Bonus Chapter B

Programming for X

In this chapter and the next, we'll take a look at writing programs to run in the usual Linux graphical environment, the **XWindow System** or **X**, http://www.x.org/Xorg.html. Modern UNIX systems and nearly all Linux distributions are shipped with a version of X.

We'll be concentrating on the programmer's view of X, and we'll assume that you are already comfortable with configuring, running, and using X on your system.

We'll cover

- X concepts
- X Windows managers
- X programming model
- Tk—its widgets, bindings, and geometry managers

In the next chapter, we'll move on to the GTK+ toolkit, which will allow us to program user interfaces in C for the GNOME system.

What Is X?

X was created at MIT as a way of providing a uniform environment for graphical programs. Nowadays it should be fair to assume that if you've used computers, you've come across either Microsoft Windows, X, or Apple MacOS before, so you'll be familiar with the general concepts underlying a graphical user interface, or GUI. Unfortunately, although a Windows user might be able to navigate around the Mac interface, it's a different story for programmers.

Each windowing environment on each system is programmed differently. The ways that the display is handled and the programs communicate with the user are different. Although each system provides the programmer with the ability to open and manipulate windows on the screen, the functions used will be different. Writing applications that can run on more than one system (without using additional toolkits) is a daunting task.
To overcome the problems associated with proprietary interface systems on mainframes, minicomputers, and workstations, The XWindow System was made publicly available and has been implemented on many systems. It defines a programming style based on a client/server model with a clean separation of hardware-dependent components and application programs.

The X Windows system comprises four major components, which we'll discuss briefly in turn:

- X server — Interacting with the user
- X protocol — Client/server communications
- X library — The programming interface
- X clients — The applications

X Server

The X server, or X display server, is a program that runs on the application user's computer and is responsible for controlling the graphical display hardware and looking after input and output. The X server responds to requests from X client applications to draw on the screen or read from the keyboard or mouse. It passes input and indications of things like mouse movements and button presses to the client programs.

Typically, there will be a different X server for each distinct hardware combination that X can run on. The most common implementation of X for Linux and other PC-based systems is XFree86 (http://www.xfree86.org). This package ships with X servers specially created for the many different video cards that can be used in PCs, for instance, the `XF86_S3` version for S3-based cards. Linux users have much to thank these guys for.

X Protocol

All interactions between X client applications and the X display server take place via message exchanges. The types and uses of the messages form the X protocol. One particularly useful feature of the X Windows system is that the X protocol can be carried across a network as well as between clients and a server running on the same machine. This means that a user with a fairly low-powered personal computer or an X terminal (a machine dedicated to running just an X server) can run X client programs on more powerful networked computers, but conduct the interaction and display the output on his/her own local machine.

Xlib

The X protocol is really of interest only to the programmers who actually implement X servers. Most X applications ultimately use a C function library as a programming interface. This is Xlib, which provides an API for X protocol exchanges. Xlib doesn't add very much on its own—it can just about only draw on the screen and respond to a mouse. If you want menus, buttons, scrollbars, and all the other goodies, you have to write them yourself.

On the other hand, neither does Xlib impose any particular GUI style. It acts as a medium through which you can create any style you choose.

X Clients

X clients are application programs that use the display and input resources of a computer that may not be the one they're running on. They do this by requesting access to these resources from the X server that manages them. The server can typically handle requests from many clients at once, and it must arbitrate use of the keyboard and mouse between clients. The client programs communicate with the server using X protocol messages that are sent and received using Xlib functions.

X Toolkits

We won't linger in the Xlib programming interface, as it's not the best tool for creating programs quickly and easily. Because of its low-level interface, like the Microsoft Windows SDK, it can make for some very complex programs that apparently achieve very little. One book on the author's shelves contains a version of the "Hello World" program written for Xlib. It does nothing other than display "Hello World" in a window, together with a button marked "Exit," which does the obvious thing when you press it. The program listing runs to five pages!

Any programmer who has written an Xlib program like this will surely have wondered if there's a better way. Of course there is! Common user interface elements such as buttons, scrollbars, and menus have been implemented many times. Collections of these elements, also known as **widgets**, are generally called **X toolkits**. Of these, the best known are the Xt Intrinsics suite that comes with X and two commercial products: Sun's OpenLook and OSF/Motif.

- **Xt** is a free library written on top of X to give it some functionality: an intermediate layer that simplifies application programming.
- **OpenLook** is a free toolkit from Sun that enforces a different look and feel. It's built on top of a library called Xview, which is similar to Xt.
- **Motif** is an OSF standard designed to bring a common look and feel to the UNIX desktop. It's built on top of Xt. Motif has two main components: a set of include files that define constants used in Xt functions and a library of convenient functions to simplify the creation of elements like dialogs and menus. Motif also defines a programming style that all programmers can follow, whether they are actually using the Motif toolkit or not.
- **Qt** is a library built by trolltech that forms the basis of the **KDE** Desktop environment, which is found with most Linux distributions.
- **GTK+** is the GIMP toolkit, and the basis of the **GNOME** system. We'll look at how to program this high-level environment in the next chapter.

Each X toolkit implements a set of widgets, usually with a distinctive look and feel. Display elements might have a flat, plain implementation (as with Xt) or a sculpted, 3D effect (like Motif).
To illustrate the difference a toolkit can make, take a look at two different text editors available for Linux, `xedit` and `textedit`. The first, `xedit`, is a very simple editor with hardly any user interface sophistication. To load a file, you need to type a filename into a box and press a button marked `Load`.

In contrast, the `textedit` editor provided by Sun's OpenWindows and written with the OpenLook toolkit provides a dialog box for opening files. This allows the user to browse the file system for the appropriate file to open. The toolkit also provides the familiar look and feel of 3D buttons.

X Window Manager

Another important element of any X system is the **window manager**. This is a special X client that is responsible for dealing with other clients. It looks after the placement of client windows on the display

and handles management tasks like moving and resizing windows. It also imposes a distinctive look and feel, depending on the X toolkit it uses.

Examples of window managers follow:

Window Manager	Description
`twm`	Tom's (or Tabbed) Window Manager, a small, fast manager that comes with X.
`fvwm`	An alternative window manager by Robert Nation. The favorite under Linux. It supports virtual desktops and has configuration files that allow it to emulate other window managers.
`fvwm95`	A version of `fvwm` that emulates the Windows 95 interface.
`gwm`	The generic window manager, programmable in a LISP dialect.
`olwm`	The OpenLook window manager.
`mwm`	The Motif window manager.

All of these are available for most UNIX and Linux systems, although `mwm` requires a license.

The X Programming Model

We've seen that the XWindow System separates responsibilities between client applications and X display servers using a communications protocol. This method of programming gives rise to a typical structure for an X application, which we'll outline briefly below.

Start-up

A typical X application will start by initializing any resources it may need. It will establish a connection with the X display server, choose which colors and fonts to use, and then create a window on the display.

`XOpenDisplay` and `XCloseDisplay` are used by client programs for connecting to and disconnecting from an X server.

```
Display *XOpenDisplay(char *display_name);
void XCloseDisplay(Display *display);
```

The `display_name` specifies the display to which we want to connect. If it's `null`, the environment variable `DISPLAY` is used. This is of the form `hostname:server[.display]`, allowing one or more X servers on a host, each of which can control more than one display. The default display is normally `:0.0`, the first available server on the local machine. To specify a second screen, for a truly awesome desktop, you would use `:0.1`.

`XOpenDisplay` returns a `Display` structure containing information about the X server selected, or `null` if no X server could be opened. After a successful return from `XOpenDisplay`, the client program may start using the X server.

When the client program has finished using the X server, it must call `XCloseDisplay` with the display structure returned from the `XOpenDisplay` call. This will destroy all windows and other resources that the client has created on the display, unless (unusually) `XSetCloseDownMode` has been called to modify the shutdown behavior. Programs should always call `XCloseDisplay` before exiting to allow any pending errors to be reported.

The user can control most of the activities at start-up. Many X applications respond to command line arguments, environment variables, and configuration file entries to allow the user to customize the application. We'll give you some examples.

As we've seen, the environment variable `DISPLAY` is used to direct the application to a particular display server, which may be on a different networked computer. The following command would cause the `xedit` program to run, but to open its display on the machine called `alex`.

```
$ DISPLAY=alex:0.0 xedit &
```

The file `.Xresources` (or sometimes `.Xdefaults`) is used to configure the X application. Each application will use configuration entries in the X resources database, typically created when an X system starts up and including the user's own, local preferences. A typical entry in a user's `.Xresources` file, stored in his or her home directory, might be

```
xedit*enableBackups: on
```

This entry changes the behavior of `edit` with respect to making backup files while editing. Each entry has the general format

```
Class*Resource: Value
```

The command line

```
$ xedit -geometry 400x200
```

causes `xedit` to start in a window 400 pixels wide by 200 high. Note that other programs may use the geometry differently. For example,

```
$ xterm -geometry 80x50
```

starts a terminal emulator that has 50 lines, each with 80 columns. Refer to your system documentation and application manual pages for more details on ways to affect X application behavior.

Main Loop

The bulk of an X application is made up of a main loop and code written to react to events. After starting, a typical X program waits for the X display server to which it's connected to send it events. It does this by calling `XNextEvent` in a loop.

There are over 30 events that an application may have to deal with. We won't cover them here because there are many (very fat) books on the topic of X Windows programming that cover the topic in great depth. However, we'll get a flavor of the kinds of events that X uses from this partial list:

Keyboard events	Key pressed, key released.
Mouse events	Button pressed, button released, mouse moving, mouse entering/leaving a window.
Window events	Window created/destroyed, window gained/lost focus, window exposed.

A low-level X program must respond to these events and more. A program that uses an advanced toolkit or application framework will be able to concentrate on the main business of the application and use sophisticated interface elements like dialog boxes without needing to deal with low-level events like these explicitly. Of course, that doesn't mean that the events aren't still taking place.

Clean-up

When it exits, a well-behaved X program will free up any X display resources it has allocated while it was running. It's often sufficient to simply break the connection with the server, but this can result in the server consuming more memory than required. Also, it's considered a little rude not to say goodbye!

Fast-Track X Programming

In the rest of this chapter, we'll leave the low-level considerations of X programming to those who need to squeeze the ultimate performance from and have the finest control over their applications.

For the rest of us who are simply keen to see immediate results and to produce good-looking highly functional X applications, we'll concentrate on a couple of recent innovations in the X programming world.

With the rise of very fast personal computers and workstations, it has become feasible to write at least the user interface part of programs in an interpreted language. We've seen a couple of these already in the shell and Tcl. We've got the power of Perl to look forward to in now downloadable Chapter D

We'll now take a look at Tk (for Tool Kit), an extension to Tcl for graphical programming, and in the next chapter GTK+, developed originally as a toolkit for controlling the GIMP (GNU Image Processor) but which forms the underlying graphical language in the GNOME desktop.

The Tk approach to X programming also brings the benefit of portability. It is available for non-X graphical environments (including Microsoft Windows) and is hardware independent. Tk programs written for one machine should run unchanged on another.

If you are interested in the benefits of a platform-independent programming system and are also looking for the power of a compiled language, then Java provides an interesting solution. The topic of Java programming is too vast to cover here, but Ivor Horton's *Beginning Java 2*, also from Wrox (ISBN 1-861002-23-8), is an excellent place to start.

The Tk Toolkit

Tk, created by John Ousterhout to be the companion of Tcl, is a rich collection of graphical user interface (GUI) abstractions (widgets) designed to simplify the essential components of graphical front-end programming under X, Microsoft Windows, and Apple MacOS.

Tk is an action-oriented, composition-based, embeddable, extensible, highly portable, event-based toolkit whose widgets are written in C and use Tcl bindings for event handlers. Tk has already been ported to use many other languages such as Perl and Python for command bindings.

The current releases of Tk 8.1 and Tk 8.2 work consistently on all the three platforms: Unix, Windows, and Macintosh.

By default, Tk's widgets have the native look and feel of the widgets of the platform they run on, but they are highly-configurable. You can operate Tk's widgets in strict Motif mode by checking one of toolkit's global variables. Because Tk's interface is consistent, most scripts written for one platform will run without any modifications on the other two platforms.

> **All the examples in this section need at least version 8.0 of Tcl and 8.0 of Tk to work. You can download the latest versions of the software from http://www.scriptics.com/resource/software/. Most of the programs in this section are written using Tcl8.0 and Tk8.0 because the latest releases of Jacl and Tcl Blend work only with the Tcl8.0 version. Jacl is the complete rewrite of Tcl interpreter in pure Java, and Tcl Blend is a dynamically loadable C extension to Tcl to interact with a Java Virtual Machine.**

Before we dive into Tk programming, you need to make sure that the Tk windowing shell, `wish`, is installed on your system with the executable in your `PATH`. If Tk is not installed at the default location, you'll need to set the environment variables `TK_LIBRARY` and `TCL_LIBRARY` to point to the right locations. If you have multiple versions of Tcl installed on your machine, you might want to make sure that you point the above-mentioned environmental variables correctly. For example, here is the shell script I use to invoke `wishf` for version 8.2b3 of Tk.

```
#!/bin/sh
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/tcl8.2b3/unix:/usr/local/tk8.2b3
/uni
x:
PATH=$PATH:/usr/local/bin:/usr/local/tcl8.2b3/unix:/usr/local/tk8.2b3/unix:
TCL_LIBRARY=/usr/local/tcl8.2b3/library
TK_LIBRARY=/usr/local/tk8.2b3/library
export LD_LIBRARY_PATH PATH TCL_LIBRARY TK_LIBRARY
/usr/local/tk8.2b3/unix/wish $*
```

> **When you type `wish` at the shell prompt, a small and (by default) gray window should pop up. To suppress this interactive `wish` window—for example, when running scripts—invoke Tk using `wish -f`. `wish` is basically `tclsh` with the Tk functions built in.**

Over the next few sections, we'll take a look at

- Basic concepts of Windows programming
- Writing our first Tk program
- Touring the Tk widget set, with some of their configuration options
- Bindings
- Geometry managers
- Application resource management
- Inter-application communication
- Window management and application embedding
- A mega-widget completely wrtten in Tcl using existing Tk components
- A real Tk example for everyday use using Tcl events

Though by no means extensive, the discussion should show you how to get started with Tk, how to program using its built-in widget set, and where to find out more about Tk as you need it.

Windows Programming

Using Tk, you can quickly create a graphical interface using the widgets provided to deal with the underlying window system. You then attach event handlers to these widgets (usually using the Tcl language) so that they react as required to the user's commands. This fits in with usual visual programming practice.

First, create the look of your program, adding in components to access the functionality you eventually want to include. Select a widget from the Tk toolkit, initialize its look, and then use Tk's geometry manager to arrange it in relation to other widgets within the screen window. Then code the response of each GUI component to user commands. What will clicking the `Load` button do? How will clicking it load an image into the canvas widget?

This is rather different from the procedural programming we've been looking at throughout the rest of the book. As a programmer, you can never control the order in which the user interacts with the program; the whole point of GUIs is to provide a more natural and intuitive user interface. The program must wait for and then act on user-initiated events.

Every time you create a widget in Tk using its `class` command, the GUI primitive known as the **widget** is created, as is a new Tcl **command** whose name is the name of the widget. You can then invoke methods (Tk calls them **configuration options**) on this newly created widget using its widget command. Widget commands are like objects in an object-oriented system: When the widget is deleted, the widget command is also deleted.

For example, the widget command

```
button .b
```

creates a new widget and a widget command named `.b`. You can use this new command to communicate with the widget; thus,

```
.b configure -text "Hello"
```

will set the title of the button `.b` to `Hello`. If you think of `.b` as an object, you are invoking the `configure` method on the object to set its `text` attribute to `Hello`.

> **Tk widgets are not completely object oriented, as they don't support inheritance, polymorphism, and so on. Their only similarity to OOP principles is the way methods are invoked.**

The widget creation and initialization parts of a Tk program will contain Tcl commands to create and arrange the widgets on the user screen. Once you've created and arranged the widgets, they interact with the user using Tcl scripts known as **event handlers**.
We'll try to present all the examples using this approach; however, it's sometimes difficult to separate the two stages completely, because event handlers are sometimes bound when you create the widget. For example, most widgets in Tk support the `command` widget handler, which is usually set immediately. Also, it sometimes makes sense to bind the event handlers as soon as you create the widget and manage the screen layout of the widgets later. No single approach is the best. To manage your design, use whichever is appropriate or is easier to understand.

To get us started and make more sense of this introduction, let's look at a program, `hello1.tk`, probably the smallest multiline label program ever created!

Try It Out—Saying Hello

Type in the following script file:

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

pack [button .b -text "Hello\nWorld!!!" \
                -justify center \
                -width 20 \
                -command {puts "Hi"}]
```

Make the script executable and run the `hello1.tk` program:

```
$ ./hello1.tk
```

This program creates the window shown in the figure and outputs the string `Hi` each time you click the button.

How It Works

Let's dissect the program and see what's going on in this ubiquitous masterpiece.

After invoking `wish -f`, we get to the single line that does all the work. It's remarkably terse, and we've expanded it to make the specified options more obvious. Ignoring the `pack` command for now, we see that `button...` creates a button named `.b` whose multiline label `Hello World!!!` is centered. The button is set to a width of 20 characters. The `-command` option attaches an event handler to the button to output the string "`Hi`" in the parent window when the user clicks the button. Note the backslashes that allow you to write the command over several lines.

The `pack` command packs the widget `.b` into the default top-level window created by the application, so that it occupies the window. Note that `pack [button .b ...]` would work just as well if we initialized the button `.b` first and then called `pack .b`.

There's no particular reason for calling the widget `.b`. You can name it `.foo` or anything else, provided the name begins with a period. An application's widgets are arranged in a hierarchy, and the default top-level, "application" widget and its corresponding widget command are named "`.`". Each widget's name is a dot-separated list describing its position in the application's hierarchy. For example, the path name `.a.b.c` implies that widget `.c` is a child of `.a.b`, a grandchild of `.a`, and a great-grandchild of the application widget. Currently, all the widgets in Tk can have any number of children, provided all the path names are listed in this way.

Configuration Files

Now let's add one more line before the widget's creation:

```
option add *b.activeForeground brown
```

The program creates the widget `.b` and sets up its default `activeForeground` color to brown. The asterisk before `b` means that any widget called `b` should have the option set, no matter what its parentage.

We can also make it into a more realistic X application by saving the line

```
*b.activeForeground: brown
```

into a file called `hello.def`, and then adding the following line into the `hello3.tk` script before we create the widget:

```
option readfile hello.def
```

This line reads the application defaults from the file `hello.def` before the widget's creation.

More Commands

You might be thinking, "Can't I create more user interactions to the widget than `-command`?" We'll go right ahead and create one such simple **event binding**. If the user presses Ctrl along with the mouse button, the widget will output the string "`Help!`"

Here's how to do this:

```
bind .b <Control-Button-1> {puts "Help!"}
```

Our final Hello World program, `hello4.tk`, with all these modifications, reads

```
#!wish -f

option readfile hello.def
pack [button .b -text "Hello\nWorld!!!" \
                -justify center \
                -width 20 \
                -command {puts "Hi"}]
bind .b <Control-Button-1> {puts "Help!"}
```

This is a simple three-line program that can do the same job as a 500-line Xlib program or 100+ lines of Motif code. It has all the features of a basic X application and is still very simple. That's what Tk is all about. It removes all the complexity and fear involved in graphical user interface programming.

Tk Widgets

It's time to look more closely at the set of widgets Tk provides. Before we review the widgets Tk supports, though, here's a simple way to find out all the methods and arguments a widget provides. Note that the symbol `%` denotes Tk's `wish` command shell prompt.

Try It Out—Learning More

First, interactively create a scale widget `.s`:

```
$ wish
% scale .s
.s
```

Call the `config` method of the widget and see its output to check out what the widget offers:

```
% .s config
```

You should see this output:

```
{-activebackground activeBackground Foreground SystemButtonFace SystemButtonFace} {-
background background Background SystemButtonFace SystemButtonFace} {-bigincrement
bigIncrement BigIncrement 0 0.0} {-bd -borderwidth} {-bg -background} {-borderwidth
borderWidth BorderWidth 2 2} {-command command Command {} {}} {-cursor cursor Cursor
{} {}} {-digits digits Digits 0 0} {-fg -foreground} {-font font Font {{MS Sans Serif}
8} {{MS Sans Serif} 8}} {-foreground foreground Foreground SystemButtonText
SystemButtonText} {-from from From 0 0.0} {-highlightbackground highlightBackground
HighlightBackground SystemButtonFace SystemButtonFace} {-highlightcolor highlightColor
HighlightColor SystemWindowFrame SystemWindowFrame} {-highlightthickness
highlightThickness HighlightThickness 2 2} {-label label Label {} {}} {-length length
Length 100 100} {-orient orient Orient vertical vertical} {-relief relief Relief flat
flat} {-repeatdelay repeatDelay RepeatDelay 300 300} {-repeatinterval repeatInterval
RepeatInterval 100 100} {-resolution resolution Resolution 1 1.0} {-showvalue
showValue ShowValue 1 1} {-sliderlength sliderLength SliderLength 30 30} {-
sliderrelief sliderRelief SliderRelief raised raised} {-state state State normal
normal} {-takefocus takeFocus TakeFocus {} {}} {-tickinterval tickInterval
TickInterval 0 0.0} {-to to To 100 100.0} {-troughcolor troughColor Background
SystemScrollbar SystemScrollbar} {-variable variable Variable {} {}} {-width width
Width 15 15}
```

Each list pair follows this combination:

```
option-switch option-name option-class option-default-value option-actual-value.
```

You can interactively experiment and learn about the widget's options and their default values. There is, however, no easy way to learn a widget's methods without perusing its manual page.

Frames

Frames are the simplest of all the Tk widgets. They are used only as containers, as you can see in the following example:

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

. config -bg steelblue

foreach frame {sunken raised flat ridge groove} {
   frame .$frame -width 0.5i -height 0.5i -relief $frame  -bd 2
   pack .$frame -side left -padx 10 -pady 10
}
```

This script creates five frames with different 3D borders:

Frames are often invisible and are almost always used to create nested layouts.

How It Works

In the above example, the `-relief` option is used to set the border relief of the frame, and the `-bd 2` option sets the widget border width to two pixels. This option is supported by all the Tk widgets and gives the 3D effect.

As for the rest of the code, you can see the use of a Tcl list to create the five frames. The frames are sized by setting the `-height` and `-width` options to `0.5i` (half an inch); they're packed to the left and padded 10 pixels on each side by the `-padx` and `-pady` options.

Top-Level

Top-level widgets are like frames, except they have their own top-level windows whereas frames are internal windows within a top level.

```
% toplevel .t -width 1.5i -height 1i  -relief ridge -bd 4
```

will create a top-level window that looks like the following:

Labels

A label is simple widget that can display multiline text. We can create a label using the `label` command:

```
% label .l -wraplength 1i -justify right -text "Hello Tk World!"
```

This creates a multiline label widget with a text length of one inch for each line. Once you pack the label using

```
% pack .l
```

it will create a widget that looks like this:

When you've created the label, you can use the widget command to communicate with it. For example, the following command will query the foreground color of the label widget:

```
% .l cget -fg
Black
```

All the Tk widgets support the `cget` widget command, which retrieves any widget configuration option. We can also use the `configure` method of the Tk widgets to set configuration options interactively. For example,

```
% .l configure -fg yellow -bg blue
```

will set the label's foreground to yellow and background to blue.

Buttons

Tk provides three kinds of buttons: ordinary push buttons, check boxes and radio buttons.

Pressing a push button performs an action. We use check boxes to select or deselect a number of options. Radio buttons are similar, but they exclusively select one choice from a group of options. You're most likely familiar with the widgets, if not the terminology.

Let's look at the following example, which illustrates most of the uses of Tk buttons.

Try It Out—A Choice of Buttons

After the script header and a couple of global variables, we create a check button to control the selection of a favorite programming language.

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

set lang tcl
set state 1

checkbutton .lan -text "Language" -command {changeState} -relief flat \
                 -variable state -onvalue 1 -offvalue 0
```

Next, we create a radio button panel, with one button for each language:

```
radiobutton .c -text "C" -variable lang -value c -justify left
radiobutton .tcl -text "Tcl" -variable lang -value tcl -justify left
radiobutton .perl -text "Perl" -variable lang -value perl -justify left
```

We need two push buttons to control the output:

```
button .show -text "Show Value" -command showVars
button .exit -text "Exit" -command {exit}
```

Having configured the buttons, we need to arrange them on screen. It's time for a bit of geometry management.

```
grid .lan  -row 1 -column 0 -sticky "w"
grid .c    -row 0 -column 1 -sticky "w"
grid .tcl  -row 1 -column 1 -sticky "w"
grid .perl -row 2 -column 1 -sticky "w"
grid .show -row 3 -column 0 -sticky "w"
grid .exit -row 3 -column 1 -sticky "w"
```

The check button needs a callback procedure, `changeState`. This is registered by the check button's `-command` option.

```
proc changeState args {
    global state
    if {$state == "0"} {
        catch {
            .c config -state disabled
            .tcl config -state disabled
            .perl config -state disabled
        }
    } else {
        .c config -state normal
        .tcl config -state normal
        .perl config -state normal
    }
}
```

The push buttons need a similar procedure, `showVars`:

```
proc showVars args {
    global state lang
    if {$state == "0"} {
        puts "No Language is selected"
    } else {
        puts "The Language selected is $lang"
    }
}
```

When you run the program, you should see this:

How It Works

The program starts off by setting up two global variables, `lang` and `state`, to serve as the initial values of the check boxes and radio buttons.

A check box is declared to select/deselect the "language" option. Every time it's invoked, its command will call the `changeState` procedure. It also sets the global variable `state` to 1 or 0, depending on the selection before execution of the command.
Then the program composes the radio button, which is there to select just one of three languages (C, Tcl, and Perl). If you look at the code, these buttons share the same global variable, `lang`, which holds the value of the current selection. This makes sure that the user can select only one radio button at a time.

Finally, we declare two push buttons; one exits the application when the user presses it, and the other outputs the selection by calling the procedure `showVars`.

The command `changeState` is used by the check box to change the state of all three radio buttons between active and inactive, depending on whether it's selected or deselected. `showVars` is used by the `Show Value` push button to output the value of the current selection.

Buttons also support many other options, including flash invoke methods. For more information, look at the `button`, `checkbutton`, `radiobutton`, and `options man` pages. Labels and buttons also support bitmaps and images as their labels. We'll learn about images later.

> **The lines in the example that start with `grid...` are used for geometry management of the created widgets. We'll cover geometry management later in this chapter.**

Messages

Messages are similar to labels and are used to display multiline text. They differ from labels in that they automatically break up text to display it in a multiline format, using word boundaries and aspect ratio. Message widgets can justify the text displayed and they can also handle nonprintable characters.

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

message .m -aspect 400 -justify center \
           -text "This is a message widget with aspect ratio 400 and \
                  center justification. Message widgets can also \
                  display control characters \240 \241 \242 \243 \251 \
                  \256 \257 \258  and tabs \t  etc..."

pack .m
```

This example will create a simple message widget containing control characters.

Entrys

Entrys (*sic*) are single-line text widgets that we can use to type in and display a single line of text. Entrys also support many key bindings for text editing. For example, here's a small program, `login.tk`, to handle user logins, though it lacks code to verify the user's password.

Try It Out—Gaining Entry

First, we set up the look of the login window. We also define a global `loginName` variable:

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

set loginName "timB"

label .name -text "Login:"
entry .nameEntry -textvariable loginName
label .passwd -text "Password:"
entry .passwdEntry -textvariable passwd -show *
```

Then we select all the text from `.nameEntry`:

```
.nameEntry selection from 1
.nameEntry selection to end
```

Finally, we arrange the widgets on the screen—we'll explain it later!

```
grid .name  -row 0 -column 0 -sticky "w"
grid .passwd -row 1 -column 0 -sticky "w"
grid .nameEntry -row 0 -column 1 -columnspan 2 -sticky "W"
grid .passwdEntry -row 1  -column 1 -columnspan 2 -sticky "W"
```

If you run the program, you'll see this:

How It Works

After the first five lines, which create two label and two entry widgets, the next two lines show how to select the text inside the entry. The `selection` command is Tk's method for moving information between widgets. The last four lines arrange the created widgets on the screen in a grid.

> **X defines a standard mechanism for supplying and retrieving the selection, and the `selection` command is Tk's way of managing inter-client communication. It obeys X's Inter-Client Communication Conventions Manual (ICCCM) rules. The reason we introduce `selection` here is to show that you can programmatically set the selection so that another, non-Tk, X client can retrieve the selection using normal X Windows conventions.**

Entry widgets use key bindings for internal navigation through the text. If you read the `man` page, you'll find that the entry widget supports lots of EMACS bindings, along with all the Motif bindings dictated by the OSF Motif style guide. Here are a few of the more common ones:

Key Binding	Description
Ctrl+a	Moves the insertion cursor to the beginning of the entry text.
Ctrl+e	Moves the cursor to the end of the entry text.
Ctrl+/	Selects all the text in the entry.

List Boxes

A list box widget can display a collection of strings and allows the user to select one or more items. The following program shows a way to use a list box to design a Motif-like prompt dialog.

Try It Out—Lists

First we create the user interface elements:

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

scrollbar .h -orient horizontal -command ".list xview"
scrollbar .v  -command ".list yview"
listbox .list -selectmode single -width 20 -height 10 \
              -setgrid 1 -xscroll ".h set" -yscroll ".v set"
label .label -text "File Selected:" -justify left
entry .e -textvariable fileSelected
```

To give widgets that Motif-ish look and feel, we use the `grid` geometry manager:

```
grid .list -row 0 -column 0 -columnspan 2 -sticky "news"
grid .v -row 0 -column 2 -sticky "ns"
grid .h -row 1 -column 0 -columnspan 2 -sticky "we"
grid .label -row 2 -column 0
grid .e -row 3 -column 0  -columnspan 3 -sticky "we"

grid columnconfigure . 0 -weight 1
grid rowconfigure . 0 -weight 1
```

We initialize the list box with the contents of the current directory:

```
foreach file [glob *] {
    .list insert end $file
}
```

Finally, we `bind` an event handler to the list box to make it react to the release of mouse button 1. This corresponds to the left mouse button for right-handed users, and the right mouse button for left-handed users. We'll continue to call it mouse button 1 in this section, as this is the convention used in the code.

```
bind .list <ButtonRelease-1> \
    {global fileSelected;set fileSelected [%W get [%W curselection]]}
```

If you run the program, you'll see this:

How It Works

The program first creates two scrollbars and then attaches them to the list box it creates. The widgets are interconnected using `-command` for the scrollbars and `-xview` and `-yview` commands for the list box. This is the way we tell two widgets how to communicate and react to one another's geometry or state. We'll explore more interconnection later in the chapter.

Next, we initialize the list box with the contents of the current directory using the `foreach` loop. The Tcl command `glob` performs pattern matching to return these filenames.

The list box provides many more configuration methods such as `delete`, `get`, `index`, `insert`, and `scan` to manipulate the displayed contents.

Scrollbars

As we saw in the previous example, scrollbars are usually connected to other widgets so that the widget's viewing area can be expanded. In the list box example, the viewing area is controlled by the two scrollbars, `.h` and `.v`, like this:

```
scrollbar .h -orient horizontal -command ".list xview"
scrollbar .v  -command ".list yview"
```

`.h` controls the horizontal viewing area of the list box using the command `.list xview`, and similarly for the vertical scrollbar, `.v`. The list box is also informed about the interconnection using the command

```
listbox .list ... -xscroll ".h set" -yscroll ".v set"
```

This, then, is how we make two widgets communicate with each other by binding them together and informing each of them about the other's behavior. You can also use implicit interconnection, as we'll see in the next section.

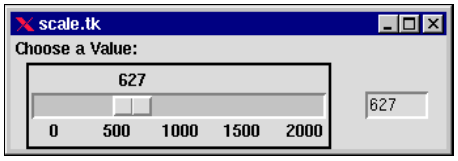
Scales

Scales display integer values and allow the user to select a particular value by moving a slider. Let's look at a simple example:

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

```
set foo 100
label .l -text "Choose a Value:" -justify left
scale .s -orient horizontal -from 0 -to 2000 -tickinterval 500 \
         -showvalue true -length 3i  -variable foo
entry .e -width 6 -justify left -textvariable foo

pack .l -side top -anchor nw
pack .s .e -side left -padx 4m -fill x
```

When you run this program, you'll see this screen:



Here, the scale and the entry have an implicit interconnection through the global variable `foo`. If you update the value of the scale, the value in the entry is automatically updated.

> **All the variables used explicitly in Tk's widget event handlers are of global scope. If the bound variable doesn't exist, Tk will automatically create one for you. So in the above example the variable `foo` is a global variable, and `scale` and `entry` share the same variable, creating an implicit interconnection in their behavior.**

Text

Tk's very versatile text widgets are used to create multiline, editable text. They support three types of annotations, **tags**, **marks,** and **embedded windows**, which affect what is displayed.

- **Tags** allow different portions of text to be displayed with different fonts, colors, and reliefs. Tcl commands can be associated with tags to make them react to user actions.
- **Marks** are used to keep track of various interesting positions in the text as it's edited.
- **Embedded window** annotations are used to insert widgets (windows) at particular points in the text. You can have any number of embedded windows in the text. All the embedded windows in the text require the text to be the parent of embedded windows.

Let's look at a demonstration of some of the text widget features.

> **Tk's text widget is so powerful that we can use it as an HTML widget without much effort. The well-known hypertext man page viewer `TkMan` uses Tk's text widget to display normal Linux `man` pages in hyper-linked form.**

Try It Out—Manipulating Text

1. First of all, we create a vertical scrollbar, which we attach to the text widget. Then we pack them side by side and tell the text window to expand to fill the available window space.

We make sure that the text window continues to fill the window even when resizing occurs by telling the packer, if there's extra vertical space, to expand both widgets to occupy that space. However, if some extra horizontal space is made available, only the text widget will be expanded.

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

scrollbar .y -command ".t yview"
text .t -wrap word -width 80 -spacing1 1m -spacing2 0.5m -spacing3 1m \
        -height 25 -yscrollcommand ".y set"

pack .t -side left -fill both -expand yes
pack .y -side left -fill y
```

2. Next, we want to create embedded windows. We don't have to worry about managing them because the text widget will look after them internally.

```
set image [image create photo -file mickey.gif -width 200 -height 200]
label .t.l -image $image
button .t.b -text "Hello World!" -command "puts Hi"
```

3. Then we configure all the tags that we're going to associate with the text window:

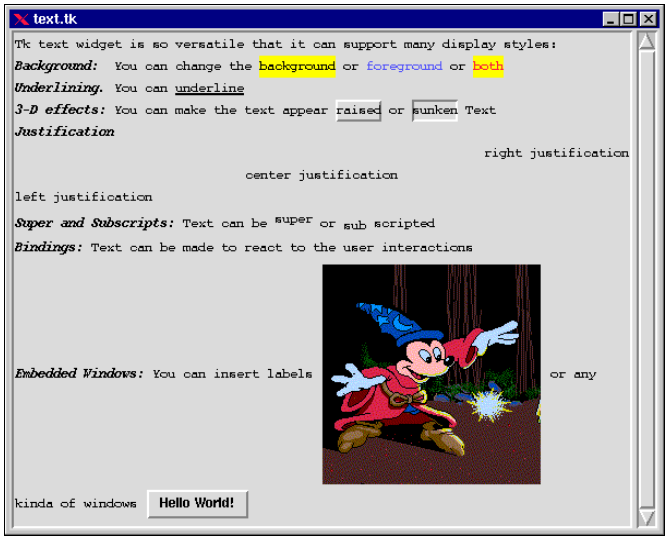
```
.t tag configure bold -font -*-Courier-Bold-O-Normal--*-120-*-*-*-*-*-*
.t tag configure yellowBg -background yellow
.t tag configure blueFg -foreground blue
.t tag configure yellowBgBlueFg -background yellow -foreground red
.t tag configure underline -underline 1
.t tag configure raised -relief raised -borderwidth 2
.t tag configure sunken -relief sunken -borderwidth 2
.t tag configure center -justify center
.t tag configure left  -justify left
.t tag configure right -justify right
.t tag configure super -offset 4p
.t tag configure sub -offset -2p
.t tag bind colorOnEnter <Any-Enter> ".t tag configure colorOnEnter \
                                        -background yellow"
.t tag bind colorOnEnter <Any-Leave> ".t tag configure colorOnEnter \
                                        -background {}"
```

4. Having configured the tags, we now insert text with those tags to show off the widget's potential, if not our graphic design.

```
.t insert end "Tk text widget is so versatile that it can support many \
               display styles:\n"
.t insert end "Background: " bold
.t insert end " You can change the "
.t insert end "background"  yellowBg
.t insert end " or "
.t insert end "foreground" blueFg
.t insert end " or "
.t insert end "both" yellowBgBlueFg
.t insert end "\nUnderlining. " bold
.t insert end "You can "
.t insert end "underline" underline
.t insert end "\n3-D effects: " bold
.t insert end "You can make the text appear "
.t insert end "raised" raised
.t insert end " or "
.t insert end "sunken" sunken
.t insert end " Text"
.t insert end "\nJustification" bold
```

```
.t insert end "\nright justification" right
.t insert end "\n center justification " center
.t insert end "\nleft justification " left
.t insert end "\nSuper and Subscripts: " bold
.t insert end "Text can be "
.t insert end "super" super
.t insert end " or "
.t insert end "sub" sub
.t insert end " scripted"
.t insert end "\nBindings: " bold
.t insert end "Text can be made to react to the user interactions" colorOnEnter
.t insert end "\nEmbedded Windows: " bold
.t insert end "You can insert labels "
.t window create end -window .t.l
.t insert end " or any kind of windows "
.t window create end -window .t.b
```

If you run this program, this is what you'll see:



How It Works

In this example, the first text window and a scrollbar are created and managed to create the basic interface. All the internal windows (`.t.l` and `.t.b`) are created, but not explicitly managed, because we're going to insert them into the text widget. Next, all the binding tags that we're going to use are configured using various configuration options. For example,

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

will create a tag called `bold`. The characters inserted with the tag will have the font `-*-Courier-Bold-O-Normal--*-120-*-*-*-*-*-*`. We'll look at the way Tk handles fonts later on. Similarly, characters inserted with `blueFg` will be displayed in blue.

We don't need to create the text tag explicitly. When we insert a piece of text with tag `foo`, `foo` will be created automatically. Here, we've configured the tags beforehand and the program has created them.

We then insert text using tags with the following format:

```
text_widget insert index chars taglist chars taglist...
```

An index is a string used to indicate a particular place within a section of text, such as a place to insert characters, or one endpoint of a range of characters with blue background. Indices have the syntax

```
base modifier modifier modifier...
```

`base` gives the starting point and the modifiers shift or adjust the index from the starting point. Modifiers can move the index in either direction from the starting point.

The `base` for the index must have one of the following forms:

Index Base	Description
`line.char`	Indicates the `char`th character of the line `line`.
`@x,y`	Indicates the character that covers the pixel within the text window whose co-ordinates are `x` and `y`.
`mark`	Indicates the character just after the mark.
`tag.first`	Indicates the first character in the text that has been tagged `tag`.; Ssimilarly for `tag.last`.
`pathname`	Indicates the position of the embedded window whose path name is `pathname.`

Modifiers can have these forms:

Modifier	Description
`+count chars`	Adjusts the index forward by `count chars`.
`-count chars`	Adjusts the index backward by `count chars`.
`+count lines`	Adjusts the index forward by `count lines`.
`-count lines`	Adjusts the index backward by `count lines`.
`Linestart`	Adjusts the index to refer to the first `char` on the line.
`Lineend`	Adjusts the index to refer to the last `char` on the line.
`Wordstart`	Adjusts the index to first `char` of the word containing the current index.
`Wordend`	Adjusts the index to last `char` of the word containing the current index.

We can associate a particular piece of text with more than one tag. For example, text can be bold and italic at the same time. When you insert text, you need to specify the location. In our last example, `end` means "insert after the last character displayed." In text, indices can also be tags and marks, so the text command

```
.t insert end "right justification" right
```

will insert the text `right justification` at the end of all the text in the text widget and will right-justify it.

> **Text supports lots of features, and we recommend that you read the text `man` page and take a look at the text demos that come with the Tk distribution. Before we finish covering text, though, think about what it would take to implement the last example in Motif or Xlib. In Motif, it would take a couple of hundred lines and in Xlib, perhaps several thousand. The power of Tk can be pretty mind boggling!**

Canvases

Tk's canvas widget is used to implement structured graphics. Canvases can display any number of items, including rectangles, circles, lines, text, and embedded windows, which can be manipulated (moved or colored) and bound to user interactions. For example, we can make a particular item change its background color when the user clicks the mouse button over it.

Before we play with the canvas widget, we need to cover some concepts: **identifiers** and **tags**.

When we create each item in the canvas, it's assigned a unique integer identifier. Items can have any number of tags associated with them. A tag is a string of characters that can take any form except an integer. Tags are used for item grouping, identifying, and manipulating purposes. The same tag can be associated with many items to group them under one category. Every item inside the canvas can be identified by its ID, or a tag associated with it. The tag `all` is implicitly associated with every item in the canvas. The tag `current` is automatically managed by Tk. It refers to the topmost item whose drawn area lies at the position of the mouse cursor.

When we specify items in canvas widget commands, if the specifier is an integer, we assume that it refers to a single item with that ID. If the specifier isn't an integer, we assume that it refers to all the items in the canvas that have the tag matching the specifier. In the next example, we use the `tagOrId` symbol to specify either an ID that selects a single item or a tag that selects zero or more items.

When we create any item on the canvas, we specify its location. The locations are floating-point numbers optionally suffixed with one of the letters `m`, `c`, `i`, and `p`:

- `m` stands for millimeters.
- `c` stands for centimeters.
- `i` stands for inches.
- `p` stands for points.

If we don't follow the coordinate with one of these letters, the program assumes that the item location is in pixels. Let's look at some of the canvas commands and see what they do. In the following commands, the `pathName` identifier refers to the canvas path name:

```
pathName create arc x1 y1 x2 y2  ?option value option value ...?
```

This creates an `arc` item on the canvas, with `x1 y1 x2 y2` specifying the coordinates of a rectangular region enclosing the oval that defines the arc. Command options in this example include `-extent`, `-fill`, and `-outline`. For example, the command

```
% set k [.c create arc 10 10 50 50 -fill red -outline blue -tags redArc]
```

creates an `arc` item inside canvas `.c` with `$k` giving the value of the ID. Its outline is drawn in blue. This arc is enclosed inside a rectangle with canvas coordinates `10 10 50 50` and is filled in with red. A tag, `redArc`, is also associated with the `arc` item.

```
pathName itemconfigure tagOrId ?option value option value ...?
```

This command is similar to the `-configure` widget command, except that we can use it to modify only specific options for the item denoted by `tagOrId`, instead of modifying the whole canvas widget. For example,

```
% .c itemconfigure redArc -fill yellow
```

will change all the `fill` colors of items associated with tag `redArc` to yellow.

```
pathName type tagOrId
```

will return the type of first item in the list of items referred to by `tagOrId`. For example,

```
% .c type redArc
arc
```

```
pathName bind tagOrId ?sequence? ?command?
```

works just like the `bind` command, but instead of applying the sequence to the whole canvas, it applies it just to the item specified by `tagOrId`. If `command` isn't given, it returns all the commands associated with the binding sequence `sequence` for canvas item `tagOrId`. If neither `sequence` nor `command` is specified, all the sequences bound to the item are returned.

```
% .c bind $k <Enter> ".c itemconfigure redArc -fill blue"
% .c bind redArc  <Leave> ".c itemconfigure redArc -fill red"
```

In this example, the first binding will fill in the item associated with `tagOrId` `$k` with blue when the mouse enters the item. The second binding fills in all the items associated with tag `redArc` with red when the mouse leaves them.

Text and canvases support so many commands that it would take much more than one chapter to explain them all. We strongly advise that you refer to the `canvas` and `text` `man` pages for the mastery of these two widgets. Here's a small example that shows a few of their features.

Try It Out—Text on Canvas

First we create the canvas and then some objects to display on it: an image of a teapot, a line of text over the image, another text object to exhort users to move the items around, and a rectangle. We pack the canvas so it will fill the window.

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

set c [canvas .c -width 300 -height 300 -relief sunken -bd 2]

set image [image create photo -file teapot.ppm -width 200 -height 200]
$c create image 150 150 -anchor center -image $image -tags item

$c create text 150 150 -text " Image Object"  -fill white

$c create text 10 10 -text " Move any Item \n using Mouse " -justify center \
                      -anchor nw -tags item  -fill red
$c create rectangle 200 10  250 40 -fill yellow -outline blue -tags item

pack .c
```

Next, we bind the canvas so that we can operate on the items shown on it. We'll define the `itemDragStart` and `dragItem` procedures:

```
bind $c <1> "itemDragStart $c %x %y"
bind $c <B1-Motion> "dragItem $c %x %y"
```

For the procedure's benefit, we need to define two global variables, `lastX` and `lastY`.

```
global lastX lastY

# event handler for the <1> event
proc itemDragStart {c x y} {
    global lastX lastY
    set lastX [$c canvasx $x]
    set lastY [$c canvasy $y]
}
# event handler for the <B1-Motion> event
proc dragItem {c x y} {
    global lastX lastY
    set x [$c canvasx $x]
    set y [$c canvasy $y]
    $c move current [expr $x-$lastX] [expr $y-$lastY]
    set lastX $x
    set lastY $y
}
```

The program produces this output:

How It Works

It's really a very simple example. We created a few canvas item types and bound the mouse buttons so that the user can move them with the mouse. For example, the line

```
$c create image 150 150 -anchor center -image $image -tags item
```

creates an image on the canvas at the canvas location (150, 150). This image is an object in its own right, so you can move it and make it react to the user by binding event handlers to the tag associated with the item. Like text, canvas supports many features, so it's very difficult to explain them comprehensively in a simple 20-line example. We'll look at some more canvas features in the final applications.
Before we leave canvas objects, we'll mention a few of their properties:

- Canvas items can have event handlers attached to them.
- An item can have many tags associated with it, but will have one unique ID.
- If an item is a widget, it should be the child of the canvas that contains it.
- If items are widgets, you can configure them just as you would had they been outside the canvas. Embedding them within the canvas doesn't change their methods.
- As you place items on the canvas, you can stack them on top of one another, obscuring some of the items beneath. You can change the stacking order using canvas `raise` and `lower` commands.

Finally, to make things simpler in the last example, we don't bind the procedures to the objects through tags, but directly through the canvas. If you look at the `dragItem` procedure, the line

```
set x [$c canvasx $x]
```

sets the values of `x` to a `canvasx` coordinate from the real screen coordinate `x`. The line

```
$c move current [expr $x-$lastX] [expr $y-$lastY]
```

moves the current object under the mouse cursor (denoted by the index `current`) to a new location, from `lastX` to `x`. `lastX` was saved when the user event handler `itemDragStart` was invoked, through the binding

```
bind $c <1> "itemDragStart $c %x %y"
```

Here, the bind means that when the user clicks on the canvas with mouse button 1, the `itemDragStart` event handler is invoked with arguments `canvas`, `%x` (the value of `x` at the mouse click), and `%y` (the value of `y` at the mouse click). We'll discuss bindings later in the chapter.

Images

Tk can display images of two built-in types: photo and bitmap. The photo type can display `gif` and `ppm` / `pgm` files, while the bitmap format can display `xbm` files. The `image` command can be used to create images. The general format of the `image` command is

```
image option ?arg arg ...?
```

where `option` can be used to create, delete, and set such options as height, names, image type, and so on. Next we're going to develop an example based on the sliding block example that comes with the Tk distribution, but we are going to jazz up the original example by using the `image` command.

Try It Out—Manipulating Images

First we create the image. Then we configure the frame that will hold the pieces of the image that form the puzzle. This we pack with a little padding:

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

set image [image create photo -file mickey.gif -width 160 -height 160]

frame .frame -width 120 -height 120 -borderwidth 2 -relief sunken \
-bg grey
pack .frame -side top -pady 1c -padx 1c
```

Now we create the individual pieces of the puzzle. This involves 15 loops of the code, which crops portions of the original image to fit on the buttons.

```
set order {3 1 6 2 5 7 15 13 4 11 8 9 14 10 12}
for {set i 0} {$i < 15} {set i [expr $i+1]} {
    set num [lindex $order $i]
    set xpos($num) [expr ($i%4)*.25]
    set ypos($num) [expr ($i/4)*.25]

    set x [expr $i%4]
    set y [expr $i/4]

    set butImage [image create photo image-${num} -width 40 -height 40]
    $butImage copy  $image  -from [expr round($x*40)] \
                                  [expr round($y*40)] \
                                  [expr round($x*40+40)] \
                                  [expr round($y*40+40)]
    button .frame.$num -relief raised -image $butImage \
                       -command "puzzleSwitch  $num" \
                       -highlightthickness 0
    place .frame.$num -relx $xpos($num) -rely $ypos($num) \
                      -relwidth .25 -relheight .25
}
```

Finally, we have the event handler that deals with the user's input. The two global variables are set to show that the initial space in the puzzle is at the bottom right-hand corner.

```
set xpos(space) .75
set ypos(space) .75

proc puzzleSwitch { num} {
    global xpos ypos
    if {(($ypos($num) >= ($ypos(space) - .01))
           && ($ypos($num) <= ($ypos(space) + .01))
           && ($xpos($num) >= ($xpos(space) - .26))
           && ($xpos($num) <= ($xpos(space) + .26)))
           || (($xpos($num) >= ($xpos(space) - .01))
           && ($xpos($num) <= ($xpos(space) + .01))
           && ($ypos($num) >= ($ypos(space) - .26))
           && ($ypos($num) <= ($ypos(space) + .26)))} {
        set tmp $xpos(space)
        set xpos(space) $xpos($num)
        set xpos($num) $tmp
        set tmp $ypos(space)
        set ypos(space) $ypos($num)
        set ypos($num) $tmp
        place .frame.$num -relx $xpos($num) -rely $ypos($num)
```

```
    }
}
```

When you run the program, you'll get this 15-piece image puzzle output:

How It Works

The first line in the program creates a photo image, using the `mickey.gif` file and assigns it to the variable `image`. Portions of this image are then copied on to the buttons with the `set butImage` line and those following in a `for` loop. The result is 15 buttons with 15 associated images, taken from the big image held in `$image`. The rest of the program deals with event handlers to arrange the buttons when the user clicks them. We'll return to that part when we deal with geometry management.

The gist of the `puzzleSwitch` algorithm logic is based on the fact that when the user clicks a button, if the button is next to an empty space, the button and empty space will be swapped. If you play with the 15-piece puzzle, you will notice that the piece that can take the place of the empty space will obey one of the following rules:

- It will have the same x position as the empty space, and its y position will be 0.25 units away (up or down) from the empty space. (The piece is on the same column as the empty space, and it is directly above or below the empty space.)
- It will have the same y position as the empty space, and its x position will be 0.25 units away (left or right) from the empty space. (The piece is on the same row as the empty space, and it is directly to the left or right of the emtpy space.)

The preceding algorithm makes use of these properties to decide whether to switch the piece with the empty space.

Buttons and labels support images as labels. Also, these labels can be embedded inside canvas and text widgets. Refer to the `bitmap`, `photo,` and `image man` pages for more information on Tk's `image` support.

Menu

Traditionally, menus are used to provide users with a set of choices in an application without changing much of the application's appearance. Menus give users convenient access to various features of the application without the user having to move away from the main window. Tk's `menu` command creates a widget that displays a list of entries in a separate top-level window. Menu is not a container widget; it is a single widget with different objects embedded in it.

Menus can have three types of entries embedded in them:

- Command entries, to run commands
- Radio entries, to select one of many choices
- Option entries, to select one or more choices from a group of options

Menus can also hold other menus in a recursive way, by using cascade entries.

Menu entries can be displayed with up to three separate fields: a label (in the form of text), a bitmap, or an image, using the `-label`, `-bitmap`, or `-image` option, respectively. A second field may use the `-accelerator` option to specify an accelerator sequence next to the label. The `-accelerator` option describes a key sequence that is used to invoke a particular entry associated with a menu entry. A third option is an indicator that radio and option entries display to the left of the label. Note that Tk does not automatically create a key binding when the `-accelerator` option is specified. The binding should be explicitly set using the `bind` command for the sequence to take effect; setting `-accelerator` simply displays the key combination in the menu.

Menu entries can be configured with different options, such as foreground and background colors and fonts, using the `entryconfigure` option of the `widget` command. Entries can also be disabled using the `-state` option. If a menu entry is disabled, it will not respond to the user action.

Tk menus are very flexible. You can enable the `-tearoff` option of the menu so that the user can tear the menu off from the menu bar and use it as a top-level window. You can also specify commands that get called when a menu is posted or torn off.

Menus are indexed using either their position numerically in the menu, their label , or "last" and "end" tags. Menus can be posted programmatically by calling the `post` and `unpost` menu commands. Tk's documentation refers to menus as being posted; to say "pulled down" or "popped up" only describes the behavior of certain menus on certain platforms. Posting is a more generic term.

Each top-level widget in Tk can have one menu widget act as the default menu bar for that window. A menu bar is a list of menus arranged side by side in a frame. Menu bars can be attached using the `-menu` widget option associated with top-level windows.

The Tk library provides a `<<MenuSelect>>` virtual event that is triggered whenever a menu or one of its entries becomes active. The `menu` command provides lots of options; for a complete list of options, refer to the `menu` command manual page.

The menu system was overhauled in Tk version 4.0, and many improvements were made in Tk version 8.0. Prior to Tk 8.0, to create menu bars users had to use functions such as

```
tk_menuBar frame ?menu menu ...

tk_bindForTraversal arg arg ..
```

These functions are deprecated and have no effect in Tk versions beyond 4.0.

Try It Out—Menus

Let's look at an example that illustrates most of the `menu` command features. This example will make use of the text widget features. Using the menus, we will change the background color and the properties of the font used to display the text. We will also create a menu to insert bitmap images inside the text widget.

First, we will create the main window components, including a text widget with an associated scrollbar, and a status widget to display menu traversal and error messages. We arrange these widgets on the screen using the `grid` command. We will also create a new font named `myFont`. We will use the menu to manipulate the font attributes so that the text in the text widget will change its appearance.

```
wm title . "Menu demonstration"
wm iconname . "Menu demo"

# create the basic UI
scrollbar .yscroll -orient vertical -command ".text yview"
font create myfont -family Courier -size 10 -weight bold -slant italic \
    -underline 1
text .text -height 10 -width 40 -bg white -yscrollcommand ".yscroll set" -font myfont
label .msg -relief sunken -bd 2 -textvariable message -anchor w -font "Helvetica 10"
.text insert end "Menu Demonstration!"

# manage the widgets using the grid geometry manager.
grid .text -row 0 -column 0  -sticky "news"
grid .yscroll -row 0 -column 1 -sticky "ns"
grid .msg -row 1 -columnspan 2 -sticky "ew"

grid columnconfigure . 0 -weight 1
grid rowconfigure . 0 -weight 1
```

Next, we will develop callback functions, which will be associated with the menu entries. The `SetBg` procedure will change the background color of the text. `ConfigureFont` will change the attributes of `myFont`. `InsertImage` will insert the named bitmap into the text buffer. The `InsertImage` procedure has a side effect: If the named bitmap already exists in the text buffer, it will be deleted and a new bitmap will be inserted. The `OpenFile` procedure will prompt the user for a file, and if the user selects a file, its contents will be displayed in the text widget.

```
# procedure to set text background color
proc SetBg {} {
    global background
    .text configure -bg $background
}

# procedure to configure the previously created font.

proc ConfigureFont {} {
    global bold italic underline
    expr {$bold ? [set weight bold]: [set weight normal]}
    expr {$italic? [set slant italic]: [set slant roman]}
    expr {$underline? [set underline 1]: [set underline 0]}
    font configure myfont -weight $weight -slant $slant -underline $underline
}

# Procedure to insert images in the text widget

proc InsertImage {image} {
    catch {destroy .text.$image}
```

```
    label .text.$image -bitmap $image
    .text window create end -window .text.$image
}

# Callback for open menubutton

proc OpenFile {} {
    global message
    set file [tk_getOpenFile]
    if {$file == ""}  {
        set message "No file selected..."
        return;
    }
    .text delete 0.0 end
    set fd [open $file "r"]
    while {[eof $fd] != 1} {
        gets $fd line
        .text insert end $line
        puts $line
        update idletasks
    }
    close $fd
}
```

Now we focus on the menu widget and its components. First, we create a menu that will become the menu bar for the top-level window.

```
# create toplevel menu

menu .menu -tearoff 0  -type menubar

# Create File menu

set m .menu.file
```

We will add a `File` submenu with `open` and `exit` entries. The open entry will prompt the user with an "open file" dialog. If the user chooses a file, that file will be displayed in the text widget using the `OpenFile` procedure. The `exit` menu entry is used to exit the application. As you can see, Tk does not create a default global binding for the menu entry just by using the `-accelerator` menu entry option. We have to explicitly create the binding in order for the accelerator to take effect.

```
menu $m -tearoff 0
.menu add cascade -label "File" -menu $m -underline 0
set modifier Meta
$m add command -label "Open..." -accelerator $modifier+o -command "OpenFile" -
underline 0 -command OpenFile
bind . <$modifier-o> "OpenFile"
$m add separator
$m add command -label "Exit..." -accelerator $modifier+x -command "exit" -underline 0
bind . <$modifier-x> "exit"
```

We next add an Options submenu to the main menu. This submenu contains Background and Font cascade menus. The Background cascade menu contains a group of radio buttons to change the background color of the text widget. The Font cascade menu provides a group of check buttons to manipulate `myfont` menu attributes.

```
#
# Create options menu
#
set m .menu.options
menu $m -tearoff 1
```

```
.menu add cascade -label "Options" -menu $m -underline 0
$m add cascade -label "Background" -menu .menu.options.bg -underline 0
$m add cascade -label "Font" -menu  .menu.options.font -underline 0

#
# create Radio button cascade menu
#

set m .menu.options.bg
menu $m -tearoff 0
$m add radio  -label "Red" -background red -variable background -value red \
    -command SetBg
$m add radio  -label "Yellow" -background yellow -variable background \
    -value yellow -command SetBg
$m add radio  -label "Blue" -background blue -variable background -value blue \
    -command SetBg
$m add radio  -label "White" -background white -variable background -value white \
    -command SetBg
$m invoke 3

#
# Insert option button cascade Menu
#

set m .menu.options.font
menu $m -tearoff 0
$m add check -label "Bold" -variable bold -command ConfigureFont
$m add check -label "Italic" -variable italic -command ConfigureFont
$m add check -label "Underline" -variable underline -command ConfigureFont
$m invoke 3
```

As you can see from the code, the entries in a menu can be configured to have different backgrounds and foregrounds, as well as other standard widget options.

Next, we will proceed to add yet another cascade entry to the main menu to insert bitmaps into the text widget. As explained earlier, these bitmap entry commands have a limitation: Only one instance of these bitmaps can be present in the text widget at any given time.

```
#
# Create insert menu option
#
set m .menu.insert
menu $m -tearoff 0
.menu add cascade -label "Insert" -menu $m -underline 0
foreach i {info questhead error} {
    $m add command -bitmap $i -command "puts {You invoked the $i bitmap}"\
         -hidemargin 1 -command "InsertImage $i"
}
$m entryconfigure 2 -columnbreak 1
```

One thing to observe from the above code snippet is that entries in a menu can be arranged in a tabular fashion using the `entryconfigure` command with the `–columnbreak` option.

Finally, we will attach the menu to the top-level widget to make it the default menu bar. We also make use of the `<<MenuSelect>>` virtual event, which is activated when any menu or one of its entries is selected. The `<<MenuSelect>>` virtual event will display a message in the message label, indicating that a particular entry has been selected:

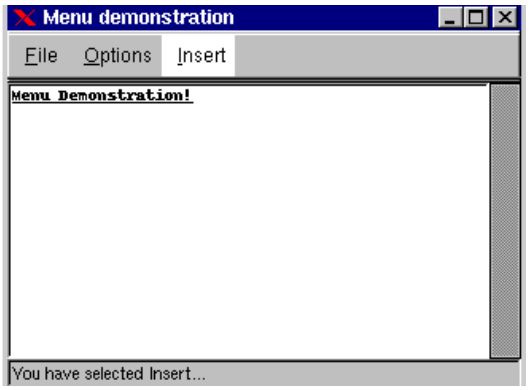
```
#
# Attach the menu to the toplevel menu
#
. configure -menu .menu


#
```

```
#  Bind  global tags
#

bind Menu <<MenuSelect>> {
    global message
    if {[catch {%W entrycget active -label} label]} {
        set label "    "
    }
    set message "You have selected $label..."
}
```

When you run the preceding example using the command

```
$ wish menu.tcl
```

you will see the following screen:



Menu Button

Menu buttons are like normal buttons with menus associated with them. They are very useful in graphical user interfaces to provide a set of choices grouped together by one button. Unlike menu bars, which usually have more than one cascade menu associated with them, a menu button has only one associated menu. There is only one menu bar associated with a top-level window, but menu buttons can be embedded anywhere in the user interface.

Menu buttons can also be organized into groups to make menu bars, which can be placed anywhere inside a user interface (unlike top-level menus). These can be used to build toolbars and similar functionality into your user interfaces. Usually, a menu button can also be configured to behave like an option menu by setting the `-indicator` option. The menu button command syntax is as follows:

```
menubutton pathName ?options?
```

Try It Out—Menu Buttons

To illustrate MenuButtons' features, we will create a simple application that draws circles and rectangles on a canvas with a specified fill color. First, we will develop a utility procedure that will draw a circle or a rectangle on the canvas. It uses three global variables: `x`, `y`, and `sqsize`. x-y coordinates are used to specify where on the canvas the object needs to be drawn. The variable `sqsize` is used either as the diameter or the side of the square.

Let's start off with the basics needed to draw circles and squares.

```
wm title . "Menubutton demonstration"
wm iconname . "Menubutton demo"
#
# Initial parameters to draw circles and squares
#
set x 50
set y 50
set sqsize 30
#
# procedure to draw canvas objects
#
proc AddObject {type} {
    global x y sqsize sqsize  fillc
    if {$type == "circle"} {
        .c create oval $x $y [expr $x+$sqsize] [expr $y+$sqsize] \
            -tags item -fill $fillc
    } elseif {$type == "square"} {
        .c create rectangle $x $y [expr $x+$sqsize] [expr $y+$sqsize] \
            -tags item -fill $fillc
    }
    incr x 10
    incr y 10
}
```

Next, we create a canvas with a frame, a menu button, and a dismiss button. The frame widget will hold the menu and dismiss buttons. We pack all the elements to create the main user interface.

```
# create the basic User Interface canvas, 2 menu buttons and a dismiss button
set c [canvas .c -width 200 -height 200 -bd 2 -relief ridge]

frame .f -bd 2

menubutton .f.m1 -menu .f.m1.menu -text "Draw" -relief raised -underline 0 \
    -direction left
button .f.exit -text "Dismiss" -command "exit"

# manage the widgets using the grid geometry manager.
pack .c -side top -fill both -expand yes
pack .f -side top -fill x -expand yes

pack .f.m1 .f.exit -side left  -expand 1
```

Finally, we add a menu to the menu button. We create a menu and add three entries to it: two command entries to draw circle and square objects, and a cascade widget for fill color selection.

```
set m .f.m1.menu
menu $m -tearoff 0

$m add command -label "Circle" -command "AddObject circle" -accelerator "Meta-c"
bind . <Meta-c> "AddObject circle"
$m add command -label "Square" -command "AddObject square" -accelerator "Meta-s"
bind . <Meta-s> "AddObject square"
$m add separator
$m add cascade -label "Fill Color.." -menu .f.m1.cascade

set m  .f.m1.cascade
menu $m -tearoff 0
$m add radio  -label "Red" -background red -variable background -value red \
    -command "set fillc red"
$m add radio  -label "Yellow" -background yellow -variable background \
    -value yellow -command "set fillc yellow"
$m add radio  -label "Blue" -background blue -variable background -value blue \
```

```
    -command "set fillc blue"
$m add radio  -label "White" -background white -variable background -value white \
    -command "set fillc white"
$m invoke 1
```

As you can see from the code, we have not only added the accelerators to the menu entries; we also have created the bindings explicitly using the `bind` command.

Pop-up Menu

Tk also supports pop-up menus. Unlike menu buttons and menu bars, which provide static menus, pop-up menus are used to provide a context-sensitive menu system. For example, if you are designing a text editor, when the user selects a block of text and clicks the right mouse button, you can programmatically create a menu with items such as "Spell...," ""Format...," "Copy," and "Delete." Pop-up menus help facilitate such tasks. Pop-up menus are very helpful in associating menus with any type of widgets, such as text and canvas widgets. Pop-up menus don't have any menu buttons associated with them. They are plain menus that get posted programmatically by associating a binding to a specific widget and invoking the `tk_popup` command in the event handler of that binding. Since the menus are posted dynamically, the entries inside a pop-up menu can be created dynamically to display only relevant items.
The general creation format of an pop-up menu is

```
tk_popup menu x y ?entry?
```

where `menu` is the menu that needs to be posted, `x` and `y` specify the coordinates, and `entry` gives the index of an entry in `menu`. The menu will be located so that the entry is positioned over the given point. Let's build some pop-up menus.

Try It Out—Pop-up Menus

First, create a menu and add a binding to the top-level window associated with the menu so that the menu will get posted when the user clicks on the window using the third mouse button.

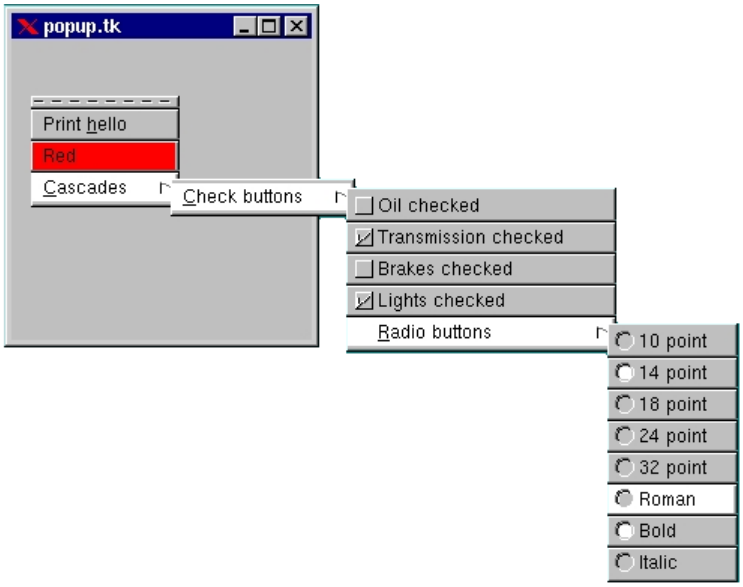
```
set w .menu
catch {destroy $w}
menu $w
bind . <Button-3> {
    tk_popup .menu %X %Y
}
```

The rest of the example creates menu entries to show that pop-up menus have exactly the same capabilities as regular menus.

```
# Add menu entries


$w add command -label "Print hello" \
        -command {puts stdout "Hello"} -underline 6

$w add command -label "Red" -background red

# Add a Cascade menu

set m $w.cascade

$w add cascade -label "Cascades" -menu $m -underline 0
menu $m -tearoff 0
```

```
$m add cascade -label "Check buttons" \
        -menu $w.cascade.check -underline 0
set m $w.cascade.check
menu $m -tearoff 0
$m add check -label "Oil checked" -variable oil
$m add check -label "Transmission checked" -variable trans
$m add check -label "Brakes checked" -variable brakes
$m add check -label "Lights checked" -variable lights
$m add separator
$m invoke 1
$m invoke 3

$m add cascade -label "Radio buttons" \
        -menu $w.cascade.radio -underline 0
set m $w.cascade.radio
menu $m -tearoff 0
$m add radio -label "10 point" -variable pointSize -value 10
$m add radio -label "14 point" -variable pointSize -value 14
$m add radio -label "18 point" -variable pointSize -value 18
$m add radio -label "24 point" -variable pointSize -value 24
$m add radio -label "32 point" -variable pointSize -value 32
$m add sep
$m add radio -label "Roman" -variable style -value roman
$m add radio -label "Bold" -variable style -value bold
$m add radio -label "Italic" -variable style -value italic
$m invoke 1
$m invoke 7
```

When you run the preceding example using the command `wish popup.tk`, you will see the following:



Option Menu

Tk's option menu is written completely in Tk to emulate the Motif option button and has the following syntax:

`tk_optionMenu w varName value ?value value...?`

The `tk_optionMenu` command creates an options menu button `w` and associates a menu with it. Together, the menu button and the menu allow the user to select one of the values given by the `value` arguments. The current value will be stored in the global variable `varName`, which users can use to manipulate the options button. Calling `tk_optionMenu` returns the menu associated with the options button.

Let's use options buttons to reimplement our earlier buttons example.

Try It Out—Menu Options

We set the global variable `state` equal to 1, and create a check button and an options menu. These are packed side by side:

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

set state 1

checkbutton .lan -text "Language" -command {changeState} -relief flat \
                 -variable state  -onvalue 1 -offvalue 0
set optMenu [tk_optionMenu .opt lang Tcl C Lisp C++]

pack .lan .opt -side left

# now make C As the default using lang variable
set lang C++
```

We need a procedure to handle the application's event:

```
proc changeState {} {
    global state
    if $state {
        .opt config -state normal
    } else {
        .opt config -state disabled
    }
}
```

In this example, we've used an options menu instead of three radio buttons. This example works the same way as `buttons.tk`, but is much terser. It also reduces the amount of space needed to display the options on the screen. When you run this example, it will look like this:***[Auth: Note that art is missing from this file.]***

If you wanted to, as before, you could add two push buttons to output the value of the selection. This example also shows that we can control option menus using the global variable with which they are associated. The line

```
set lang C++
```

sets the selection to C++ by setting the variable.

Dialogs

Dialogs are used extensively in a user interface design cycle. Tk provides a custom dialog called `tk_dialog`. It's very simple but can be used in many clever ways to implement most tasks. `tk_dialog` has the syntax

```
tk_dialog window title text bitmap default string string ..
```

This will create a modal dialog with title `title`, with message text `text`, and with the specified bitmap inside. It will also create buttons with titles given by the `string` arguments. When the user presses one of the buttons, `tk_dialog` will return that button number and then destroy itself.
Let's look at a simple example:

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

wm withdraw .

set i [tk_dialog .info  "Info" "Simple Info Dialog" info  0 Ok Cancel]
if {$i==0} {
    puts "Ok Button Pressed"
} else {
    puts "Cancel Button Pressed"
}
exit
```

The first line unmaps the default top-level window, created by `wish`. This is necessary because `tk_dialog` creates a top-level window itself, and we don't want to have two windows popping up in this simple example. The next line creates a modal dialog `.info` with title "Info" and message "Simple Info Dialog" and adds in a built in `info` bitmap. It also creates two buttons, "Ok" and "Cancel," and makes button number 0 (Ok) the default.

When we were explaining the dialog, we used the word *modal*. What's that? It means that the user's range of choices is restricted. The user won't be able to do anything with the application unless they first respond to the dialog by clicking either Ok or Cancel. Once the user has clicked on one of these buttons, control passes back to the application, where `i` is set depending on the button the user invoked.

We can achieve modal interactions using the `grab` and `tkwait` commands. Take a look at the `man` pages or Tk books for more information on these topics.

Tk's Built-in Dialogs

In addition to providing the tk_dialog command, Tk provides many utility dialog procedures. Most GUI-based applications have lots of common functionality, such as prompting users for input or output files, color choices, and the like. Tk provides utility dialog boxes for these operations. Also, these built-in dialogs are written in such a way that they have the native look and feel of whichever operating system the Tcl script is run on. We will explore the utility dialogs in this section. These utility dialogs are not Tk's built-in commands, but rather utility scripts that provide a specific dialog functionality.

Color Chooser

`tk_chooseColor ?option value ...?`

Most GUI-based applications provide their users with ways to customize the look feel of the application using color and font choices. Humans are accustomed to using descriptive names for colors, but underneath, most graphic systems deal with color using different schemes such as RGB or CMYK. RGB is a Red, Green, Blue color scheme. Any color in the system can be represented using a combination of these three colors.

Tk describes colors in a similar way to HTML. If you've programmed Web pages before, you may be familiar with the notation. There are also a large number of valid color names that directly refer to specific color values. However, if you want to control Tk's colors more precisely, you'll have to become familiar with the way it represents them internally.
All colors in Tk are represented as hexadecimal integers. You can use various lengths of hex number, either 3, 6, 9, or 12 digits. For example, you could use 6-digit hex numbers between #000000 and #ffffff. Each pair of digits represents the level of red, green, or blue, in that order. So #ff9900, for example, would represent a red value of #ff, a green value of #99, and a blue value of #00. In fact, the resulting color would be a shade of orange. In this system, #000000 is black and #ffffff is white. Values such as #a5a5a5, where all three components are the same, will lead to shades of gray. Any other value will be a color of some sort.

The other hex number lengths acceptable to Tk are, of course, divided into three equal-length hex numbers to form the three components in exactly the same way as for the six-digit example.

`tk_chooseColor` dialog provides a simple way to choose a color using the RGB color scheme. This dialog also provides a way to inquire about the RGB values for a given color by name. The procedure `tk_chooseColor` pops up a dialog box for the user to select a color. The following option-value pairs are possible as command line arguments:

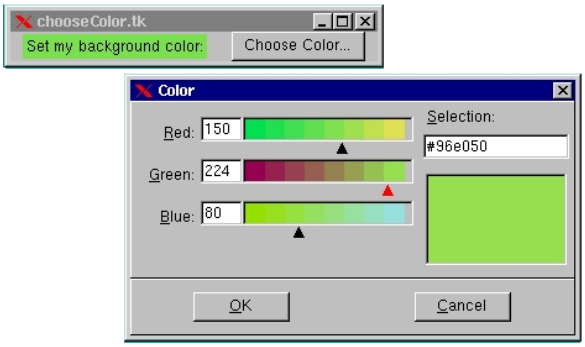
`-initialcolor color`	Specifies the color to display in the color dialog when it pops up. `color` must be in a form acceptable to the `Tk_GetColor` function, for example, `red` or `#ff0000`. (`#ff0000` is the RGB equivalent of red.)
`-parent window`	Makes `window` the logical parent of the color dialog. The color dialog is displayed on top of its parent window.
`-title titleString`	Specifies a string to display as the title of the dialog box. If this option is not specified, a default title will be displayed.

If the user selects a color, `tk_chooseColor` will return the name of the color in a form acceptable to Tk widget commands. If the user cancels the operation, both commands will return the empty string.

The following scripts illustrate the use of `tk_chooseColor` dialog box.

```
#
# tk_chooseColor demo
#

label .l -text "Set my background color:"
button .b -text "Choose Color..." -command ".l config -bg \[tk_chooseColor\]"
pack .l .b -side left -padx 10
```

When you run the above script and click the "Choose Color…" button, you will see the following:



Get Open/Save Files

`tk_getOpenFile` and `tk_getSaveFile` are convenience functions to prompt the user for input or output file selection, respectively. In most GUI-based operating systems, all applications provide some dialog boxes for selecting input and output files. Tk makes this functionality available to all Tk-based applications by providing these convenience functions. These dialog boxes have native look, feel, and behavior. The dialog boxes handle most error conditions so that the programmer does not have to do much other than create and initialize these dialogs. They provide interfaces so that developers can specify filters to select only those files matching certain patterns. The `Tk_getOpenFile` command is usually associated with the "Open" command in the File menu, and `tk_getSaveFile` is usually associated with the "Save as…" command.

If the user enters a file that already exists, the dialog box prompts the user for confirmation as to whether or not the existing file should be overwritten. The syntax of these commands is as follows:

```
tk_getOpenFile ?option value ...?
tk_getSaveFile ?option value ...?
```

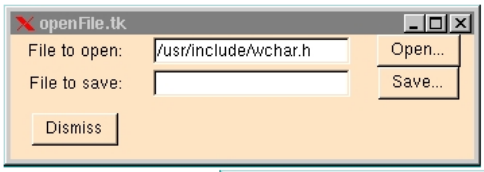
For a complete list of options for these commands, refer to the `tk_getOpenFile` manual page.

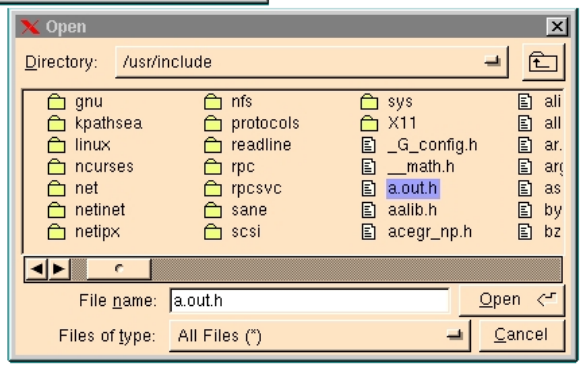
The following example illustrates the use of these commands.

```
# tk_getOpenFile demo
# tk_getSaveFile demo

label .o -text "File to open:"
entry .oe -textvariable open
set types {
    {{Text Files}       {.txt}   }
    {{TCL Scripts}      {.tcl}   }
    {{C Source Files}   {.c}      TEXT}
    {{GIF Files}        {.gif}        }
    {{GIF Files}        {}         GIFF}
    {{All Files}        *             }}

button .ob -text "Open..." -command "set open \[tk_getOpenFile -filetypes \$types \]"

label .s -text "File to save:"
entry .se -textvariable save
button .sb -text "Save..." -command "set save \[tk_getSaveFile\]"

# Create a dismiss button

button .b -text "Dismiss" -command "exit"

# Manage the widgets
```

```
grid .o -row 0 -column 0 -sticky e -padx 10
grid .oe -row 0 -column 1 -padx 10
grid .ob -row 0 -column 2 -padx 10
grid .s -row 1 -column 0 -sticky e -padx 10
grid .se -row 1 -column 1 -padx 10
grid .sb -row 1 -column 2 -padx 10
grid .b -row 2 -pady 10
```

As you can see from the code, these dialog routines have a way of specifying file filters using patterns. When you run the above example and click "Open...," you will see the following:





Color Schemes

When you create new widgets using Tk widget commands in Tk 4.0 and later versions, all the widgets have a black foreground and a gray background. So if you create a complete application (like most of the examples above), that application will have a gray background and a black foreground. What if you wanted to create an application with a light blue background? One way to accomplish this is to configure all of the created widgets' backgrounds to light blue. This will make the application code bloated and unreadable, however, since most of the commands in the code will be configuration commands—obscuring the application logic. To solve this problem, Tk provides a convenient way of globally changing the color scheme of the application. The following commands are used to set overall color scheme for any application.

```
tk_setPalette background
tk_setPalette name value ?name value ...?
tk_bisque
```

If `tk_setPalette` is invoked with one argument, then that argument is taken as the default background color for all widgets and `Tk_setPalette` will compute the color palette using this color. For example, the commands

```
tk_setPalette steelblue
button .b -text "Linux is cool!"
pack .b
```

will create a button and display it in a steel blue background. It will also set the background color of any future widgets in the same application to steel blue.
Alternatively, the arguments to `tk_setPalette` may consist of any number of name-value pairs, where the first argument of the pair is the name of an option in the Tk option database and the second argument is the new value to use for that option. The following option database names can be specified currently.

`activeBackground`	`disabledForeground`	`foreground`
`highlightColor`	`highlightBackground`	`insertBackgroun d`
`selectBackground background`	`selectColoractiveForeground`	`selectForegroun d`
`troughColor`		

Refer to the `options(n)` manual page on the option database description. `tk_setPalette` tries to compute reasonable defaults for any options that you don't specify. You can specify options other than the ones above and Tk will change those options on widgets as well.

The procedure `tk_bisque` is provided for backward compatibility: It restores the application's colors to the light brown ("bisque") color scheme used in Tk 3.6 and earlier versions.

Fonts

If you have ever programmed using Xlib or Motif on an X Window System, you know that fonts are one of the murkier areas of X. You have to specify font names in X Logical Font Description (XLFD) structures. For example, in applications created with pre–Tk 8.0 versions, if a button had to be created with a specific font, the command would look something like this:

```
Button .b –text  "Hello" –font -font -*-Courier-Bold-O-Normal--*-120-*-*-*-*-*-*
```

We actually encountered this notation earlier, but we're going to see how we can get around this ugly format now. The reason X was designed this way was to adhere to the requirement that X client applications be portable across server implementations, with very different file systems, naming conventions, and font libraries. X clients must also be able to dynamically determine the fonts available on any given server, so that understandable information can be presented to the user and intelligent fallbacks can be chosen. XLFD provides an adequate set of typographic font properties, such as `FOUNDRY`, `FAMILY_NAME`, `WEIGH_NAME`, and `SLANT`. To learn more about XLFD, refer to its specification in the X Windows system documentation or play with the `xfontsel` command on your Linux box.

Even though XLFD is extremely powerful and flexible, it is not simple and intuitive. As in the case of colors, humans tend to associate simple names with fonts, such as "Helvetica 12-point italic." Porting Tk to other, non-X platforms introduced another complexity because other windowing systems do not use XLFD, so the users were forced to learn XLFD. In addition, XLFD does not support a way of creating fonts by name.
Tk 8.0 introduced a new mechanism to deal with fonts, using the `font` command. New named fonts can be created using human-readable metrics. Tk internally will take care of translating these fonts to system-specific interfaces. One advantage of the new `font` command is that it gives a platform-independent way

of specifying fonts. It also provides a way to associate names with created fonts. The `font` command syntax is as follows:

```
font create ?fontname? ?option value ...?
font configure fontname ?option? ?value option value ...?
```

The `font` command takes options such as `–family`, `-size, and –slant`. For a complete list of `font` command options, take a look at the `font` manual page.

For example, you can create a font called `myfont` using this command:

```
font create myfont -family Courier -size 20 -weight bold -slant italic\
          -underline 1 -overstrike 1
```

Once the font is created, you can use that font name to specify a value for a `–font` widget option. After creating the font, if you run the following code using `wish`,

```
button .b -text "Hello World!" -font myfont
pack .b
```

You should see the following:

Bindings

Once we've created widgets, we can attach event handlers to them to make them respond to the user. For example, in the final "Hello World" program, we attached an event handler to the button:

```
bind .b <Control-Button-1> {puts "Help!"}
```

We use the `bind` command to attach these event interactions to the widgets. The `bind` command associates Tcl scripts with X events and is very powerful. Its general syntax is

```
bind tag
bind tag sequence
bind tag sequence script
bind tag sequence +script
```

The `tag` argument determines which window(s) the binding applies to. If `tag` begins with a dot, as in `.a.b.c`, then it must be the path name for a window; otherwise, it may be an arbitrary string. Each window has an associated list of tags, and a binding applies to a particular window. If its tag is among those specified for the window, the default binding tags provide the following behavior:

- If `tag` is the name of an internal window, the binding applies to that window.
- If `tag` is the name of a top-level window, the binding applies to the top-level window and all its internal windows.
- If `tag` is the name of a class of widgets, such as `button`, the binding applies to all widgets in that class.

- If `tag` has the value `all`, the binding applies to all windows in the application.

For example, let's see what happens if we invoke `bind` on the `button` class:

```
% bind Button
<Key-space> <ButtonRelease-1> <Button-1> <Leave> <Enter>
```

The result says that there are bindings for these five event sequences in the `button` class. Let's experiment further and see what happens when we invoke the second form of `bind` command on one of these bindings:

```
% bind Button <Key-space>
    tkButtonInvoke %W
```

The result says that `<Key-space>` binding on the `button` widget class (all the push buttons belong to this `button` class) will invoke the Tcl script `tkButtonInvoke` with the button path as the argument.

We can also use class names when we associate the binding. For example,

```
% bind Button <Control-Button-1> {puts "Help!"}
```

will set the `<Control-Button-1>` binding on all the button widgets in the application. It's possible for several bindings to match a given X event. If the bindings are associated with different tags, each of the bindings will be executed in order. By default, a widget class binding will be executed first if it exists, followed by a binding for the widget, a binding for its top level, and finally, an `all` binding if one exists for that event. We can use the `bindtags` command to change this order for a particular window, or to associate additional binding tags with the window.

When a binding matches a particular sequence, the script associated with that binding will be invoked. While we're invoking the script, we can inform the `bind` command to pass some arguments to that script from the X event that invoked the binding. For this we use special modifiers. For example, in the canvas example, we had a binding

```
bind $c <1> "itemDragStart $c %x %y"
```

There we informed `bind` that while invoking the `itemDragStart` command, it should pass `$c`, `%x`, and `%y`, which `bind` replaces with the x and y coordinates of the X event structure. The `bind` command supports lots of substitution parameters; for a complete listing, refer to the `bind man` page.

Binding Tags

As just described, the `bind` command is used to associate an action with a binding. When an association is created with the `bind` command, a tag is specified. The `tag` argument specifies which windows the binding applies to. Usually, `tag` is the name of the widget, the name of the widget class, the keyword `all`, or any other text string. Each window has an associated list of tags, and a binding applies to a particular window if its tag is among those specified for the window. When an event occurs in a window, it is applied to each of the window's tags in order; for each tag, the most specific binding that matches the given tag and event is executed. For example, after executing the following code snippet in the `wish` shell,

```
bind . <F1> "puts Toplevel"
entry .e
pack .e
bind .e <F1> "puts Entry"
```

if you press the F1 key inside the entry widget, you will see the strings "`Entry`" and "`Toplevel`" printed in that order, because if you invoke the `bindtags` command on `.e`, the result will be `.e Entry . all`; this means that when an event is triggered on `.e`, it is first checked in the `.e` tag and later in the top level that includes the entry. What if you want to make the top-level binding fire before the entry's binding? You can use the `bindtags` command to change the order:

```
Bindtags .e {. .e Entry all}
```

By default, each window has four binding tags consisting of the following, in order:

- The name of the window
- The window's class name
- The name of the window's nearest top-level ancestor
- All

Top-level windows have only three tags by default, since the top-level name is the same as that of the window.

The `bindtags` command can be used to introduce arbitrary additional binding tags for a window. In fact this function of `bindtags` accomplishes many things. It aids in creating a binding once and and associating it with as many widgets as needed by simply inserting the binding tag in the widget's bind tags list. Second, it allows widgets to have more than the standard four binding tags. Tags aid in identifying an action by name rather than by a key sequence. The following example illustrates a practical use for binding tags.

```
set count 0
button .b -text "Tick(ms)"
label .ticker -textvariable count
pack .b .ticker

bind timer <ButtonPress-1> {
    set count 0
    StartTimer %W
}

bind timer <ButtonRelease-1> {
    StopTimer %W
}

proc StartTimer { widget } {
    global pending count
    incr count 200
    set pending [after 200 [list StartTimer $widget]]
}

proc StopTimer { widget } {
    global pending
    after cancel $pending
}

bindtags .b [linsert [bindtags .b] 0  timer]
```

In this example we first created a simple user interface with a button and a label to display timer ticks. Next we created a binding with a tag `timer`. We added two key sequences to the tag timer. The first sequence, `<ButtonPress-1>`, starts the timer, and `<ButtonRelease-1>` stops the timer. The code is pretty simple to understand. The point to observe here is the use of `bindtags`. We have easily added these key sequences to the button `.b` in a single line. Without the `bindtags` command, we would have to do something like

```
bind .b  <ButtonPress-1> "+{set count 0; StartTimer %W}"
```

and the same for the binding `<ButtonRelease-1>`. Also, these bindings do not really imply what we are trying to achieve. Using the `bindtags` command, we have identified these sequences with the name `timer`. If we create another button to handle another timer, all we have to do is invoke the `bindtags` command on that button and insert the binding.

Geometry Management

After we've created the widgets and bound the event handlers using `bind`, we need to arrange the widgets on the screen in a way that makes the GUI meaningful and useful. Geometry managers perform this job. Tk currently supports three explicit geometry managers:

- Packer, using the `pack` command
- Placer, using the `place` command
- Table or grid manager, using the `grid` command

Packer

We use the `pack` command to arrange the slave widgets of a master window or widget in order around the edges of the master. The syntax of the `pack` command takes one of these forms:

```
pack option arg ?arg ...?
pack slave ?slave ...? ?options?
pack configure slave ?slave ...? ?options?
```

Let's look at an example and explore some of the `pack` options:

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

foreach i  {1 2 3 4 } {
    button .b$i -text "Btn $i"
    pack .b$i -side left -padx 2m -pady 1m
}
```

This will produce the following output:

Here the buttons are packed to the left, with a horizontal space of two millimeters between them and with a space of one millimeter vertically to the master's boundary.

In the pack sequence

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

foreach i  {1 2 3 4 } {
    button .b$i -text "Btn $i"
    pack .b$i -side left -ipadx 2m -ipady 1m
}
```

`-ipadx` specifies that the slaves be internally padded with two millimeters horizontally. Internal padding causes the slave (button) to expand to fill the extra space created.

> **There are many more combinations of the `pack` command. Refer to the `pack man` page and Ousterhout's *Tcl and the Tk Toolkit*, Addison-Wesley (ISBN 0-201-63337-X), for more information.**

Placer

The placer geometry manager is used for fixed placement of windows, where you specify the exact size and location of one window (the slave) within another window (the master). We'll use the `image` example to explain the placer geometry manager. The following code fragment shows how the buttons are created in the puzzle.

```
set order {3 1 6 2 5 7 15 13 4 11 8 9 14 10 12}
for {set i 0} {$i < 15} {set i [expr $i+1]} {
    set num [lindex $order $i]
    set xpos($num) [expr ($i%4)*.25]
    set ypos($num) [expr ($i/4)*.25]

    set x [expr $i%4]
    set y [expr $i/4]

    set butImage [image create photo image-${num} -width 40 -height 40]
    $butImage copy  $image  -from [expr round($x*40)] \
                                  [expr round($y*40)] \
                                  [expr round($x*40+40)] \
                                  [expr round($y*40+40)]
    button .frame.$num -relief raised -image $butImage \
                       -command "puzzleSwitch  $num" \
                       -highlightthickness 0
    place .frame.$num -relx $xpos($num) -rely $ypos($num) \
                      -relwidth .25 -relheight .25
}
```

This loop creates buttons and places them relative to the master, `.frame`. Here, `-relx 0` is the left edge of the master and `-relx 1` is the right edge of the master, and similarly for `-rely 0` and `-rely 1`. Now if you decipher the loop code, you'll see how all the buttons are arranged to form the puzzle.

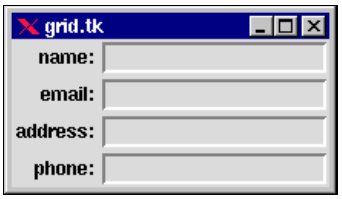
Grid

The grid geometry manager arranges widgets (slaves) in rows and columns inside another window, called the **geometry master**. Grid is a very powerful geometry manager; using it, we can create complex layouts with ease. Let's see just see how simple it is to create an entry for personnel information using 10 lines of Tk code:

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

```
set row 0
foreach item {name email address phone} {
  label .$item-label -text "${item}:"
  entry .$item-entry  -width 20
  grid .$item-label -row $row -column 0 -sticky e
  grid .$item-entry -row $row -column 1 -columnspan 2 -sticky "ew"
  incr row
}

grid columnconfigure . 1 -weight 1
```

If you run this program, you'll see output like this:



Here the slaves `.$item-label` and `.$item-entry` are arranged in the master, using the `-row $row` and `-column` options. You can also specify row and column span options for the slave using the `-rowspan` and `-columnspan` options. These options will span the slaves to occupy `span` number of rows or columns or both. The line

```
grid .$item-entry -row $row -column 1 -columnspan 2 -sticky "ew"
```

specifies that the `entry` widget will occupy two columns, and `-sticky "ew"` implies that the slave will stretch from east to west in the parcel space available for it. If you specify just one letter in the `-sticky` option, it behaves as an anchoring option. The last line,

```
grid columnconfigure . 1 -weight 1
```

specifies that if the master (`.`) is resized horizontally, then column 1 should get the resized portion.

In developing these examples, we've made much use of the grid geometry manager. This is because grid makes it so much easier to design and understand layouts. It's been in Tk since version `4.1`.

Focus and Navigation

When you have multiple top-level windows on your computer screen and you press a key, which one of the windows will receive the key press event? The answer is the top-level window with the focus. So focus determines the target of the keyboard input. Top-level window focus management is done automatically by the window manager. For example, some window managers automatically set the input focus to a top-level window whenever the mouse enters it; others redirect the input focus only when the user clicks on a window. Usually, the window manager will set the focus only to top-level windows, leaving it up to the application to redirect the focus among the children of the top level.

Tk provides two application-level focus models: `implicit`, which sets the focus to the widget the mouse is currently on, and `explicit`, where the user must either explicitly click on the widget or navigate to

that widget using the keyboard. By default, tab keys are used to navigate focus between widgets in an explicit model.

Tk remembers one focus window for each top level (the most recent descendant of that top level to receive the focus); when the window manager gives the focus to a top level, Tk automatically redirects it to the remembered window. Within a top level, by default, Tk uses an explicit focus model. Moving the mouse within a top level does not normally change the focus; the focus changes only when a widget decides explicitly to claim the focus (e.g., because of a button click) or when the user types a key, such as Tab, that moves the focus.

The focus command syntax is as follows:

```
focus
focus window
focus option ?arg arg ...?
```

For a complete list of the `focus` command's usage, refer to `focus` manual page.

Once the application or any of its top-level windows gets focus, the Tcl procedure `tk_focusFollowsMouse` may be invoked to create an implicit focus model. It reconfigures Tk so that the focus is set to a window whenever the mouse enters it. For example, the following example will instruct the window manager to give focus to whichever component the mouse is on, once the application top level gets the focus. If you run the example, you will notice that the buttons get focus as soon as the mouse is moved over them without a mouse click.

```
tk_focusFollowsMouse

button .b1  -text "Button 1"
button .b2  -text "button 2"
button .b3  -text "button 3"

pack .b1 .b2 .b3 -side left -padx 10
```

The Tcl procedures `tk_focusNext` and `tk_focusPrev` implement a focus order among the children of a top level; among other things, these are used in the default bindings for Tab and Shift+Tab.

The syntax of the `tk_focusNext` and `tk_focusPrev` commands is as follows:

```
tk_focusNext window
tk_focusPrev window
```

`tk_focusNext` is a utility procedure used for keyboard traversal. It returns the "next" window after `window` in focus order. The focus order is determined by the stacking order of windows and the structure of the window hierarchy. Among siblings, the focus order is the same as the stacking order, with the lowest window being first. If a window has children, the window is visited first, followed by its children (recursively) and then by its next sibling. Top-level windows other than `window` are skipped, so that `tk_focusNext` never returns a window in a different top level from `window`.

After computing the next window, `tk_focusNext` examines the window's `-takefocus` option, to see whether it should be skipped. If so, `tk_focusNext` continues on to the next window in the focus order until it eventually finds a window that will accept the focus or returns back to `window`.

The `tk_focusPrev` command is similar to `tk_focusNext`, except that it returns the window just before the window in the focus order.

The following example illustrates how a widget can avoid the focus by specifying the `–takefocus 0` option. When you run the example, the "skip focus" button does not take focus, even though the mouse is on it.

```
tk_focusFollowsMouse

button .b1  -text "Button 1"
button .b2  -text "skip focus" -takefocus 0
button .b3  -text "button 3"

pack .b1 .b2 .b3 -side left -padx 10
```

Option Database

Just as in Motif or Xt, every widget in Tk has a **class,** which can be retrieved using the command

```
winfo class widget-path-name
```

These class names are used to specify application defaults and class bindings for the widget. Tk uses a special database, called the **option database**, to store and retrieve application resources. For example, in the very first program, we used the line

```
option add *b.activeForeground brown
```

This informed the option database that buttons with name `.b` (it can have any parent) should have a brown `activeForeground` color. We could have specified the button class instead and had the same effect:

```
option add *Button.activeForeground brown
```

The reason these commands change `activeForeground` is that when Tk creates a widget, after setting the command line it searches the option database to set the appropriate resources. If it finds a match to the resource, it will use that option; otherwise it will use a default value.

The `option` command does the same things as an X resource file. In fact, we can store all the resources in a file and let the `option` command read the file, just as we did with `hello4.tk` program. We can also use the `option` command to query the options stored, using the syntax

```
option get window name class
```

The option database is very versatile, much more so than the simple `.Xdefaults` file. We can use it to simulate the same application default loading mechanism that is supported by any Xt-based application. For example, before the application is loaded, the defaults file can be located in the directories specified by X Windows environment variables, such as `XFILESEARCHPATH`, `XAPPLRESDIR`, `XUSERFILESEARCHPATH` and `XENVIRONMENT`.

We can now assign priorities to the application defaults file found in those directories while reading them into the application using the `option readfile ...` command. The code for this emulation would look something like this:

```
global env

if [info exists env(XFILESEARCHPATH)] {
   look for the app-defaults file in XFILESEARCHPATH dir
   load the file with priority 1
   } else {
   look in one of {/usr/lib/X11/app-defaults, /usr/openwin/lib/app-defaults,
/usr/lib/app-defaults..} directories and load the file with priority 1
   }
   if [info exists env(XUSERFILESEARCHPATH)] {
      look for the app-defaults file in XUSERFILESEARCHPATH dir
      load the file with priority 2 over riding XFILESEARCHPATH priority
       } elseif [info exists env(XAPPLRESDIR)] {
             look for the app-defaults file in XAPPLRESDIR dir
      load the file with priority 2 over riding XFILESEARCHPATH priority
       } elseif
             load app defaults file if exists from current dir with priority 2
        }
   if [ the defaults exist in .Xdefaults ] {
      load them with priority 3  }
        if [info exists env(XENVIRONMENT)] {
  load the file XENVIRONMENT as the app defaults file with second-highest priority
   }
   finally load command-line options with the highest priority
```

Inter-application Communication

Tk provides a very powerful mechanism for two applications that share a display server (though they can be on different screens) to communicate with each other, using the `send` command. For example, application `A` can send application `B` (i.e., with Tcl interpreter name `B`) a command to output the string "`hello`":

```
send B [list puts "hello"]
```

Application `B` will receive this command and execute the `puts "hello"` command. For more information on this, refer to the `send` `man` page.

Selection

Imagine the scenario where a user is operating on a desktop with multiple xterms. The user highlights a selection of text in one xterm, using the left mouse button, and then pastes the selection in another xterm, using the middle button. There are a lot of things going on under the covers during this transaction.

When the user decides to select something in an xterm, the xterm must first of all figure out what information is being selected and then it should become the selection owner. Being the selection owner means that when another application decides to request the selection, the owner should convert the selection to the type specified by the requested application. A client wishing to obtain the selection in a particular format requests the selection from the selection owner. All of these cooperative interactions between X clients is described in the X Inter-Client Communication Conventions Manual (ICCCM).

Selection can be of various types: `PRIMARY`, `SECONDARY`, and `CLIPBOARD`. By default, the `PRIMARY` selection, named `XA_PRIMARY`, is used by all the clients. A `SECONDARY` selection, named `XA_SECONDARY,` is used when applications need more than one selection. The `CLIPBOARD` selection is usually used to hold deleted data.

The `selection` command provides a Tcl interface to the X selection mechanism and implements the full selection functionality described in the X Inter-Client Communication Conventions Manual (ICCCM). The following example shows selection manipulation. It creates a new slave Tk interpreter, with a text widget.

It automatically selects the text inside the text widget using the `sel` tag of `text`. Using the `sel` tag of the `text` command will make the selection by default `PRIMARY`. So when the slave interpreter calls the `selection own`, it is owning the primary selection. The master interpreter then retrieves the selection and outputs it to `stdout`.

```
#!wish
interp create foo

foo eval {
    load {} Tk
    text .t
    pack .t
    .t insert end "Hello World!"
    .t tag add sel  0.0 end
    selection own
    .t insert end "\n"
#   .t insert end "[selection get -selection SECONDARY]"
    .t insert end "[selection get ]"


}
puts "[selection get]"
```

If the commented line in the above example is uncommented, the application will output an error because there is no secondary selection. A selection owner can also reject selection retrievals by any other application or component.

Clipboard

In X, `CLIPBOARD` is another type of selection mechanism. The `CLIPBOARD` selection can be used to hold deleted data. For example, a client can post deleted data to the clipboard and exit. Another client can retrieve the deleted selection from the clipboard, even thought the original client no longer exists. This is not possible with the `PRIMARY` and `SECONDARY` selection types, because when a client requests `PRIMARY` or `SECONDARY` selection, the owner will be sent a request. If the owner no longer exists, the selection request will fail.

The `clipboard` command provides a Tcl interface to the Tk clipboard, which stores data for later retrieval, using the `selection` mechanism. `Tk_clipboard` is not the same as the system clipboard that you see on various operating systems. `Tk_clipboard` is designed to hold deleted data between applications developed using the Tk toolkit. To copy data to the clipboard, `clipboard clear` must be called, followed by a sequence of one or more calls to `clipboard append`. The following command illustrates the use of the `clipboard` command.

```
#!wish
interp create foo

foo eval {
    load {} Tk
    clipboard clear
    clipboard append -type STRING "Clipboard Data"

}
       interp delete foo
puts "[selection get -selection CLIPBOARD]"
```

This example creates a new Tk interpreter called `foo`. `foo` appends data to the clipboard. The master interpreter deletes the slave interpreter and retrieves the data from the clipboard. As you can see, even though the slave does not exist, the selection can still be retrieved.

Window Manager

Tk provides the `wm` command so that windows can communicate with the window manager. Window manager functions typically include managing the keyboard focus between application windows, setting up top-level window properties, allocating colormaps to windows, and positioning top-level windows on the screen. Since window manager deals only with the top-level windows of any application and leaves the internal window management to the application, `wm` command arguments must be top-level windows. The kinds of functions that a client can request from the window manager include the following:

- Iconifying and de-iconifying top-level windows
- Positioning top-level windows at a particular point on the screen
- Requesting the initial sizes of top-level windows
- Setting the titles of top-level windows
- Setting the focus model of an application
- Requesting the height and width of a top-level window
- Overriding the default window manager decorations

For example, using the `wm` command, one can query the state of a top-level window in an application as follows:

```
% wm iconify .
% puts "[wm state .]"
iconic
```

Users can also set up window manager protocols on a top-level window. For example, we can set up a handler on a top-level window that will get called when the window receives focus or when the window is deleted.

```
% wm protocol . WM_TAKE_FOCUS {puts "window . got focus"}
% wm protocol . WM_DELETE_WINDOW {puts "window . is being deleted"; exit}
```

When run, this code snippet will output the string "`window . got focus`" when the top-level window "`.`" gets focus, and the string "`window . is being deleted`" when the top-level window "`.`" is deleted using the window manager delete button.

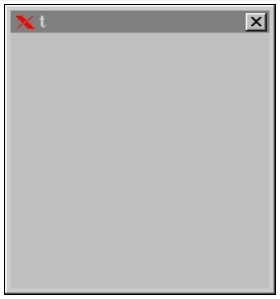
`wm` commands can also be used to set or query the title of a top-level window as follows:

```
wm title ?newtitle?.
```

A client can also request that a window manager not decorate a top-level window. When the client makes such a request, the window manager will not display the iconify, de-iconify, or resize button on the window manager frame of the requested top level. The following code snippet requests the window manager not to add any decorations to the top-level window`.`

```
toplevel .t
wm withdraw .t
wm transient .t .
wm deiconify .t
```

When this code is run, you will see the following output:



Actually, the code snippet shows more than one functionality of the window manager command. The application first requests the window manager to withdraw the top-level window, it then asks the window manager to remove the minimize/maximize decorations for the top-level window, and finally it asks the window manager to map the window back to the screen.

Dynamic/Static Loading

In the previous sections, we created new interpreters and loaded Tk statically. There are two ways in which interpreters can load the Tk toolkit—statically and dynamically, using the `load` command. For static loading, the executable should be preloaded with Tk. For example, the code

```
interp create debugInterp

debugInterp eval {
    load {} Tk
    text .t
    pack .t
    update
}

proc debug {interp args} {
    debugInterp eval [list  .t insert end "$args"]
}

debug . "hello world!"
```

statically loads the Tk executable to a newly created interpreter. The example also shows how to communicate between interpreters. The master interpreter creates a `debug` routine, which communicates with `debugInterp` to display debugging information.

So how do we load a Tk interpreter dynamically into a `tclsh` application? The following examples illustrate how.

```
interp create debugInterp

debugInterp eval {
    load /usr/local/tk8.2b3/unix/libtk8.2.so Tk
    text .t
    pack .t
    update
}

proc debug {interp args} {
    debugInterp eval [list  .t insert end "$args"]
}
```

```
debug . "hello world!"
vwait foob
```

This code assumes that you have compiled the Tcl/Tk distributions so that they are dynamically loadable and that the Tk dynamic library `libtk8.2.so` is located in the `/usr/local/tk8.2b3/unix/` directory. As you can see from the code, `tclsh` creates a new interpreter and loads Tk dynamically into the newly created interpreter, and the master interpreter enters the event loop using the `vwait` command. If the master interpreter does not enter the event loop, the application exits without warning. If you run the preceding code using the command

```
Tclsh8.0 dynamicLoad.tk
```

the Tk interpreter will be loaded dynamically!

Safe Tk

Suppose you download a Tcl script from the network and execute it. If the script is malicious, it can do a lot of damage to your system. For example, it can delete all of your files or transfer files to another computer. How do you ensure that untrusted scripts do not do any damage? In 1994 Marshall and Rose created Safe-Tcl, which was originally conceived as a mechanism to allow e-mail messages to contain Tcl scripts that would execute on the receiver's computer. Safe-Tcl was incorporated into the core code in Tcl version 7.5.

The goal of Safe-Tcl is to create a sandbox that allows users to safely execute untrusted Tcl scripts in the sandbox without having to worry about any side effects.

Safe interpreters have a restricted command set. The following commands are hidden in a safe interpreter.

`cd`	`exec`	`exit`	`fconfigure`
`glob`	`load`	`open`	`socket`
`source`	`vwait`	`pwd`	`file`

The `safe base` Tcl manual page describes how to create these safe interpreters. Sometimes it might be necessary to give the newly created safe interpreter some restricted access; for example, it might be allowed to open files in a particular directory. Safe-Tcl provides mechanisms for allowing such restricted access. Interpreters created with the `::safe::interpCreate` command give mediated access to potentially dangerous functionality through untrusted scripts by using the alias mechanism. Thus, Safe-Tcl is a mechanism for executing untrusted Tcl scripts safely and for providing mediated access by such scripts to potentially dangerous functionality.

Just as with Safe-Tcl, it is necessary to create sandboxes to execute untrusted Tk scripts. For example, you don't want the Tk applet to delete all of your top-level windows or to steal your X selection. Safe Tk adds the ability to configure the interpreter for safe Tk operations and load the Tk script into a safe interpreter. By default, you can't load Tk into a safe interpreter, because the safe interpreter does not allow `load` commands. Safe Tk also isolates the untrusted Tk scripts to be executed in a sandbox so that no damage can be done.

The `::safe::loadTk` command initializes the required data structures in the named safe interpreter and then loads Tk into it. The command returns the name of the safe interpreter. `::safe::loadTk` adds the value of `tk_library` taken from the master interpreter to the virtual access path of the safe interpreter, so that auto-loading will work in the safe interpreter.
The following examples show that you cannot load Tk into a safe interpreter unless you use the `::safe::loadTk` command.

```
::safe::interpCreate  safeInterp

::safe::interpAddToAccessPath safeInterp /tmp

::safe::loadTk safeInterp

interp create -safe safeInterp2

puts " 1 -> [interp hidden safeInterp]"
puts " 2 -> [interp hidden safeInterp2]"

puts " 1 -> [interp aliases safeInterp]"
puts " 2 -> [interp aliases safeInterp2]"

safeInterp eval {
    text .t
    pack .t
    update
}

safeInterp2 eval {
     load {} Tk
 }
```

When you run this script, you'll see the error message:

```
$ wish safeInterp.tk
 1 -> file socket send open pwd glob exec encoding clipboard load fconfigure source
exit toplevel wm grab menu selection tk bell cd
 2 -> file socket open pwd glob exec encoding load fconfigure source exit cd
 1 -> file load source exit encoding
 2 ->
Error in startup script: invalid command name "safeInterp2"
    while executing
"safeInterp2 eval {
     load {} Tk
 }"
    (file "safeInterp.tk" line 26)
```

This informs you that you cannot load Tk into a safe interpreter unless it is created using the `::safe::createInterp` command. If you want to see how safe interpreters are used, take a look at the `safeDebug.tk` example in the distribution.

A Mega-Widget

We have now seen how to use various widget commands to create applications. Sometimes people need to display their information in ways that require new types of widgets, such as panes, spreadsheets, notebooks, and spinboxes. Although the Tk team at Scriptics is planning to add these widgets to the core, as of this writing they have not been added. So the question is, how does one go about creating custom widgets?

There are two ways to do this. One way is to use Tcl's and Tk's C extension API, also known as TEA. The other way is to use pure Tcl and existing Tk widgets. Since we have not yet discussed Tcl's and Tk's C API, and it would in fact merit far more coverage than we can give it here, let's go ahead and use pure Tcl and existing Tk widgets to create a mega-widget. The mega-widget that we are going to create is a tree widget. As of now, Tk does not have a built-in tree widget. This widget example is by no means complete, but it is quite useful and will steer you in the right direction.

The rendering algorithm and some of the interface ideas in this example are taken from GPLed comp.lang.tcl posts. None of the other implementations I have encountered have the flexibility of this package/namespace-based implementation.

We will use Tcl's package and namespace mechanisms to encapsulate our tree widget into an abstract data structure. The next question is, what kind of widget and configuration commands should we provide for this tree widget? We should provide most of the standard configuration options such as `-font`, `-backgroundcolor`, and some more tree-specific options. Since the tree widget supports hierarchies, we should provide methods such as `additem`, `delitem`, `config`, `setselection`, and `getselection`. So let's go ahead and define the tree widget package as follows.

```
#

#       Sample tree mega-widget.Can be used to display hierachies. The clients
#       who use this package need to specify parent and tail procedures for any
#       element of the tree hierarchy. All the nodes that get stored inside the
#       tree are complete path names separated by '/'. The top-level node is
#       always /
#

package provide tree 1.0

namespace eval tree {
   variable tree


   #
   # default font setup
   #

   switch $tcl_platform(platform) {
      unix {
    set tree(font) \
        -adobe-helvetica-medium-r-normal-*-11-80-100-100-p-56-iso8859-1
      }
      windows {
    set tree(font) \
        -adobe-helvetica-medium-r-normal-*-14-100-100-100-p-76-iso8859-1
      }
   }

   #
   # Bitmaps used to show which parts of the tree can be opened/closed
   #

   set maskdata "#define solid_width 9\n#define solid_height 9"
   append maskdata {
      static unsigned char solid_bits[] = {
    0xff, 0x01, 0xff, 0x01, 0xff, 0x01, 0xff, 0x01, 0xff, 0x01, 0xff, 0x01,
    0xff, 0x01, 0xff, 0x01, 0xff, 0x01
      };
```

```
   }
   set data "#define open_width 9\n#define open_height 9"
   append data {
      static unsigned char open_bits[] = {
    0xff, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x7d, 0x01, 0x01, 0x01,
    0x01, 0x01, 0x01, 0x01, 0xff, 0x01
      };
   }

   set tree(openbm) [image create bitmap openbm -data $data \
          -maskdata $maskdata \
          -foreground black -background white]

   set data "#define closed_width 9\n#define closed_height 9"
   append data {
      static unsigned char closed_bits[] = {
    0xff, 0x01, 0x01, 0x01, 0x11, 0x01, 0x11, 0x01, 0x7d, 0x01, 0x11, 0x01,
    0x11, 0x01, 0x01, 0x01, 0xff, 0x01
      };
   }
   set tree(closedbm) [image create bitmap closedbm -data $data \
            -maskdata $maskdata \
            -foreground black -background white]


    namespace export create additem delitem config setselection getselection
    namespace export openbranch closebranch labelat
}
```

As shown in the example, the tree widget exports one package variable called `tree` as well as methods such as `additem`, `delitem`, `config`, and `setselection`. A `tree` package variable is used to hold instance information for all the trees created. As you can see from the code, `tree` also creates some images to display open and closed branches and stores them inside the tree data structure.

Our next step is to define the widget commands. First, we will define the `tree::create` command. This command basically parses the configuration options and creates a canvas with the path name specified by the `create` command. The `create` command also looks for `-parent` and `-tail` widget creation options. These options are procedures specified by the client so that the tree command can determine the parent and tail of any of its nodes. The tail is basically the end part of the node name. For example, suppose the node is named `a/b/c`. The `tail` command will return `c` (presuming the character "`/`" is used as a path separator). The `parent` command will return `/a/b`. The tree mega-widget enforces "`/`" to be the path separator and all nodes are represented with absolute paths.
The philosophy behind these `-parent` and `-tail` option commands is to allow the tree to display any hierarchical information, and not just directory structures. The `create` command also initializes variables, such as `selection` and `selidx`, which are the currently selected node and its tag. The `create` command arranges the tree to be redrawn at a later time.

```
# tree::create --
#
#   Create a new tree widget. Canvas is used to emulate a tree
#   widget. Initialized all the tree-specific data structures. $args become
#   the configuration arguments to the canvas widget from which the tree is
#   constructed.  #

# Arguments:
#    -paren   proc
#
#      sets the parent procedure provided by the application. tree
#      widget will use this procedure to determine the parent of an
#      element. This procedure will be called with the node as an
#      argument
```

```
#
#   -tail    proc  [Given a complete path this proc will give the end-element
#                        name]
#
# Results:   A tree widget with the path $w is created.
#

proc tree::create {w args} {
   variable tree
   set newArgs {}

   for {set i 0} {$i < [llength $args]} {incr i} {
      set arg [lindex $args $i]
      switch -glob -- $arg {
    -paren* {set tree($w:parenproc) [lindex $args [expr $i +1]]; incr i}
    -tail* {set tree($w:tailproc) [lindex $args [expr $i +1]]; incr i}
    default {lappend newArgs $arg}
    }
      }

   if ![info exists tree($w:parenproc)] {
      set tree($w:parenproc) parent
   }

   if ![info exists tree($w:tailproc)] {
      set tree($w:tailproc) tail
   }

  eval canvas $w -bg white $newArgs
  bind $w <Destroy> "tree::delitem $w /"
  tree::DfltConfig $w /
  tree::BuildWhenIdle $w
  set tree($w:selection) {}
  set tree($w:selidx) {}
}
```

When the tree is created, a root node by the name of / is automatically created. Every time a new node is added, the nodes are initialized with some default configuration, including, for example, the children associated with the node, whether the node should be displayed open, and the icon and tags associated with the node. `tree::DfltConfig` is for node initialization.

```
# tree::DfltConfig  --
#
#   Internal function used to initialize the attributes associated with an item/node.
Usually called when an item is added into the tree
#
#  Arguments:
#   wid   tree widget
#   node  complete path of the new node
#
#  Results:
#   Initializes the attributes associated with a node.


proc tree::DfltConfig {wid node} {
  variable tree
  set tree($wid:$node:children) {}
  set tree($wid:$node:open) 0
  set tree($wid:$node:icon) {}
  set tree($wid:$node:tags) {}

}
```

Just like any other Tk widget, tree widget should support the `-config` widget method. Tree widget supports this using the `tree::config` class method.

```
# tree::config --
#
#   Function to set tree widget configuration options.
#
#  Arguments:
#   args  any valid configuration option a canvas widget takes
#
#  Results:
#   Configures the underlying canvas widget with the options
#

proc tree::config {wid args} {
variable tree
    set newArgs {}
    for {set i 0} {$i < [llength $args]} {incr i} {
      set arg [lindex $args $i]
      switch -glob -- $arg {
    -paren* {set tree($w:parenproc) [lindex $args [expr $i +1]]; incr i}
    -tail* {set tree($w:tailproc) [lindex $args [expr $i +1]]; incr i}
    default {lappend newArgs $arg}
    }
      }
  eval $wid config $newArgs
}
```

Now that we are done with the creation and configuration of the tree widget, the next step is to add an item. This routine makes sure that a duplicate item is not added to the tree. It finds out the parent of the new item and adds it into its children. The routine also parses the item tags, such as `-image` and `-tags`, and sets the attributes of the item. The `-image` option is used to display an icon next to the item during rendering. The `-tags` option attaches tags to the newly added item. The `additem` routine also arranges the tree widget to be drawn when the application is not busy.

```
#  tree::additem --
#
#       Called to add a new node to the tree.
#
#  Arguments:
#       wid   tree widget
#       node  complete path name of the node (path is separated by /)
#       args  can be -image val, -tags {taglist} to identify the item
#
#  Results:
#       Adds the new item and configures the new item
#

proc tree::additem {wid node args} {
  variable tree
   set parent [$tree($wid:parenproc) $node]
   set n [eval $tree($wid:tailproc) $node]
  if {![info exists tree($wid:$parent:open)]} {
    error "parent item \"$parent\" is missing"
  }
  set i [lsearch -exact $tree($wid:$parent:children) $n]
  if {$i>=0} {
     return
  }
  lappend tree($wid:$parent:children) $n
  set tree($wid:$parent:children) [lsort $tree($wid:$parent:children)]
  tree::DfltConfig $wid $node
  foreach {op arg} $args {
    switch -exact -- $op {
      -image {set tree($wid:$node:icon) $arg}
      -tags {set tree($wid:$node:tags) $arg}
    }
```

```
  }
  tree::BuildWhenIdle $wid
}
```

`delitem` does the opposite of `additem` and removes the item from the tree data structure and from its parent's children list. It also arranges the tree widget to be drawn at a later time.

```
#  tree::delitem  --
#
#   Deletes the specified item from the widget
#
#  Arguments:
#   wid   tree widget
#   node  complete path of the node
#
#  Results:
#   If the node exists, it will be deleted.
#

proc tree::delitem {wid node} {
  variable tree
  if {![info exists tree($wid:$node:open)]} return
  if {[string compare $node /]==0} {
    # delete the whole widget
    catch {destroy $wid}
    foreach t [array names tree $wid:*] {
      unset tree($t)
    }
  }
  foreach c $tree($wid:$node:children) {
    catch {tree::delitem $wid $node/$c}
  }
  unset tree($wid:$node:open)
  unset tree($wid:$node:children)
  unset tree($wid:$node:icon)
  set parent [$tree($wid:parenproc) $node]
   set n [eval $tree($wid:tailproc) $node]
  set i [lsearch -exact $tree($wid:$parent:children) $n]
  if {$i>=0} {
    set tree($wid:$parent:children) [lreplace $tree($wid:$parent:children) $i $i]
  }
  tree::BuildWhenIdle $wid
}
```

The user has control over which node in the item can be assigned as a selection. The `setselection` and `getselection` routines are used to accomplish the job. The selection object is drawn with a highlighted background.

```
 tree::setselection  --
#
#   Makes the given node the currently selected node.
#
#  Arguments:
#   wid - tree widget
#   node - complete path of the one of nodes
#
#  Results:
#   The given node will be selected
#

proc tree::setselection {wid node} {
  variable tree
  set tree($wid:selection) $node
  tree::DrawSelection $wid
}
```

```
#  tree::getselection  --
#
#   Get the currently selected tree node
#
#  Arguments:
#   wid - tree widget
#
#  Results:
#   If a node is currently selected it will be returned; otherwise NULL
#

proc tree::getselection wid {
  variable tree
  return $tree($wid:selection)
}
```

The next task is building/rendering the tree. The algorithm recursively calls each node to draw itself and its children. After the tree is drawn, it will set the scroll region so that when the tree widget is associated with scrollbars it will behave properly. It also draws the current selection.

```
#  tree::Build --
#
#   Internal function to rebuild the tree
#
#  Arguments:
#   wid -  tree widgets
#
#  Results:
#
#     This routine has no complex logic in it. Deletes all the current items
#    on the canvas associated with the tree and rebuilds the tree.  #
#
#

proc tree::Build wid {
  variable tree
  $wid delete all
  catch {unset tree($wid:buildpending)}
  set tree($wid:y) 30
  tree::BuildNode $wid / 10
  $wid config -scrollregion [$wid bbox all]
  tree::DrawSelection $wid
}
```

The meat of the tree-drawing algorithm is `BuildNode`. It is a basic algorithm that draws the parent and each of its children if the open attribute of the parent node is set. If the open attribute of any of the child nodes is set, `BuildNode` will be called recursively to display its children. The rendering algorithm should be pretty self-explanatory.

```
#  tree::BuildNode --
#
#   Function called by tree::build to incrementally build each node
#
#  Arguments:
#   wid - tree widget
#       node - complete path of the node
#   in - the starting x-coordinate
#
#  Results:
#   The node gets drawn
#
proc tree::BuildNode {wid node in} {
  variable tree
```

```
  if {$node=="/"} {
    set vx {}
  } else {
    set vx $node
  }
  set start [expr $tree($wid:y)-10]
  foreach c $tree($wid:$node:children) {
    set y $tree($wid:y)
    incr tree($wid:y) 17
    $wid create line $in $y [expr $in+10] $y -fill gray50
    set icon $tree($wid:$vx/$c:icon)
    set taglist x
    foreach tag $tree($wid:$vx/$c:tags) {
      lappend taglist $tag
    }
    set x [expr $in+12]
    if {[string length $icon]>0} {
      set k [$wid create image $x $y -image $icon -anchor w -tags $taglist]
      incr x 20
      set tree($wid:tag:$k) $vx/$c
    }
    set j [$wid create text $x $y -text $c -font $tree(font) \
                                  -anchor w -tags $taglist]
    set tree($wid:tag:$j) $vx/$c
    set tree($wid:$vx/$c:tag) $j
    if {[string length $tree($wid:$vx/$c:children)]} {
      if {$tree($wid:$vx/$c:open)} {
         set j [$wid create image $in $y -image $tree(openbm)]
         $wid bind $j <1> "set tree::tree($wid:$vx/$c:open) 0; tree::Build $wid"
         tree::BuildLayer $wid $vx/$c [expr $in+18]
      } else {
         set j [$wid create image $in $y -image $tree(closedbm)]
         $wid bind $j <1> "set tree::tree($wid:$vx/$c:open) 1; tree::Build $wid"
      }
    }
  }
  set j [$wid create line $in $start $in [expr $y+1] -fill gray50 ]
  $wid lower $j
}
```

Now, after the tree gets displayed, if the user chooses to open any of the branches by clicking the "+" image next to a node, the following `openbranch` routine will arrange to redraw the tree by displaying the node's children:

```
#  tree::openbranch --
#
#       A callback that gets called to open a node to show its children
#
#  Arguments:
#       wid - tree widget
#       node - the node whose children should be shown
#
#  Results:
#       The children of the node will be drawn
#

proc tree::openbranch {wid node} {
  variable tree
  if {[info exists tree($wid:$node:open)] && $tree($wid:$node:open)==0
      && [info exists tree($wid:$node:children)]
      && [string length $tree($wid:$node:children)]>0} {
    set tree($wid:$node:open) 1
    tree::Build $wid
  }
}
```

Similarly, when the user clicks on the "-" image next to a node, the `closebranch` routine will arrange for the tree to be redrawn by closing the branch and undisplaying the children:

```
# tree::closebranch   --
#
#       The opposite of open branch, see above
#
#  Arguments:
#
#
#  Results:
#

proc tree::closebranch {wid node} {
  variable tree
  if {[info exists tree($wid:$node:open)] && $tree($wid:$node:open)==1} {
    set tree($wid:$node:open) 0
    tree::Build $wid
  }
}
```

The `DrawSelection` routine will highlight the currently selected node.

```
#  tree::DrawSelection --
#
#       Highlights the current selection
#
#  Arguments:
#       wid - tree widget
#
#  Results:
#       The current selection will be highlighted with sky blue
#

proc tree::DrawSelection wid {
  variable tree
  if {[string length $tree($wid:selidx)]} {
    $wid delete $tree($wid:selidx)
  }
  set node $tree($wid:selection)
  if {[string length $node]==0} return
  if {![info exists tree($wid:$node:tag)]} return
  set bbox [$wid bbox $tree($wid:$node:tag)]
  if {[llength $bbox]==4} {
    set i [eval $wid create rectangle $bbox -fill skyblue -outline {{}}]
    set tree($wid:selidx) $i
    $wid lower $i
  } else {
    set tree($wid:selidx) {}
  }
}
```

The `BuildWhenIdle` routine is used to minimize the drawing refreshes by collecting all the redraw routines and arranging an event handler to draw the tree.

```
#  tree::BuildWhenIdle --
#
#       Function to reduce the number of redraws of the tree. When a redraw is not
#       immediately warranted this function gets called
#
#  Arguments:
#       wid  - tree wiget
#
#  Results:
#       Set the tree widget to be redrawn in future.
#

proc tree::BuildWhenIdle wid {
```

```
  variable tree
  if {![info exists tree($wid:buildpending)]} {
    set tree($wid:buildpending) 1
    after idle "tree::Build $wid"
  }
}
```

Finally, the `tree::labelat` routine will return the node at a given `x-y` widget position. The magic is to use the canvas built-in commands:

```
#  tree::labelat --
#
#       Returns the tree node closest to  x,y coordinates
#
#  Arguments:
#       wid      tree widget
#       x,y      coordinates
#
#  Results:
#       The node closest to x,y will be returned.


proc tree::labelat {wid x y} {
  set x [$wid canvasx $x]
  set y [$wid canvasy $y]
  variable tree
  foreach m [$wid find overlapping $x $y $x $y] {
    if {[info exists tree($wid:tag:$m)]} {
      return $tree($wid:tag:$m)
    }
  }
  return ""
}
```

Since the underlying widget for the tree is canvas, this tree widget will support all the canvas binding commands with the same syntax.

Package File Generation

Now that we have defined a tree mega-widget, how do we use it? Before we dive into developing a new application using the tree widget, we have to generate a `pkgIndex` file for the tree widget. To do this, copy the `tree.tcl` file into `/usr/local/lib/tcl` and run the following command in a `wish` shell:

```
$ wish
% cd /usr/local/lib/tcl
% pkg_mkIndex . tree.tcl
```

This command will create a `pkgIndex.tcl` file in the `/usr/local/lib/tcl` directory. Make sure that there is no prior `pkgIndex.tcl` file before you create it, because if there is, Tcl will clobber it.

Once you have generated `pkgIndex.tcl` file, you need to instruct your application that you want to use the tree widget. To accomplish this, append `/usr/local/lib/tcl` to the `auto_loadpath` by adding the following lines:

```
Lappend auto_path /usr/local/lib/tcl
Package require tree
```

An Application Using the Tree Mega-Widget

Let's use our tree package and develop a simple application. The application we are going to develop will display the root system file hierarchy.

We first inform the application of the location of the package file and use it to load the tree package:

```
#!/usr/local/bin/wish
#
# Simple application showing the use of tree mega-widget
#
lappend auto_path /usr/local/lib/tk
package require tree
```

Now we have to inform the tree widget on parent and tail routines. By default, they are normal file `dirname` and `tail` routines, because we are displaying a root file system:

```
#
# Create utility procs that tree widget uses to query parent
# and tail components of a node
#
proc parent {item} {
   return [file dirname $item]
}
proc tail {item} {
   return [file tail $item]
}
```

We create images to display directory and file images:

```
# Create imanges that we use to display directory and a normal file
#

image create photo idir -data {
    R0lGODdhEAAQAPIAAAAAAHh4eLi4uPj4APj4+P///wAAAAAAACwAAAAAEAAQAAADPVi63P4w
    LkKCtTTnUsXwQqBtAfh910UU4ugGAEucpgnLNY3Gop7folwNOBOeiEYQ0acDpp6pGAFArVqt
    hQQAO///
}
image create photo ifile -data {
    R0lGODdhEAAQAPIAAAAAAHh4eLi4uPj4+P///wAAAAAAAAAAACwAAAAAEAAQAAADPkixzPOD
    yADrWE8qC8WN0+BZAmBq1GMOqwigXFXCrGk/cxjjr27fLtout6n9eMIYMTXsFZsogXRKJf6u
    P0kCADv/
}
```

Next, we create a routine that dynamically adds the children of the node if the node happens to be the directory when the user double-clicks on the item:

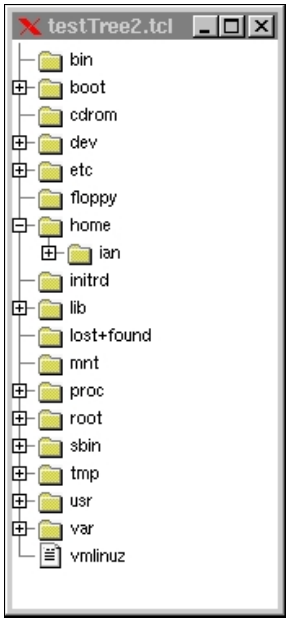
```
#
# Dynamically add entries to the tree widget
#
proc AddDir {wid dir} {
    if ![file isdirectory $dir ] {
        return;
    }
    foreach file [exec ls $dir] {
        set file [file join $dir $file]
        if [file isdirectory $file] {
            tree::additem $wid [file join $dir $file] -image idir
        } else {
            tree::additem $wid [file join $dir $file] -image ifile
        }
```

```
    }
}
```

The main process creates the tree and sets up the double-click bindings for the tree widget. It also adds the top-level node to the tree.

```
#
# main proc
#
#
# Create tree widget and set up bindings
#
tree::create .t -width 150 -height 400

#
# open a node when gets double-clicked.
#
.t bind x <Double-1> {
    puts "Callled"
    set child [tree::labelat %W %x %y]
    AddDir %W $child
    tree::openbranch %W $child
}


AddDir .t /

# manage the widget

pack .t -fill both -expand 1
update
```

When you run the program you should see something similar to the following:



Tk Process Log Viewer

Now that we have seen how to create a mega-widget, how about creating an application using Tcl 8.0's new features, such as the event mechanism? More often than not, Linux users find themselves running the `tail –f` or `find / -print` command every day. So why don't we develop an application to display the outputs of those commands in a text window? We will also create shortcuts (nicks) to the commands they run, so that they can rerun them by clicking on the shortcut.

This application supports two types of logs: file logs and command logs. File logs are `tail –f filename`–type commands, and command logs are of the type '`find / -print`'. For file log outputs, the user will specify a filename and a nick (shortcut). A command of the type `tail –f filename` will be constructed and associated with the given nick. For command logs, the user has to specify the entire command name and nickname.

Let's call this application "Tk Process Log Viewer." So what do we need to build such an application? User interface–wise, we need a menu bar for process commands, a text widget to display the output, a status bar to display error messages, a couple of entry boxes to specify commands and their nicknames, and an option button to display currently available nicks. We also need a stop button to stop the current view process, and a delete button in the menu bar to delete any unwanted shortcuts.
Let's start by declaring the global variables that we will use to build the application. We store all these global variables in an array called `tailOpts`.

```
#!/usr/local/bin/wish8.0
# logView.tk  --
#
#       Essentially a general-purpose GUI wrapper to tail, gui, and any commands
#       that will output data continuously. This GUI has the ability to record
#       the commands as smart buttons, so that you can rerun the same commands
#       again and again without having to retype.


set tailRc "~/.tailrc"
wm title .  "Process Log Viewer"
wm iconname . "Log Viewer"

global tailSize textw fileName tailFd curNick tailOpts statusImgWin

# tailSize --> size in lines of tail output to display
# fileName --> File name: variable
# tailFd   --> proc fd or file fd of current tail process
# curNick  --> current nick being shown; nick essentially a shortcut to a cmd.
# tailOpts --> saved options
# statusImgWin --> window showing what kind of error it is!

set tailSize 20
set fileName "/usr/local/processlog/logView.tk";     #include your own path here

#
# file types for the file selection dialog box.
#

set tailOpts(types) {
   {"All files"                *}
   {"Text files"               {.txt .doc}      }
   {"Text files"               {}           TEXT}

   {"Tcl Scripts"              {.tcl}       TEXT}
   {"C Source Files"           {.c .h}          }
   {"All Source Files"         {.tcl .c .h}     }
```

```
   {"Image Files"               {.gif}           }
   {"Image Files"               {.jpeg .jpg}     }
   {"Image Files"               ""    {GIFF JPEG}}
}
set tailOpts(wins) {}
```

Next, we will build the user interface. We start with the menu bar with "File" and "Edit" commands. The File menu will support the addition of new command nicks through an "Add New…" command button. The File menu will also contain an exit button to exit the application. The Edit menu will contain a "Delete Nicks" button to edit the current nicks.

```
proc BuildTailGui {w} {
    global tailSize textw fileName tailFd curNick tailOpts statusImgWin
    global viewOptMenu

    if {$w == "."} {
       set w "";
    }

    #
    # Build Menu for file
    #

    menu $w.menu -tearoff 0
    # File menu
    set m $w.menu.file
    menu $m -tearoff 0
    $w.menu add cascade -label "File" -menu $m -underline 0
    $m add command -label "Add New ..." -command {AddNew} -underline 0
    $m add command -label "Exit" -command {exit} -underline 0

    # Edit Menu

    set m $w.menu.edit
    menu $m -tearoff 0
    $w.menu add cascade -label "Edit" -menu $m -underline 0
    $m add command -label "Delete Nicks.." -command {DeleteNicks} -underline 0

    # Help Menu
    set m $w.menu.help
    menu $m -tearoff 0
    $w.menu add cascade -label "Help" -menu $m -underline 0
    $m add command -label "About..." -underline 0 -command {
        tk_messageBox -parent . -title "Process Log Viewer" -type \
        ok -message    "Tk Tail Tool \nby Krishna Vedati(kvedati@yahoo.com)"
    }
```

The routine then adds a text widget to display the output of any log process and a status bar to display error and informational messages.

Next, the routine builds rows of widgets. The first row will enable users to add file-type nicks to the application. The second row will enable users to add command-type nicks. The last row contains a stop button to stop the current log process and an option button to quickly choose a shortcut.

```
    #
    # Create status/error message window
    #

    frame $w.status  -relief sunken -bd 2
    set statusImgWin [label $w.status.flag -bitmap info]
    label $w.status.lab -textvariable statusText -anchor w -bg "wheat"
    pack $w.status.flag -side left
```

```
    pack $w.status.lab -side left -fill both -expand 1

    #
    # File name: entry panel
    #

    frame $w.file
    label $w.file.label -text "File name:" -width 13 -anchor w
    entry $w.file.entry -width 30 -textvariable fileName
    button $w.file.choose -text "..." -command \
        "set fileName \[tk_getOpenFile -filetypes \$tailOpts(types) \
         -parent \[winfo toplevel $w.file\]\];"

    button $w.file.button -text "Tail File" \
       -command "AddToView  file \$fileName"
    pack $w.file.label $w.file.entry -side left
    pack $w.file.choose -side left -pady 5 -padx 10
    pack $w.file.button -side left -pady 5 -padx 10
    bind $w.file.entry <Return> " AddToView  file \$fileName"
    focus $w.file.entry

    #
    # Command entry panel
    #
    frame $w.fileC
    label $w.fileC.cLabel -text "Command:" -width 13 -anchor w
    entry $w.fileC.cEntry -width 40 -textvariable command
    label $w.fileC.nLabel -text "Nick:" -anchor w
    entry $w.fileC.nEntry -width 15 -textvariable nick
    button $w.fileC.add -text "Add" -command "AddToView \"command\"\
         \$command \$nick;"
    pack $w.fileC.cLabel $w.fileC.cEntry -side left
    pack $w.fileC.nLabel -side left -pady 5 -padx 10
    pack $w.fileC.nEntry -side left -pady 5 -padx 5
    pack $w.fileC.add -side left -pady 5 -padx 5

    # Option Menu command panel

    frame $w.optF
    label $w.optF.label -text "View:" -width 12 -anchor w
    set viewOptMenu [tk_optionMenu $w.optF.optB curNick " "]
    button $w.optF.stop -text "Stop" -command Stop

    pack $w.optF.label -side left
    pack $w.optF.optB -side left -pady 5
    pack $w.optF.stop -side left -pady 5

    # create text widget
    frame $w.textf -bg red
    text $w.textf.text -height [expr $tailSize] -xscrollcommand \
       "$w.textf.scrollh set" -yscrollcommand "$w.textf.scrollv set" -bg lightblue
    set textw $w.textf.text
    scrollbar $w.textf.scrollh -orient horizontal -command "$w.textf.text xview"
    scrollbar $w.textf.scrollv -orient vertical -command "$w.textf.text yview"

    pack $w.textf.scrollv -side right -fill y -expand 1
    pack $w.textf.scrollh -side bottom -fill x -expand 1
    pack $w.textf.text -fill x -fill y -expand 1


    # pack all the frames
    [winfo toplevel $w.textf] configure -menu $w.menu
    pack $w.status -side bottom -fill x -pady 2m
    pack $w.file -side top -fill x -expand 1
    pack $w.fileC -side top -fill x -expand 1
    pack $w.optF -side top -fill x -expand 1
    pack $w.textf -side top -fill x -fill y -expand 1

}
```

Once the user sets up a command- or file-type nick, the `TailFile` method will get called. This method makes sure that the specified file exists. It creates the command and opens it as a process. Then it binds a read event to the file descriptor and returns. The read event will call `TailUpdate` every time the file identifier associated with the process is readable.

```
#  TailFile --
#
#       Show the tail of the request file.
#
#  Arguments:
#       file name to be tailed.
#
#  Results:
#       The tail of the file is shown in the window.
#

proc TailFile { type file {nick ""}} {
    global tailSize tailFd  textw curNick
    set w $textw
    catch {
        fileevent $tailFd readable {}
        close $tailFd
        update
    }
    $w delete 1.0 end

    if {$type == ""} {
        $w insert end "Illegal type...";
        return
    }
    if {$type == "file"} {
        if {$file == ""} {
            $w insert end "please specify a valid filename..."
            return
        }
        if ![file exists $file] {
            DeleteFromView $file
            $w insert end "file $file does not exist..."
            return
        }
        set nick $file
    } elseif {$type == "command"} {
        if {$file == ""} {
            $w insert end "please specify a command..."
            return
        }
    }

    if {$type == "file"} {
        set tailFd [open "|tail -f $file" r]
        wm title [winfo toplevel $w] "Tail tool \[tail -f $file\]"
    } elseif {$type == "command"} {
        if [catch {set tailFd [open "|$file" r]}] {
            SetStatus error  "can't execute command $file..."
            DeleteFromView $nick
            set curNick ""
            return
        }

        wm title [winfo toplevel $w] "Tail tool \[tail |$file\]"
    }
    fconfigure $tailFd -blocking 0
    set lines 0
    fileevent $tailFd readable "TailUpdate \$tailFd"
}
```

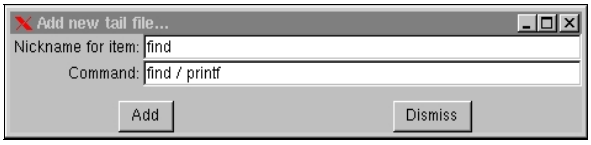
The `TailUpdate` procedure gets called as part of the event handler on the current log process read status. When this procedure gets called, it collects the output from the process and inserts it into the text widget. It also makes sure that at any given time, no more than `$tailSize` lines are shown in the text window.

```
proc TailUpdate {fileFd} {
   global textw curNick
   global tailSize tailFd

   set w $textw
   if [eof $tailFd] {
      fileevent $tailFd readable {}
      $w insert end "Tailing \"$curNick\" done..."
      return
   }

   set line [gets $tailFd]

   $w insert end $line
   $w insert end "\n"

   set lines [lindex [split [$w index end] .] 0]
   if {$lines == [expr $tailSize+1]} {
      $w delete 1.0 2.0
   }
   $w yview moveto 1.0

}
```

The `Stop` call-back is used to stop the current `logProcess`.

```
#
# Stop the current tailing process
#

proc Stop {} {
   global tailFd
   set pid [pid $tailFd]
   catch {exec kill -9 $pid}
}
```

`The AddNew` procedure gets called every time the user adds a new shortcut through the File menu's "Add New..." menu command. It creates a simple GUI for the user to create a new command nick:




```
#  AddNew --
#       Add a new tail file to the system
#
#  Arguments:
#       none.
#
#  Results:
#

proc AddNew {args} {
   toplevel .addnew
   set w .addnew
   wm title $w "Add new tail file..."

   frame  $w.top
```

```
   frame $w.sep -bd 2  -relief ridge
   frame $w.bot


   set k $w.top

   label $k.name -text "Nickname for item:"
   label $k.command -text "Command:"

   grid $k.name -row 0 -column 0 -sticky e
   grid $k.command -row 1 -column 0 -sticky e

   entry $k.nameE -textvariable nameE -width 40
   entry $k.commandE -textvariable commandE -width 50

   grid $k.nameE -row 0 -column 1 -sticky ew
   grid $k.commandE -row 1 -column 1 -sticky ew

   grid columnconfigure $k 1 -weight 1
   grid propagate $k 1

   pack $w.top  -side top -fill both -expand 1
   pack $w.sep -side top -fill x -expand 1 -pady 5
   pack $w.bot -side top -fill x -expand 1


   button $w.bot.apply -text "Add" -command "AddToView \"command\"  \"\$commandE\"
\"\$nameE\""
   button $w.bot.dismiss -text "Dismiss" -command {destroy .addnew}
   pack $w.bot.apply $w.bot.dismiss -side left -expand  1
   PositionWindow $w
}
```

The `SetStatus` procedure is used to set GUI status messages in the status window. It's a general-purpose routine to display both error and informational messages. If a type error occurs, an error bitmap will be displayed in the status window.

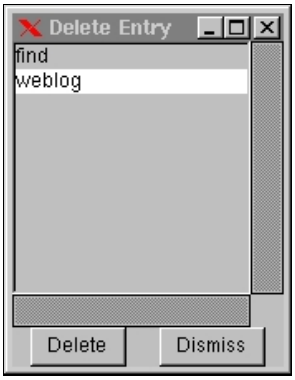
```
proc SetStatus {type text {timer 5000}} {
   global statusText  statusImgWin
   set statusText $text
   after $timer "set statusText \"\""
   $statusImgWin config -bitmap $type
   after $timer "$statusImgWin config -bitmap \"\""
}
```

The `AddToView` command will add a nick to the option button. Before it adds the item to the option menu, it makes sure that the user has supplied the required information.

```
proc AddToView  {type command {nick ""}}  {
   global tailOpts  viewOptMenu


    catch {Stop}
   if {$type == "file"} {
      set nick $command
      if {$command == ""} {
          SetStatus error "Please supply File name..."
      }
   } elseif  {$type == "command"} {
      if {($nick == "") || ($command == "") } {
         SetStatus error "Please supply both nick and command names..."
         return
      }
   }
   set l [list  "$type" "$nick" "$command"]
   if ![info exists tailOpts(wins)] {
      set tailOpts $l
```

```
      return
   } else {
      foreach item $tailOpts(wins) {
         if {$nick == [lindex $item 1]} {
            if {$type == "file"} {
               SetStatus info "File $nick is all ready in the tail list...."
            } else {
               SetStatus info "Nick $nick is all ready in the tail list...."
            }
            return;
         }
      }
      lappend tailOpts(wins) $l
   }

   UpdateOptionMenu
   $viewOptMenu invoke end
}
```

The `DeleteNicks` routine will create a simple listbox-based user interface for the user to delete the nicks:




```
proc DeleteNicks {} {
   global   tailOpts

   if ![info exists tailOpts(wins)] {
      SetStatus info "No entries to delete..."
      return;
   }
   if {$tailOpts(wins) == {}} {
      SetStatus info "No entries to delete..."
      return;
   }
    catch {destroy .delent}
   toplevel .delent
   set w .delent
   wm title $w "Delete Entry"

   scrollbar $w.h -orient horizontal -command "$w.list xview"
   scrollbar $w.v -orient vertical -command "$w.list yview"
   listbox $w.l -selectmode single -width 20 -height 10 \
        -setgrid 1 -xscroll "$w.h set" -yscroll "$w.v set"

    frame $w.buts
    button $w.buts.d -text "Delete" -command {
        set index [.delent.l curselection ];
        if {$index == ""} {return}
        set sel [.delent.l get $index ];
        puts "sel $sel ; index $index"
        DeleteFromView $sel;
        .delent.l  delete $index
    }
```

```
    button $w.buts.dismiss -text "Dismiss" -command "destroy $w"

    grid $w.l -row 0 -column 0 -columnspan 2 -sticky "news"
    grid $w.v -row 0 -column 2 -sticky "ns"
    grid $w.h -row 1 -column 0 -columnspan 2 -sticky "we"
    grid $w.buts -row 2 -column 0 -columnspan 3

    pack $w.buts.d $w.buts.dismiss -side left -padx 10

   foreach ent $tailOpts(wins) {
     $w.l insert end [lindex $ent 1]
   }

    PositionWindow $w
}
```

The `DeleteFromView` routine is an internal routine that removes the specified nick from the data structures and updates the option button:

```
proc DeleteFromView {nick} {
   global tailOpts

    if {$nick == ""} {
        return
    }
   set newList {}
   if ![info exists tailOpts(wins)] {
      return
   }
   foreach item $tailOpts(wins) {
      if {$nick != [lindex $item 1]} {
         lappend newList $item
      }
   }
   set tailOpts(wins) $newList
   UpdateOptionMenu
}
```

The `PositionWindow` routine centers a top-level dialog box around its parent. It is used to map dialog boxes on the main window, instead of some far-away corner of the screen.

```
#
# PositionWindow  --
#
#       Position the top-level window centered on its parent.
#
#  Arguments:
#       toplevel window.
#
#  Results:
#       Positions the window
#
proc PositionWindow {w} {
   set paren [winfo parent $w]
   wm iconify $w
   set parenConf [wm geometry $paren]
   set parenConf [split $parenConf {+ - x}]
   set winConf [split [wm geometry $w] {+ - x}]
   set X [expr [lindex $parenConf 2] + [lindex $parenConf 0]/2 - \
              [winfo reqwidth $w]/2]

   set Y [expr [lindex $parenConf 3] + [lindex $parenConf 1]/2 - \
              [winfo reqheigh $w]/2]
   wm geometry $w +$X+$Y
   wm deiconify $w
```

```
}
```

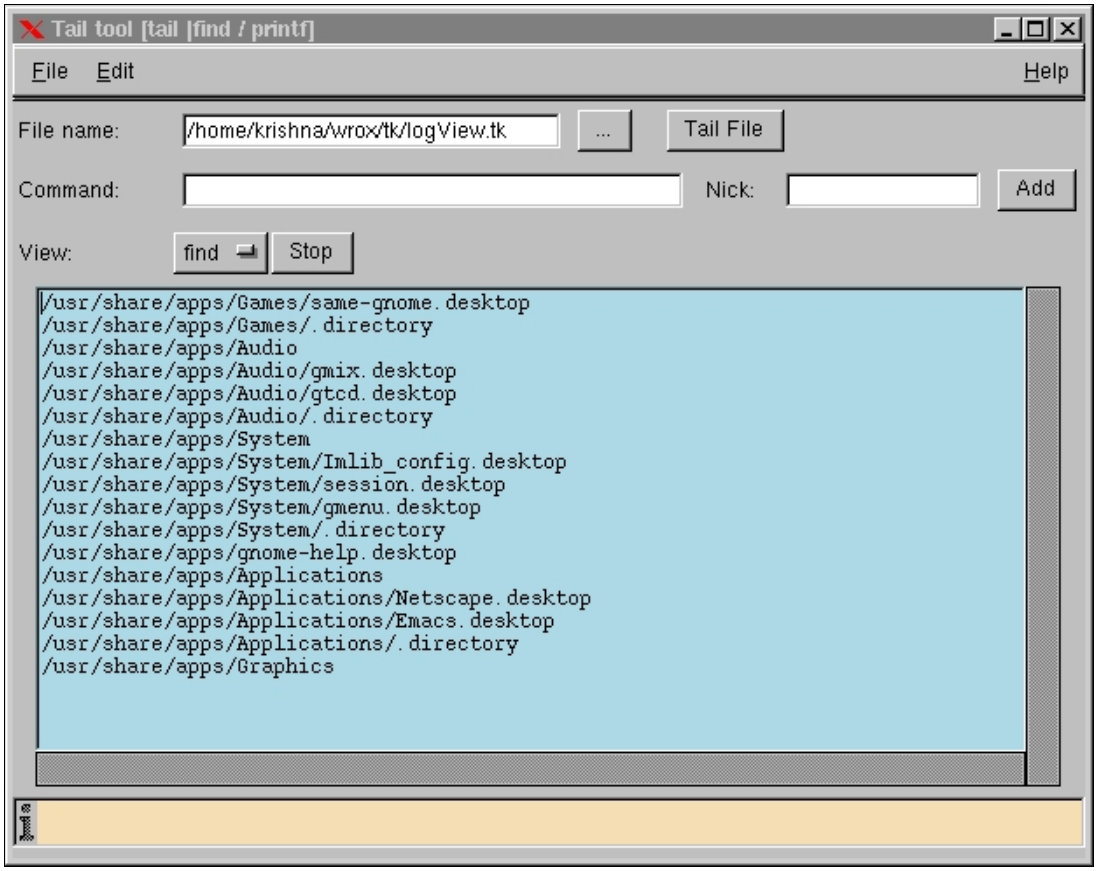
The `UpdateOptionMenu` command updates the nicks option menu widget with the current set of active nicks:

```
#  UpdateOptionMenu --
#
#
#
#  Arguments:
#
#
#  Results:
#

proc UpdateOptionMenu {} {
   global tailOpts curNick viewOptMenu

   $viewOptMenu delete 0 end
   if ![info exists tailOpts(wins)] {
      return
   }
   if {$tailOpts(wins) == {}} {
      set curNick ""
      return
   }
   foreach item $tailOpts(wins) {
       $viewOptMenu add command -label [lindex $item 1] -command "catch Stop; TailFile
\"[lindex $item 0]\" \"[lindex $item 2]\" \"[lindex $item 1]\" "
      set curNick [lindex $item 1]
   }
}
```

Finally, we map the main window:

```
wm withdraw .

toplevel .t

BuildTailGui .t
```

When you run this application, you should see the following:



Internationalization

One thing we have not covered in this survey of Tk widgets is internationalization. Tcl 8.1 has lots of new features such as Unicode support, functions to create and access message catalogs (so you can store the text of all your dialog boxes in multiple languages), support for different language encodings, and generalized string manipulation. Beginning in Tcl 8.1, all Tcl string manipulation functions expect and return Unicode strings encoded in UTF-8 format. Because the use of Unicode/UTF-8 encoding is internal to Tcl, you should see no difference in Tcl 8.0 and 8.1 string handling in your scripts. In fact, all the commands in Tcl 8.1 onward handle Unicode seamlessly. For example, you can read a file that uses `shiftjis` encoding into Tcl, and the file read command converts the `shiftjis` encoding to UTF-8 encoding automatically.

```
set fd [open $file r]
fconfigure $fd -encoding shiftjis
set jstring [read $fd] close $fd
close $fd
```

Furthermore, the regular expression implementation introduced in Tcl 8.1 handles the full range of Unicode characters.

Since all strings in Tcl are encoded in Unicode, Tk widgets automatically handle any encoding conversions necessary to display the characters in a particular font. For example, the code snippet

```
set str "\u592a\u9177"
??
% button .b -text $str
.b
% pack .b
```

should display the Chinese transliteration of "Tcl" (*TAI-KU*) as the button label, provided you have the correct X fonts installed to display this text. If the master font that you set for a widget doesn't contain a glyph for a particular Unicode character that you want to display, Tk attempts to locate a font that does. Tk attempts to locate a font that matches as many characteristics of the widget's master font as possible (weight, slant, etc.). Once Tk finds a suitable font, it displays the character in that font. In other words, the widget uses the master font for all characters it is capable of displaying, and alternative fonts only as needed.

Internationalization is a fascinating topic. Unfortunately it requires more space than we have in this chapter.

Where Now?

If you ever get stuck in Tk, in addition to the `man` pages, there's always the Tk Widget Tour with its examples of how to use the Tk widget set. Run the program by typing `widget`.

Some notable programs written in Tk include `Xadmin`, `Exmh`, `ical`, `TkMan`, `TkElm`, `TkWWW`, and `SurfIT`. `tkWWW` is an HTML editor, so you can use it to prepare pages for the World Wide Web, and `SurfIT` is a Web browser written in Tcl that has the distinction of being able to download and execute Tcl programs directly from Web pages. This, of course, can be a dangerous facility to allow with unknown hosts! `Ical` is an X-based calendar program

At the time of writing, there are many things happening within the Tk community, so we'll finish off with a brief survey of some notable projects.

Tix

Tix extends Tk with over 40 professional Motif look-and-feel widgets. Tix widgets are so powerful that they even give Motif 2.0 a run for its money. Check out Tix at http://tix.sourceforge.net

[incr Tk]

[incr Tcl] and [incr Tk] form an object-oriented extension to Tcl/Tk. Version 3.0 has recently been announced and is available at http://www.tcltk.com/itcl/. The following description is from their Web pages:

"[incr Tcl] provides the extra language support needed to build large Tcl/Tk applications. It introduces the notion of objects, which act as building blocks for an application. Each object is a bag of data with a set of procedures or 'methods' that are used to manipulate it. Objects are organized into 'classes' with identical characteristics, and classes can inherit functionality from one another. This object-oriented paradigm adds another level of organization on top of the basic variable/procedure elements, and the resulting code is easier to understand and maintain.

[incr Tk] is a framework for building 'mega-widgets' using the [incr Tcl] object system. Mega-widgets are high-level widgets like a file browser or a tab notebook that act like simple Tk widgets but are themselves constructed using Tk widgets as component parts, without having to write any C code. In effect, a mega-widget looks and acts exactly like a Tk widget, but is considerably easier to implement."

BLT

The BLT-2.1 toolkit extends Tk by providing many new widgets—for example, `blt_graph`, which is used to plot line and bar graphs, and `blt_htext`, a hypertext widget, and widgets for background execution. The BLT homepage is http://www.tcltk.com/blt.

Finally, comp.lang.tcl and comp.lang.tcl.announce are the best places to post Tk questions. Usually, people are quite friendly and somebody will always answer your questions. Before posting your questions to comp.lang.tcl, though, please read its frequently asked questions (FAQ) list, which is posted regularly to the newsgroup.

Lots of Tk resources can be found at Tcl's new home, the Scriptics Corporation Web page: http://www.scriptics.com/resource.

That about completes our survey of Tk. There is far more to Tk than this brief survey can do justice to, as John Ousterhout intended Tcl/Tk to be an extensible and embeddable tool. Tcl commands and Tk widgets are written in C, and you can code your own and add them in, or take advantage of the widget extensions available on the Internet. This is an advanced topic and space prevents us from covering it here, but it is the way in which you would access a C program from your graphical Tcl/Tk front end. Please refer to John Ousterhout's book for details and, in the meantime, happy Tcl/Tk'ing!

Summary

In this chapter, we rushed through the world of X Windows programming.

After an overview of the thinking behind X Windows and the different ways in which this was implemented, we learned enough Tk to complement the Tcl that we learned in Chapter 15 and enable us to rapidly develop GUI front ends to our applications using Tk's rich widget set.

Next we're going to look at an exciting new way to program graphical applications in C: the GTK+ GNOME toolkit.