions*. The best example of a modal interaction is a dialog box: the application is carrying out some function requested by the user (e.g. writing information to a file) when it discovers that it needs additional input from the user (e.g. the name of the file to write). It displays a dialog box and forces the user to respond to the dialog box (e.g. type in a file name). Once the user responds, the application completes the operation and returns to its normal mode of operation where the user can do anything he or she pleases.

Tk provides two mechanisms for use in modal interactions. First, the `grab` command allows you to temporarily restrict the user so that he or she can only interact with certain of the application's windows (e.g. only the dialog box). Second, the `tkwait` command allows you to suspend the evaluation of a script (e.g. saving a file) until a particular event has occurred (e.g. the user responded to the dialog box), and then continue the script once this has happened. These commands are summarized in Table 24.1.

24.1 Grabs

Mouse events such as button presses and mouse motion are normally delivered to the window under the mouse cursor. However, it is possible for a window to claim ownership of the mouse so that mouse events are only delivered to that window and its descendants in the Tk window hierarchy. This is called a *grab*. When the mouse is over one of the win-

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

`grab ?-global? window`	Same as `grab set` command described below.
`grab current ?window?`	Returns the name of the current grab window for *window*'s display, or an empty string if there is no grab for that display. If *window* is omitted, returns a list of all windows grabbed by this application for all displays.
`grab release window`	Releases the grab on *window*, if there is one.
`grab set ?-global? window`	Sets a grab on *window*, releasing any previous grab on *window*'s display. If `-global` is specified then the grab is global; otherwise it is local.
`grab status window`	Returns `none` if no grab is currently set on *window*, `local` if a local grab is set, and `global` if a global grab is set.
`tkwait variable varName`	Waits until variable *varName* changes value, then returns.
`tkwait visibility window`	Waits until the visibility state of *window* changes, then returns.
`tkwait window window`	Waits until *window* is destroyed, then returns.

Table 24.1. A summary of the `grab` and `tkwait` commands.

dows in the grab sub-tree, mouse events are delivered and processed just as if no grab were in effect. When the mouse is outside the grab sub-tree, button presses and releases and mouse motion events are delivered to the grab window instead of the window under the mouse, and window entry and exit events are discarded. Thus a grab prevents the user from interacting with windows outside the grab sub-tree.

The `grab` command sets and releases grabs. For example, if you've created a dialog box named `.dlg` and you want to restrict interactions to `.dlg` and its subwindows, you can invoke the command

```
grab set .dlg
```

Once the user has responded to the dialog box you can release the grab with the command

```
grab release .dlg
```

If the dialog box is destroyed after the user has responded to it then there's no need to invoke `grab release`: Tk releases the grab automatically when the grab window is destroyed.

Tk provides two forms of grab, local and global. A local grab affects only the grabbing application: if the user moves the mouse into some other application on the display then he or she can interact with the other application as usual. You should normally use local grabs, and they are the default in the `grab set` command. A global grab takes over the entire display so that you cannot interact with any application except the one that set the grab. To request a global grab, specify the `-global` switch to `grab set` as in the following command:

```
grab set -global .dlg
```

Global grabs are rarely needed and they are tricky to use (if you forget to release the grab your display will become unusable). One place where they are used is for pull-down menus.

Note: X will not let you set a global grab on a window unless it is visible. Section 24.3 describes how to use the `tkwait visibility` *command to wait for a window to become visible. Local grabs are not subject to the visibility restriction.*

The most common way to use grabs is to set a grab on a top-level window so that only a single panel or dialog box is active during the grab. However, it is possible for the grab sub-tree to contain additional top-level windows; when this happens then all of the panels or dialogs corresponding to those top-level windows will be active during the grab.

24.2 Keyboard handling during grabs

Local grabs have no effect on the way the keyboard is handled: keystrokes received anywhere in the application will be forwarded to the focus window as usual. Most likely you will set the focus to a window in the grab sub-tree when you set the grab. Windows outside the grab sub-tree can't receive any mouse events so they are unlikely to claim the focus away from the grab sub-tree. Thus the grab is likely to have the effect of restricting the keyboard focus to the grab sub-tree; however, you are free to move the focus anywhere you wish. If you move the mouse to another application then the focus will move to that other application just as if there had been no grab.

During global grabs Tk also sets a grab on the keyboard so that keyboard events go to the grabbing application even if the mouse is over some other application. This means that you cannot use the keyboard to interact with any other application. Once keyboard events arrive at the grabbing application they are forwarded to the focus window in the usual fashion.

24.3 Waiting: the tkwait command

The second aspect of a modal interaction is waiting. Typically you will want to suspend a script during a modal interaction, then resume it when the interaction is complete. For example, if you display a file selection dialog during a file write operation, you will probably want to wait for the user to respond to the dialog, then complete the file write using the name supplied in the dialog interaction. Or, when you start up an application you might wish to display an introductory panel that describes the application and keep this panel visible while the application initializes itself; before going off to do the main initialization you'll want to be sure that the panel is on the screen. The `tkwait` command can be used to wait in situations like these.

`Tkwait` has three forms, each of which waits for a different event to occur. The first form is used to wait for a window to be destroyed, as in the following command:

```
tkwait window .dlg
```

This command will not return until `.dlg` has been destroyed. You might invoke this command after creating a dialog box and setting a grab on it; the command won't return until after the user has interacted with the dialog in a way that causes it to be destroyed. While `tkwait` is waiting the application responds to events so the user can interact with the application's windows. In the dialog box example you should have set up bindings that destroy the dialog once the user's response is complete (e.g. the user clicks on the `OK` button). The bindings for the dialog box might also save additional information in variables (such as the name of a file, or an identifier for the button that was pressed). This information can be used once `tkwait` returns.

The script below creates a panel with two buttons labelled `OK` and `Cancel`, waits for the user to click on one of the buttons, and then deletes the panel:

```
toplevel .panel
button .panel.ok -text OK -command {
        set label OK
        destroy .panel
}
button .panel.cancel -text Cancel -command {
        set label Cancel
        destroy .panel
}
pack .panel.ok -side left
pack .panel.cancel -side right
grab set .panel
tkwait window panel
```

When the `tkwait` command returns the variable `label` will contain the label of the button that was clicked upon.

The second form for `tkwait` waits for the visibility state of a window to change. For example, the command

```
tkwait visibility .intro
```

will not return until the visibility state of `.intro` has changed. Typically this command is invoked just after a new window has been created, in which case it won't return until the window has become visible on the screen. `Tkwait visibility` can be used to wait for a window to become visible before setting a global grab on it, or to make sure that an introductory panel is on the screen before invoking a lengthy initialization script. Like all forms of `tkwait`, `tkwait visibility` will respond to events while waiting.

The third form of tkwait provides a general mechanism for implementing other forms of waiting. In this form, the command doesn't return until a given variable has been modified. For example, the command

```
tkwait variable x
```

will not return until variable x has been modified. This form of tkwait is typically used in conjunction with event bindings that modify the variable. For example, the following procedure uses tkwait variable to implement something analogous to tkwait window except that you can specify more than one window and it will return as soon as any of the named windows has been deleted (it returns the name of the window that was deleted):

```
proc waitWindows args {
        global dead
        foreach w $args {
            bind $w <Destroy> "set dead $w"
        }
        tkwait variable dead
        return $dead
}
```

Chapter 25
Odds and Ends

This chapter describes five additional Tk commands: `destroy`, which deletes widgets; `after`, which delays execution or schedules a script for execution later; `update`, which forces operations that are normally delayed, such as screen updates, to be done immediately; `winfo`, which provides a variety of information about windows, such as their dimensions and children; and `tk`, which provides access to various internals of the Tk toolkit. Table 25.1 summarizes these commands. This chapter also describes several global variables that are read or written by Tk and may be useful to Tk applications.

25.1 Destroying windows

The `destroy` command is used to delete windows. It takes any number of window names as arguments, for example:

```
destroy .dlg1 .dlg2
```

This command will destroy `.dlg1` and `.dlg2`, including all of their widget state and the widget commands named after the windows. It also recursively destroys all of their children. The command "`destroy .`" will destroy all of the windows in the application; when this happens most Tk applications (e.g. `wish`) will exit.

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

`after` *`ms`*	Delays for *`ms`* milliseconds.
`after` *`ms arg`* ?*`arg arg`* ...?	Concatenates all the *`arg`* values (with spaces as separators) and arranges for the resulting script to be executed after *`ms`* milliseconds have elapsed. Returns without waiting for the script to be executed.
`destroy` *`window`* ?*`window window`* ...?	Deletes each of the `windows`, plus all of the windows descended from them. The corresponding widget commands (and all widget state) are also deleted.
`tk colormodel` *`window`* ?*`value`*?	Sets the color model for *`window`*'s screen to value, which must be either `color` or `monochrome`. If *`value`* isn't specified, returns the current color model for *`window`*'s screen.
`update ?idletasks?`	Brings display up to date and processes all pending events. If `idletasks` is specified then no events are processed except those in the idle task queue (delayed updates).
`winfo` *`option`* ?*`arg arg`* ...?	Returns various pieces of information about windows, depending on *`option`* argument. See reference documentation for details.

Table 25.1. A summary of the commands discussed in this chapter.

25.2 Time delays

The `after` command allows you to incorporate timing into your Tk applications. It has two forms. If you invoke `after` with a single argument, then the argument specifies a delay in milliseconds, and the command delays for that number of milliseconds before returning. For example,

```
after 500
```

will delay for 500 milliseconds before returning. If you specify additional arguments, as in the command

```
after 5000 {puts "Time's up!"}
```

then the after command returns immediately without any delay. However, it concatenates all of the additional arguments (with spaces between them) and arranges for the resulting script to be evaluated after the specified delay. The script will be evaluated at global level as an event handler, just like the scripts for bindings. In the example above, a message will be printed on standard output after five seconds. The script below uses `after` to build a general-purpose blinking utility:

```
proc blink {w option value1 value2 interval} {
        $w config $option $value1
        after $interval [list blink $w $option \
            $value2 $value1 $interval]
}
blink .b -bg red black 500
```

The `blink` procedure takes five arguments, which are the name of a widget, the name of an option for that widget, two values for that option, and a blink interval in milliseconds. The procedure arranges for the option to switch back and forth between the two values at the given blink interval. It does this by immediately setting the option to the first value and then arranging for itself to be invoked again at the end of the next interval with the two option values reversed, so that option is set to the other value. The procedure reschedules itself each time it is called, so it executes periodically forever. `Blink` runs "in background": it always returns immediately, then gets reinvoked by Tk's timer code after the next interval expires.

25.3 The update command

Tk normally delays operations such as screen updates until the application is idle. For example, if you invoke a widget command to change the text in a button, the button will not redisplay itself immediately. Instead, it will schedule the redisplay to be done later and return immediately. When the application becomes idle (i.e. the current event handler has completed, plus all events have been processed, so that the application has nothing to do but wait for the next event) then it carries out all the delayed operations. Tk delays redisplays because it saves work in situations where a script changes the same window several different times: with delayed redisplay the window only gets redrawn once at the end. Tk also delays many other operations, such as geometry recalculations and window creation.

For the most part the delays are invisible. Tk rarely does very much work at a time, so it becomes idle again very quickly and updates the screen before the user can perceive any delay. However, there are times when the delays are inconvenient. For example, if a script is going to execute for a long time then you may wish to bring the screen up to date at certain times during the execution of the script. The `update` command allows you to do this. If you invoke the command

```
update idletasks
```

then all of the delayed operations like redisplays will be carried out immediately; the command will not return until they have finished.

The following procedure uses `update` to flash a widget synchronously:

```
proc flash {w option value1 value2 interval count} {
        for {set i 0} {$i < $count} {incr i} {
            $w config $option $value1
```

```
                update idletasks
                after $interval
                $w config $option $value2
                update idletasks
                after $interval
            }
    }
```

This procedure is similar to `blink` except that it runs in foreground instead of background: it flashes the option a given number of times and doesn't return until the flashing is complete. Tk never becomes idle during the execution of this procedure so the `update` commands are needed to force the widget to be redisplayed. Without the `update` commands no changes would appear on the screen until the script completed, at which point the widget's option would change to `value2`.

If you invoke `update` without the `idletasks` argument, then all pending events will be processed too. You might do this in the middle of a long calculation to allow the application to respond to user interactions (e.g. the user might invoke a "cancel" button to abort the calculation).

25.4 Information about windows

The `winfo` command provides information about windows. It has more than 40 different forms for retrieving different kinds of information. For example,

```
winfo exists .x
```

returns a `0` or `1` value to indicate whether there exists a window `.x`,

```
winfo children .menu
```

returns a list whose elements are all of the children of `.menu`,

```
winfo screenmmheight .dialog
```

returns the height of `.dialog`'s screen in millimeters, and

```
winfo class .x
```

returns the class of widget `.x` (e.g. `button`, `text`, etc.). Refer to the Tk reference documentation for details on all of the options provided by `winfo`.

25.5 The tk command: color models

The `tk` command provides access to various aspects of Tk's internal state. At present only one aspect is accessible: the *color model*. At any given time, Tk treats each screen as being either a color or monochrome screen; this is the screen's color model. When creating widgets, Tk will use different defaults for configuration options depending on the color model of the screen. If you specify a color other than black or white for a screen whose color model is monochrome, then Tk will round the color to either black or white.

By default Tk picks a color model for a screen based on the number of bits per pixel for that screen: if the screen has only a few bits per pixel (currently four or fewer) then Tk uses a monochrome color model; if the screen has many bits per pixel then Tk treats the screen as color. You can invoke the `tk` command to change Tk's color model from the default. For example, the following command sets the color model for the main window's screen to monochrome:

```
tk colormodel . monochrome
```

If the color model for a screen is color and Tk finds itself unable to allocate a color for a window on that screen (e.g. because the colormap is full) then Tk generates an error that is processed using the standard `tkerror` mechanism described in Section 19.7. Tk then changes the color model to monochrome and retries the allocation so the application can continue in monochrome mode. If the application finds a way to free up more colors, it can reset the color model back to color again.

25.6 Variables managed by Tk

Several global variables are significant to Tk, either because it sets them or because it reads them and adjusts its behavior accordingly. You may find the following variables useful:

`tk_version`	Set by Tk to its current version number. Has a form like 3.2, where 3 is the major version number and 2 is a minor version number. Changes in the major version number imply incompatible changes in Tk.
`tk_library`	Set by Tk to hold the path name of the directory containing a library of standard Tk scripts and demonstrations. This variable is set from the `TK_LIBRARY` environment variable, if it exists, or from a compiled-in default otherwise.
`tk_strictMotif`	If set to 1 by the application, then Tk goes out of its way to observe strict Motif compliance. Otherwise Tk deviates slightly from Motif (e.g. by highlighting active elements when the mouse cursor passes over them).

In addition to these variables, which may be useful to the application, Tk also uses the associative array `tk_priv` to store information for its private use. Applications should not use or modify any of the values in `tk_priv`.

Chapter 26
Examples

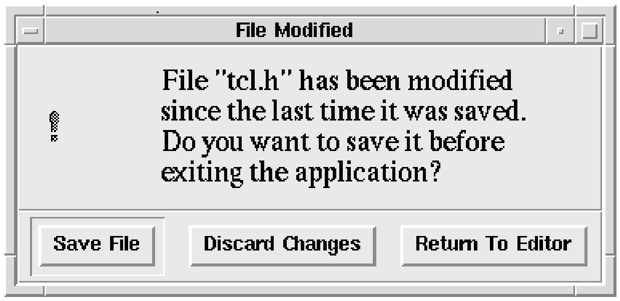
This chapter presents two relatively complete examples that illustrate many of the features of Tk. The first example is a procedure that generates dialog boxes, waits for the user to respond, and returns the user's response. The second example is an application that allows you to "remote-control" any other Tk application on the display: it connects itself to that application so that you can type commands to the other application and see the results.

26.1 A procedure that generates dialog boxes

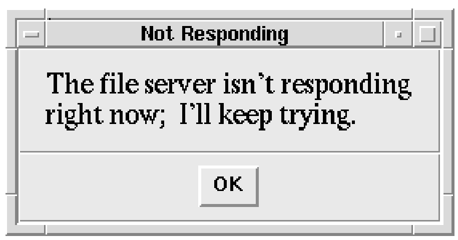
The first example is a Tcl procedure named `dialog` that creates dialog boxes like those shown in Figure 26.1. Each dialog contains a text message at the top plus an optional bitmap to the left of the text. At the bottom of the dialog box is a row of any number of buttons. One of the buttons may be specified as the default button, in which case it is displayed in a sunken frame. `Dialog` creates a dialog box of this form, then waits for the user to respond by clicking on a button. Once the user has responded, `dialog` destroys the dialog box and returns the index of the button that was invoked. If the user types a return and a default button was specified, then the index of the default button is returned. `Dialog` sets a grab so that the user must respond to the dialog box before interacting with the application in any other way.

Figures 26.2 and 26.3 show the Tcl code for `dialog`. It takes six or more arguments. The first argument, `w`, gives the name to use for the dialog's top-level window. The second argument, `title`, gives a title for the window manager to display in the dialog's decorative frame. The third argument, `text`, gives a message to display on the right side of the dialog. The fourth argument, `bitmap`, gives the name of a bitmap to display on the left

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.




```
dialog .d {File Modified} {File "tkInt.h" has been modified since the last
time it was saved. Do you want to save it before exiting the application?}
warning 0 {Save File} {Discard Changes} {Return To Editor}
```




```
dialog .d {Not Responding} {The file server isn't responding right
now; I'll keep trying.} {} -1 OK
```

Figure 26.1. Two examples of dialog boxes created by the `dialog` procedure. Underneath each dialog box is the command that created it.

side of the dialog; if it is specified as an empty string then no bitmap is displayed.The fifth argument, `default`, gives the index of a default button, or -1 if there is to be no default button. The sixth and additional arguments contain the strings to display in the buttons.

The code for `dialog` divides into five major parts, each headed by a comment. The first part of the procedure creates the dialog's top-level window. It sets up information for the window manager, such as the title for the window's frame and the text to display in the dialog's icon. Then it creates two frames, one for the bitmap and message at the top of the dialog, and the other for the row of buttons at the bottom.

The second part of `dialog` creates a message widget to hold the dialog's text string and a label widget to hold its bitmap, if any. The widgets are arranged on the right and left sides of the top frame, respectively, using the packer.

```
proc dialog {w title text bitmap default args} {
    global button

    # 1. Create the top-level window and divide it into top
    # and bottom parts.

    toplevel $w -class Dialog
    wm title $w $title
    wm iconname $w Dialog
    frame $w.top -relief raised -bd 1
    pack $w.top -side top -fill both
    frame $w.bot -relief raised -bd 1
    pack $w.bot -side bottom -fill both

    # 2. Fill the top part with the bitmap and message.

    message $w.top.msg -width 3i -text $text \
        -font -Adobe-Times-Medium-R-Normal-*-180-*
    pack $w.top.msg -side right -expand 1 -fill both \
        -padx 5m -pady 5m
    if {$bitmap != ""} {
        label $w.top.bitmap -bitmap $bitmap
        pack $w.top.bitmap -side left -padx 5m -pady 5m
    }

    # 3. Create a row of buttons at the bottom of the dialog.

    set i 0
    foreach but $args {
        button $w.bot.button$i -text $but -command \
            "set button $i"
        if {$i == $default} {
            frame $w.bot.default -relief sunken -bd 1
            pack $w.bot.default -side left -expand 1\
                -padx 5m -pady 2m
            pack $w.bot.button$i -in $w.bot.default -side left
\
                -padx 3m -pady 3m -ipadx 2m -ipady 1m
        } else {
            pack $w.bot.button$i -side left -expand 1 \
                -padx 5m -pady 5m -ipadx 2m -ipady 1m
        }
        incr i
    }
```

Figure 26.2. A Tcl procedure that generates dialog boxes with a text message, optional bitmap, and any number of buttons. Continued in Figure 26.3.

```
    # 4. Set up a binding for <Return>, if there's a default,
    # set a grab, and claim the focus too.

    if {$default > 0} {
        bind $w <Return> "$w.bot.button$default flash; \
            set button $default"
    }
    set oldFocus [focus]
    grab $w
    focus $w

    # 5. Wait for the user to respond, then restore the focus
    # and return the index of the selected button.

    tkwait variable button
    destroy $w
    focus $oldFocus
    return $button
}
```

Figure 26.3. Procedure to generate dialog boxes, cont'd.

The third part of the procedure creates the row of buttons. Since `args` was used as the name of the last argument to `dialog`, the procedure can take any number of arguments greater than or equal to five; `args` will be a list whose elements are all the additional arguments after `default`. For each of these arguments, `dialog` creates a button that displays the argument value as its text. The default button, if any, is packed in a special sunken ring (`$w.bot.default`). The buttons are packed with the `-expand` option so that they spread themselves evenly across the width of the dialog box; if there is only a single button then it will be centered. Each button is configured so that when the user clicks on it the global variable `button` will be set to the index of that button.

Note: It's important that the value of the `-command` *option is specified in quotes, not curly braces, so that* `$i` *(the button's index) is substituted into the command immediately. If the value were surrounded by braces, then the value of* `$i` *wouldn't be substituted until the command is actually executed; this would use the value of global variable* `i`*, not the variable* `i` *from the* `dialog` *procedure.*

The fourth part of `dialog` sets up a binding so that typing a return to the dialog box will flash the default button and set the `button` variable just as if the button had been invoked. It also sets the input focus to the dialog box and sets a local grab on the dialog box to give it control over both the keyboard and the mouse.

The last part of the procedure waits for the user to interact with the dialog. It does this by waiting for the `button` variable to change value, which will happen when the user

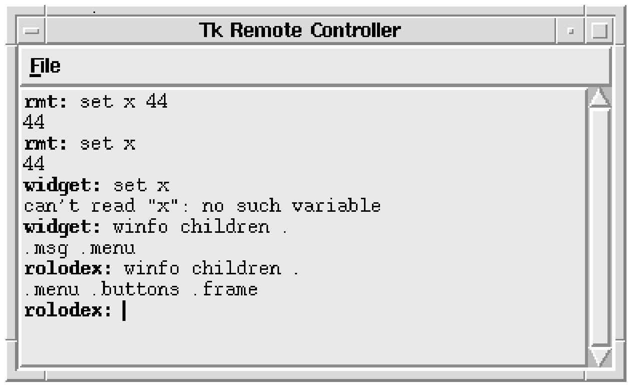


Figure 26.4. The `rmt` application allows users to type interactively to any Tk application on the display. It contains a menu for selecting an application plus a text widget for typing commands and displaying results. In this example the user has issued commands to three different applications: first the `rmt` application itself, then an application named `widget`, and finally one named `rolodex` (the prompt on each command line indicates the name of the application that executed the command).

clicks on a button in the dialog box or types a return. When the `tkwait` command returns, the `button` variable contains the index of the selected button. `Dialog` then destroys the dialog box (which also releases its grab), restores the input focus to its old window, and returns.

26.2 A remote-control application

The second example is an application called `rmt`, which allows you to type Tcl commands interactively to any Tk application on the display. Figure 26.4 shows what `rmt` looks like on the screen. It contains a menu that can be used to select an application plus a text widget and scrollbar. At any given time `rmt` is "connected" to one application; lines that you type in the text widget are forwarded to the current application using `send` and the results are displayed in the text widget. `Rmt` displays the name of the current application in the prompt at the beginning of each command line. You can change the current application by selecting an entry in the menu, in which case the prompt will change to display the new application's name. You can also type commands to the `rmt` application itself by selecting `rmt` as the current application. When `rmt` starts up it connects to itself.

The script that creates `rmt` is shown in Figures 26.5-26.9. The script is designed to be placed into a file and executed directly. The first line of the script,

```
#!/usr/local/bin/wish -f

# 1. Create basic application structure:  menu bar on top of
# text widget, scrollbar on right.

frame .menu -relief raised -bd 2
pack .menu -side top -fill x
scrollbar .s -relief flat -command ".t yview"
pack .s -side right -fill y
text .t -relief raised -bd 2 -yscrollcommand ".s set" \
    -setgrid true
.t tag configure bold -font *-Courier-Bold-R-Normal-*-120-*
pack .t -side left -fill both -expand 1
wm title . "Tk Remote Controller"
wm iconname . "Tk Remote"
wm minsize . 1 1

# 2. Create menu button and menus.

menubutton .menu.file -text "File" -underline 0 -menu
.menu.file.m
menu .menu.file.m
.menu.file.m add cascade -label "Select Application" \
    -underline 0 -accelerator => -menu .menu.file.m.apps
.menu.file.m add command -label "Quit" -underline 0 \
    -command "destroy ."
menu .menu.file.m.apps -postcommand fillAppsMenu
pack .menu.file -side left
tk_menuBar .menu .menu.file
proc fillAppsMenu {} {
    catch {.menu.file.m.apps delete 0 last}
    foreach i [lsort [winfo interps]] {
        .menu.file.m.apps add command -label $i \
            -command [list newApp $i]
    }
}
```

Figure 26.5. A script that generates `rmt`, an application for remotely controlloing other Tk applications. This figure contains basic window set-up code. The script continues in Figures 26.6-26.9

```
#!/usr/local/bin/wish -f
```

is similar to the first line of a shell script: if you invoke the script file directly from a shell then the operating system will invoke `wish` instead, passing it two arguments: `-f` and the name of the script file. `Wish` will then execute the contents of the file as a Tcl script.

```
# 3. Create bindings for text widget to allow commands to
# be entered and information to be selected.  New characters
# can only be added at the end of the text (can't ever move
# insertion point).

bind .t <1> {
    set tk_priv(selectMode) char
    .t mark set anchor @%x,%y
    if {[lindex [%W config -state] 4] == "normal"} {focus %W}
}
bind .t <Double-1> {
    set tk_priv(selectMode) word
    tk_textSelectTo .t @%x,%y
}
bind .t <Triple-1> {
    set tk_priv(selectMode) line
    tk_textSelectTo .t @%x,%y
}
bind .t <Return> {.t insert insert \n; invoke}
bind .t <BackSpace> backspace
bind .t <Control-h> backspace
bind .t <Delete> backspace
bind .t <Control-v> {
    .t insert insert [selection get]
    .t yview -pickplace insert
    if [string match *.0 [.t index insert]] {
        invoke
    }
}
```

Figure 26.6. Bindings for the `rmt` application. These are modified versions of the default Tk bindings, so they use existing Tk facilities such as the variable `tk_priv` and the procedure `tk_textSelectTo`

The `rmt` script contains about 100 lines of Tcl code in all, which divide into seven major parts. It makes extensive use of the facilities of text widgets, including marks and tags; you may wish to review the reference documentation for texts as you read through the code for `rmt`.

The first part of the `rmt` script sets up the overall window structure, consisting of a menu bar, a text widget, and a scrollbar. It also passes information to the window manager, such as titles to appear in the window's decorative frame and icon. The command "`wm minsize . 1 1`" enables interactive resizing by the user as described in Section 22.1. Since the text widget has been packed with the `-expand` option set to 1, it will receive any extra space; since it is last in the packing order, it will also shrink if the user resizes

```
# 4. Procedure to backspace over one character, as long as
# the character isn't part of the prompt.

proc backspace {} {
    if {[.t index promptEnd] != [.t index {insert - 1 char}]}
{
        .t delete {insert - 1 char} insert
        .t yview -pickplace insert
    }
}
```

Figure 26.7. Procedure that implements backspacing for `rmt`.

the application to a smaller size than it initially requested. The `-setgrid` option for the text widget enables gridding as described in Section 22.2: interactive resizing will always leave the text widget with dimensions that are an integral number of characters.

The command

```
.t tag configure bold -font \
        *-Courier-Bold-R-Normal-*-120-*
```

creates a *tag* named `bold` for the text widget and associates a bold font with that tag. The script will apply this tag to the characters in the prompts so that they appear in boldface, whereas the commands and results appear in a normal font.

The second part of the script fills in the menu with two entries. The top entry displays a cascaded submenu with the names of all applications, and the bottom entry is a command entry that causes `rmt` to exit (it executes the script "`destroy .`", which destroys all of the application's windows; when `wish` discovers that it no longer has any windows left then it exits). The cascaded submenu is named `.menu.file.m.apps`; its `-postcommand` option causes the script "`fillAppsMenu`" to be executed each time the submenu is posted on the screen. `FillAppsMenu` is a Tcl procedure defined at the bottom of Figure 26.5; it deletes any existing entries in the submenu, extracts the names of all applications on the display with "`winfo interps`", and creates one entry in the menu for each application name. When one of these entries is invoked by the user, the procedure `newApp` will be invoked with the application's name as argument.

Note: *The command "*`[list newApp $i]`*" creates a Tcl list with two elements. As described in Section XXX, when a list is executed as a command each element of the list becomes one word for the command. Thus this form guarantees that newApp will be invoked with a single argument consisting of the value of* `$i` *at the time the menu entry is created, even if* `$i` *contains spaces or other special characters.*

The third part of the `rmt` script, shown in Figure 26.6, creates event bindings for the text widget. Tk defines several default bindings for texts, which handle mouse clicks,

```
# 5. Procedure that's invoked when return is typed:  if
# there’s not yet a complete command (e.g. braces are open)
# then do nothing.  Otherwise, execute command (locally or
# remotely), output the result or error message, and issue
# a new prompt.

proc invoke {} {
     global app
     set cmd [.t get {promptEnd + 1 char} insert]
     if [info complete $cmd] {
         if {$app == [winfo name .]} {
             catch [list uplevel #0 $cmd] msg
         } else {
             catch [list send $app $cmd] msg
         }
         if {$msg != ""} {
             .t insert insert $msg\n
         }
         prompt
     }
     .t yview -pickplace insert
}

proc prompt {} {
     global app
     .t insert insert "$app: "
     .t mark set promptEnd {insert - 1 char}
     .t tag add bold {insert linestart} promptEnd
}
```

Figure 26.8. Procedures that execute commands and output prompts for `rmt`.

character insertion, and common editing keystrokes such as backspace. However, `rmt`’s text widget has special behavior that is inconsistent with the default bindings, so the code in Figure 26.6 overrides many of the defaults. You don’t need to understand the details of the bindings; they have been copied from the defaults in Tk’s startup script and modified so that (a) the user can’t move the insertion cursor (it always has to be at the end of the text), (b) the procedure `backspace` is invoked instead of Tk’s normal text backspace procedure, and (c) the procedure `invoke` is called whenever the user types a return or copies in text that ends with a newline.

The fourth part of the `rmt` script is a procedure called `backspace`. It implements backspacing in a way that disallows backspacing over the prompt (see Figure 26.7). `Backspace` checks to see if the character just before the insertion cursor is the last character of the most recent prompt. If not, then it deletes the character; if so, then it does noth-

```
# 6. Procedure to select a new application.  Also changes
# the prompt on the current command line to reflect the new
# name.

proc newApp appName {
    global app
    set app $appName
    .t delete {promptEnd linestart} promptEnd
    .t insert promptEnd "$appName:"
    .t tag add bold {promptEnd linestart} promptEnd
}

# 7. Miscellaneous initialization.

set app [winfo name .]
prompt
focus .t
```

Figure 26.9. Code to select a new application for rmt, plus miscellaneous initialization code.

ing, so that the prompt never gets erased. To keep track of the most recent prompt, `rmt` sets a *mark* named `promptEnd` at the position of the last character in the most recent prompt (see the `prompt` procedure below for the code that sets `promptEnd`).

The fifth part of the `rmt` script handles command invocation; it consists of two procedures, `invoke` and `prompt` (see Figure 26.8). The `invoke` procedure is called whenever a newline character has been added to the text widget, either because the user typed a return or because the selection was copied into the widget and it ended with a newline. Invoke extracts the command from the text widget (everything from the end of the prompt to the current insertion point) and then invokes `info complete` to make sure that the command is complete. If the command contains unmatched braces or unmatched quotes then `invoke` returns without executing the command so the user can enter the rest of the command; after each return is typed `invoke` will check again, and once the command is complete it will be invoked. The command is invoked by executing it locally or sending it to the appropriate application. If the command returns a non-empty string (either as a normal reult or as an error message) then the string is added to the end of the text widget. Finally, `invoke` outputs a new prompt and scrolls the view in the text to keep the insertion cursor visible.

The `prompt` procedure is responsible for outputting prompts. It just adds characters to the text widget, sets the `promptEnd` mark to the last character in the prompt, and then applies the `bold` tag to all the characters in the prompt so that they'll appear in a bold font.

The sixth part of the `rmt` script consists of the `newApp` procedure in Figure 26.9. `NewApp` is invoked to change the current application. It sets the global variable `app`, which identifies the current application, then overwrites the most recent prompt to display the new application's name.

The last part of `rmt` consists of miscellaneous initialization (see Figure 26.9). It connects the application to itself initially, outputs the initial prompt, and sets the input focus to the text window.