CHAPTER 22

Tk by Example

This chapter introduces Tk through a series of short examples. The ExecLog runs a program in the background and displays its output. The Example Browser displays the Tcl examples from the book. The Tcl Shell lets you type Tcl commands and execute them in a slave interpreter.

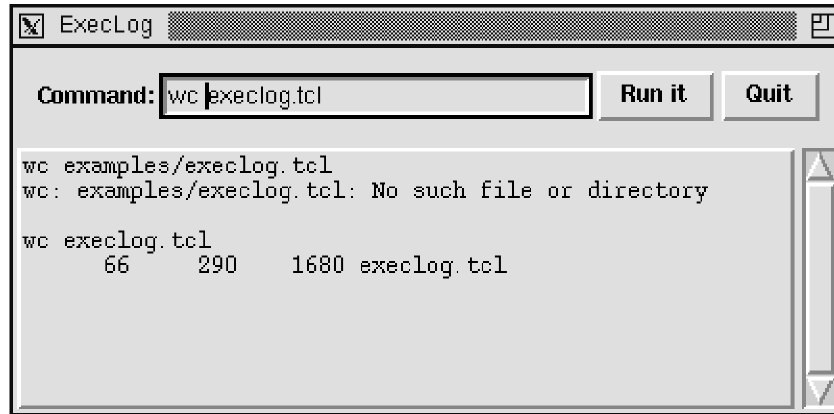
This chapter is from *Practical Programming in Tcl and Tk*, 3rd Ed.
© 1999, Brent Welch
http://www.beedub.com/book/

Tk provides a quick and fun way to generate user interfaces. In this chapter we will go through a series of short example programs to give you a feel for what you can do. Some details are glossed over in this chapter and considered in more detail later. In particular, the `pack` geometry manager is covered in Chapter 23 and event bindings are discussed in Chapter 26. The Tk widgets are discussed in more detail in later chapters.

ExecLog

Our first example provides a simple user interface to running another program with the `exec` command. The interface consists of two buttons, `Run it` and `Quit`, an entry widget in which to enter a command, and a text widget in which to log the results of running the program. The script runs the program in a pipeline and uses the `fileevent` command to wait for output. This structure lets the user interface remain responsive while the program executes. You could use this to run *make*, for example, and it would save the results in the log. The complete example is given first, and then its commands are discussed in more detail.

Example 22–1 Logging the output of a program run with `exec`.




```
#!/usr/local/bin/wish
# execlog - run a program with exec and log the output
# Set window title
wm title . ExecLog

# Create a frame for buttons and entry.

frame .top -borderwidth 10
pack .top -side top -fill x

# Create the command buttons.

button .top.quit -text Quit -command exit
set but [button .top.run -text "Run it" -command Run]
pack .top.quit .top.run -side right

# Create a labeled entry for the command

label .top.l -text Command: -padx 0
entry .top.cmd -width 20 -relief sunken \
   -textvariable command
pack .top.l -side left
pack .top.cmd -side left -fill x -expand true

# Set up key binding equivalents to the buttons

bind .top.cmd <Return> Run
bind .top.cmd <Control-c> Stop
focus .top.cmd

# Create a text widget to log the output

frame .t
set log [text .t.log -width 80 -height 10 \
   -borderwidth 2 -relief raised -setgrid true \
```

```
    -yscrollcommand {.t.scroll set}]
scrollbar .t.scroll -command {.t.log yview}
pack .t.scroll -side right -fill y
pack .t.log -side left -fill both -expand true
pack .t -side top -fill both -expand true

# Run the program and arrange to read its input

proc Run {} {
    global command input log but
    if [catch {open "|$command |& cat"} input] {
        $log insert end $input\n
    } else {
        fileevent $input readable Log
        $log insert end $command\n
        $but config -text Stop -command Stop
    }
}

# Read and log output from the program

proc Log {} {
    global input log
    if [eof $input] {
        Stop
    } else {
        gets $input line
        $log insert end $line\n
        $log see end
    }
}

# Stop the program and fix up the button

proc Stop {} {
    global input but
    catch {close $input}
    $but config -text "Run it" -command Run
}
```


Window Title

The first command sets the title that appears in the title bar implemented by the window manager. Recall that dot (i.e., `.`) is the name of the main window:

```
wm title . ExecLog
```

The `wm` command communicates with the window manager. The window manager is the program that lets you open, close, and resize windows. It implements the title bar for the window and probably some small buttons to close or resize the window. Different window managers have a distinctive look; the figure shows a title bar from *twm*, a window manager for X.

A Frame for Buttons

A frame is created to hold the widgets that appear along the top of the interface. The frame has a border to provide some space around the widgets:

```
frame .top -borderwidth 10
```

The frame is positioned in the main window. The default packing side is the top, so `-side top` is redundant here, but it is used for clarity. The `-fill x` packing option makes the frame fill out to the whole width of the main window:

```
pack .top -side top -fill x
```

Command Buttons

Two buttons are created: one to run the command, the other to quit the program. Their names, `.top.quit` and `.top.run`, imply that they are children of the `.top` frame. This affects the `pack` command, which positions widgets inside their parent by default:

```
button .top.quit -text Quit -command exit
set but [button .top.run -text "Run it" \
   -command Run]
pack .top.quit .top.run -side right
```

A Label and an Entry

The label and entry are also created as children of the `.top` frame. The label is created with no padding in the X direction so that it can be positioned right next to the entry. The size of the entry is specified in terms of characters. The `relief` attribute gives the entry some looks to set it apart visually on the display. The contents of the entry widget are linked to the Tcl variable `command`:

```
label .top.l -text Command: -padx 0
entry .top.cmd -width 20 -relief sunken \
   -textvariable command
```

The label and entry are positioned to the left inside the `.top` frame. The additional packing parameters to the entry allow it to expand its packing space and fill up that extra area with its display. The difference between packing space and display space is discussed in Chapter 23 on page 337:

```
pack .top.l -side left
pack .top.cmd -side left -fill x -expand true
```

Key Bindings and Focus

Key bindings on the entry widget provide an additional way to invoke the functions of the application. The `bind` command associates a Tcl command with an event in a particular widget. The `<Return>` event is generated when the user presses the `Return` key on the keyboard. The `<Control-c>` event is generated when the letter `c` is typed while the `Control` key is already held down. For the

events to go to the entry widget, .top.cmd, input focus must be given to the widget. By default, an entry widget gets the focus when you click the left mouse button in it. The explicit focus command is helpful for users with the focus-follows-mouse model. As soon as the mouse is over the main window the user can type into the entry:

```
bind .top.cmd <Return> Run
bind .top.cmd <Control-c> Stop
focus .top.cmd
```

A Resizable Text and Scrollbar

A text widget is created and packed into a frame with a scrollbar. The width and height of the text widget are specified in characters and lines, respectively. The setgrid attribute of the text widget is turned on. This restricts the resize so that only a whole number of lines and average-sized characters can be displayed.

The scrollbar is a separate widget in Tk, and it can be connected to different widgets using the same setup as is used here. The text's yscrollcommand updates the display of the scrollbar when the text widget is modified, and the scrollbar's command scrolls the associated widget when the user manipulates the scrollbar:

```
frame .t
set log [text .t.log -width 80 -height 10 \
   -borderwidth 2 -relief raised -setgrid true\
   -yscrollcommand {.t.scroll set}]
scrollbar .t.scroll -command {.t.log yview}
pack .t.scroll -side right -fill y
pack .t.log -side left -fill both -expand true
pack .t -side top -fill both -expand true
```

A side effect of creating a Tk widget is the creation of a new Tcl command that operates on that widget. The name of the Tcl command is the same as the Tk pathname of the widget. In this script, the text widget command, .t.log, is needed in several places. However, it is a good idea to put the Tk pathname of an important widget into a variable because that pathname can change if you reorganize your user interface. The disadvantage of this is that you must declare the variable with global inside procedures. The variable log is used for this purpose in this example to demonstrate this style.

The Run Procedure

The Run procedure starts the program specified in the command entry. That value is available in the global command variable because of the textvariable attribute of the entry. The command is run in a pipeline so that it executes in the background. The leading | in the argument to open indicates that a pipeline is being created. The catch command guards against bogus commands. The variable input is set to an error message, or to the normal open return that is a file

descriptor. The program is started like this:

```
if [catch {open "|$command |& cat"} input] {
```

Trapping errors from pipelines.

The pipeline diverts error output from the command through the *cat* program. If you do not use *cat* like this, then the error output from the pipeline, if any, shows up as an error message when the pipeline is closed. In this example it turns out to be awkward to distinguish between errors generated from the program and errors generated because of the way the `Stop` procedure is implemented. Furthermore, some programs interleave output and error output, and you might want to see the error output in order instead of all at the end.

If the pipeline is opened successfully, then a callback is set up using the `fileevent` command. Whenever the pipeline generates output, then the script can read data from it. The `Log` procedure is registered to be called whenever the pipeline is readable:

```
fileevent $input readable Log
```

The `command` (or the error message) is inserted into the log. This is done using the name of the text widget, which is stored in the `log` variable, as a Tcl command. The value of the command is appended to the log, and a newline is added so that its output will appear on the next line.

```
$log insert end $command\n
```

The text widget's `insert` function takes two parameters: a *mark* and a string to insert at that mark. The symbolic mark `end` represents the end of the contents of the text widget.

The run button is changed into a stop button after the program begins. This avoids a cluttered interface and demonstrates the dynamic nature of a Tk interface. Again, because this button is used in a few different places in the script, its pathname has been stored in the variable `but`:

```
$but config -text Stop -command Stop
```

The Log Procedure

The `Log` procedure is invoked whenever data can be read from the pipeline, and when end of file has been reached. This condition is checked first, and the `Stop` procedure is called to clean things up. Otherwise, one line of data is read and inserted into the log. The text widget's `see` operation is used to position the view on the text so that the new line is visible to the user:

```
if [eof $input] {
   Stop
} else {
   gets $input line
   $log insert end $line\n
   $log see end
}
```

The Stop Procedure

The `Stop` procedure terminates the program by closing the pipeline. The `close` is wrapped up with a `catch`. This suppresses the errors that can occur when the pipeline is closed prematurely on the process. Finally, the button is restored to its run state so that the user can run another command:

```
catch {close $input}
$but config -text "Run it" -command Run
```

In most cases, closing the pipeline is adequate to kill the job. On UNIX, this results in a signal, `SIGPIPE`, being delivered to the program the next time it does a write to its standard output. There is no built-in way to kill a process, but you can `exec` the UNIX *kill* program. The `pid` command returns the process IDs from the pipeline:

```
foreach pid [pid $input] {
    catch {exec kill $pid}
}
```

If you need more sophisticated control over another process, you should check out the *expect* Tcl extension, which is described in the book *Exploring Expect* (Don Libes, O'Reilly & Associates, Inc., 1995). *Expect* provides powerful control over interactive programs. You can write Tcl scripts that send input to interactive programs and pattern match on their output. *Expect* is designed to automate the use of programs that were designed for interactive use.

Cross-Platform Issues

This script will run on UNIX and Windows, but not on Macintosh because there is no `exec` command. One other problem is the binding for `<Control-c>` to cancel the job. This is UNIX-like, while Windows users expect `<Escape>` to cancel a job, and Macintosh users expect `<Command-period>`. `Platform_CancelEvent` defines a virtual event, `<<Cancel>>`, and `Stop` is bound to it:

Example 22–2 A platform-specific cancel event.

```
proc Platform_CancelEvent {} {
    global tcl_platform
    switch $tcl_platform(platform) {
        unix {
            event add <<Cancel>> <Control-c>
        }
        windows {
            event add <<Cancel>> <Escape>
        }
        macintosh {
            event add <<Cancel>> <Command-period>
        }
    }
}
bind .top.entry <<Cancel>> Stop
```

III. Tk Basics

There are other virtual events already defined by Tk. The `event` command and virtual events are described on page 378.

The Example Browser

Example 22–3 is a browser for the code examples that appear in this book. The basic idea is to provide a menu that selects the examples, and a text window to display the examples. Before you can use this sample program, you need to edit it to set the proper location of the `exsource` directory that contains all the example sources from the book. Example 22–4 on page 327 extends the browser with a shell that is used to test the examples.

Example 22–3 A browser for the code examples in the book.

```
#!/usr/local/bin/wish
#   Browser for the Tcl and Tk examples in the book.

# browse(dir) is the directory containing all the tcl files
# Please edit to match your system configuration.

switch $tcl_platform(platform) {
    "unix" {set browse(dir) /cdrom/tclbook2/exsource}
    "windows" {set browse(dir) D:/exsource}
    "macintosh" {set browse(dir) /tclbook2/exsource}
}

wm minsize . 30 5
wm title . "Tcl Example Browser"

# Create a row of buttons along the top

set f [frame .menubar]
pack $f -fill x
button $f.quit -text Quit -command exit
button $f.next -text Next -command Next
button $f.prev -text Previous -command Previous

# The Run and Reset buttons use EvalEcho that
# is defined by the Tcl shell in Example 22-4 on page 327

button $f.load -text Run -command Run
button $f.reset -text Reset -command Reset
pack $f.quit $f.reset $f.load $f.next $f.prev -side right

# A label identifies the current example

label $f.label -textvariable browse(current)
pack $f.label -side right -fill x -expand true

# Create the menubutton and menu
```

```
menubutton $f.ex -text Examples -menu $f.ex.m
pack $f.ex -side left
set m [menu $f.ex.m]

# Create the text to display the example
# Scrolled_Text is defined in Example 30-1 on page 428

set browse(text) [Scrolled_Text .body \
    -width 80 -height 10\
    -setgrid true]
pack .body -fill both -expand true

# Look through the example files for their ID number.

foreach f [lsort -dictionary [glob [file join $browse(dir) *]]] {
    if [catch {open $f} in] {
        puts stderr "Cannot open $f: $in"
        continue
    }
    while {[gets $in line] >= 0} {
        if [regexp {^# Example ([0-9]+)-([0-9]+)} $line \
                x chap ex] {
            lappend examples($chap) $ex
            lappend browse(list) $f
            # Read example title
            gets $in line
            set title($chap-$ex) [string trim $line "# "]
            set file($chap-$ex) $f
            close $in
            break
        }
    }
}

# Create two levels of cascaded menus.
# The first level divides up the chapters into chunks.
# The second level has an entry for each example.

option add *Menu.tearOff 0
set limit 8
set c 0; set i 0
foreach chap [lsort -integer [array names examples]] {
    if {$i == 0} {
        $m add cascade -label "Chapter $chap..." \
            -menu $m.$c
        set sub1 [menu $m.$c]
        incr c
    }
    set i [expr ($i +1) % $limit]
    $sub1 add cascade -label "Chapter $chap" -menu $sub1.sub$i
    set sub2 [menu $sub1.sub$i ]
    foreach ex [lsort -integer $examples($chap)] {
        $sub2 add command -label "$chap-$ex $title($chap-$ex)" \
            -command [list Browse $file($chap-$ex)]
```

```
    }
}

# Display a specified file. The label is updated to
# reflect what is displayed, and the text is left
# in a read-only mode after the example is inserted.

proc Browse { file } {
    global browse
    set browse(current) [file tail $file]
    set browse(curix) [lsearch $browse(list) $file]
    set t $browse(text)
    $t config -state normal
    $t delete 1.0 end
    if [catch {open $file} in] {
        $t insert end $in
    } else {
        $t insert end [read $in]
        close $in
    }
    $t config -state disabled
}

# Browse the next and previous files in the list

set browse(curix) -1
proc Next {} {
    global browse
    if {$browse(curix) < [llength $browse(list)] - 1} {
        incr browse(curix)
    }
    Browse [lindex $browse(list) $browse(curix)]
}
proc Previous {} {
    global browse
    if {$browse(curix) > 0} {
        incr browse(curix) -1
    }
    Browse [lindex $browse(list) $browse(curix)]
}

# Run the example in the shell

proc Run {} {
    global browse
    EvalEcho [list source \
        [file join $browse(dir) $browse(current)]]
}

# Reset the slave in the eval server

proc Reset {} {
    EvalEcho reset
}
```

More about Resizing Windows

This example uses the `wm minsize` command to put a constraint on the minimum size of the window. The arguments specify the minimum width and height. These values can be interpreted in two ways. By default they are pixel values. However, if an internal widget has enabled *geometry gridding*, then the dimensions are in grid units of that widget. In this case the text widget enables gridding with its `setgrid` attribute, so the minimum size of the window is set so that the text window is at least 30 characters wide by five lines high:

```
wm minsize . 30 5
```

In older versions of Tk, Tk 3.6, gridding also enabled interactive resizing of the window. Interactive resizing is enabled by default in Tk 4.0 and later.

Managing Global State

The example uses the `browse` array to collect its global variables. This makes it simpler to reference the state from inside procedures because only the array needs to be declared global. As the application grows over time and new features are added, that `global` command won't have to be adjusted. This style also serves to emphasize what variables are important. The `browse` array holds the name of the example directory (`dir`), the Tk pathname of the text display (`text`), and the name of the current file (`current`). The `list` and `curix` elements are used to implement the `Next` and `Previous` procedures.

Searching through Files

The browser searches the file system to determine what it can display. The `tcl_platform(platform)` variable is used to select a different example directory on different platforms. You may need to edit the on-line example to match your system. The example uses `glob` to find all the files in the `exsource` directory. The `file join` command is used to create the file name pattern in a platform-independent way. The result of `glob` is sorted explicitly so the menu entries are in the right order. Each file is read one line at a time with `gets`, and then `regexp` is used to scan for keywords. The loop is repeated here for reference:

```
foreach f [lsort -dictionary [glob [file join $browse(dir) *]]] {
    if [catch {open $f} in] {
        puts stderr "Cannot open $f: $in"
        continue
    }
    while {[gets $in line] >= 0} {
        if [regexp {^# Example ([0-9]+)-([0-9]+)} $line \
                x chap ex] {
            lappend examples($chap) $ex
            lappend browse(list) $f
            # Read example title
            gets $in line
            set title($chap-$ex) [string trim $line "# "]
            set file($chap-$ex) $f
```

```
            close $in
            break
        }
    }
}
```

The example files contain lines like this:

```
# Example 1-1
# The Hello, World! program
```

The `regexp` picks out the example numbers with the `([0-9]+)-([0-9]+)` part of the pattern, and these are assigned to the `chap` and `ex` variables. The `x` variable is assigned the value of the whole match, which is more than we are interested in. Once the example number is found, the next line is read to get the description of the example. At the end of the `foreach` loop the `examples` array has an element defined for each chapter, and the value of each element is a list of the examples for that chapter.

Cascaded Menus

The values in the `examples` array are used to build up a cascaded menu structure. First a menubutton is created that will post the main menu. It is associated with the main menu with its `menu` attribute. The menu must be a child of the menubutton for its display to work properly:

```
menubutton $f.ex -text Examples -menu $f.ex.m
set m [menu $f.ex.m]
```

There are too many chapters to put them all into one menu. The main menu has a `cascade` entry for each group of eight chapters. Each of these submenus has a `cascade` entry for each chapter in the group, and each chapter has a menu of all its examples. Once again, the submenus are defined as a child of their parent menu. Note the inconsistency between menu entries and buttons. Their text is defined with the `-label` option, not `-text`. Other than this they are much like buttons. Chapter 27 describes menus in more detail. The code is repeated here:

```
set limit 8 ; set c 0 ; set i 0
foreach key [lsort -integer [array names examples]] {
    if {$i == 0} {
        $m add cascade -label "Chapter $key..." \
            -menu $m.$c
        set sub1 [menu $m.$c]
        incr c
    }
    set i [expr ($i +1) % $limit]
    $sub1 add cascade -label "Chapter $key" -menu $sub1.sub$i
    set sub2 [menu $sub1.sub$i]
    foreach ex [lsort -integer $examples($key)] {
        $sub2 add command -label "$key-$ex $title($key-$ex)" \
            -command [list Browse $file($key-$ex)]
    }
}
```

A Read-Only Text Widget

The `Browse` procedure is fairly simple. It sets `browse(current)` to be the name of the file. This changes the main label because of its `textvariable` attribute that links it to this variable. The `state` attribute of the text widget is manipulated so that the text is read-only after the text is inserted. You have to set the `state` to `normal` before inserting the text; otherwise, the `insert` has no effect. Here are a few commands from the body of `Browse`:

```
global browse
set browse(current) [file tail $file]
$t config -state normal
$t insert end [read $in]
$t config -state disabled
```

A Tcl Shell

This section demonstrates the text widget with a simple Tcl shell application. It uses a text widget to prompt for commands and display their results. It uses a second Tcl interpreter to evaluate the commands you type. This dual interpreter structure is used by the console built into the Windows and Macintosh versions of *wish*. The *TkCon* application written by Jeff Hobbs is an even more elaborate console that has many features to support interactive Tcl use.

Example 22–4 is written to be used with the browser from Example 22–3 in the same application. The browser's `Run` button runs the current example in the shell. An alternative is to have the shell run as a separate process and use the `send` command to communicate Tcl commands between separate applications. That alternative is shown in Example 40–2 on page 563.

Example 22–4 A Tcl shell in a text widget.

```
#!/usr/local/bin/wish
# Simple evaluator. It executes Tcl in a slave interpreter

set t [Scrolled_Text .eval -width 80 -height 10]
pack .eval -fill both -expand true

# Text tags give script output, command errors, command
# results, and the prompt a different appearance

$t tag configure prompt -underline true
$t tag configure result -foreground purple
$t tag configure error -foreground red
$t tag configure output -foreground blue

# Insert the prompt and initialize the limit mark

set eval(prompt) "tcl> "
$t insert insert $eval(prompt) prompt
$t mark set limit insert
```

III. Tk Basics

```
$t mark gravity limit left
focus $t
set eval(text) $t

# Key bindings that limit input and eval things. The break in
# the bindings skips the default Text binding for the event.

bind $t <Return> {EvalTypein ; break}
bind $t <BackSpace> {
    if {[%W tag nextrange sel 1.0 end] != ""} {
        %W delete sel.first sel.last
    } elseif {[%W compare insert > limit]} {
        %W delete insert-1c
        %W see insert
    }
    break
}
bind $t <Key> {
    if [%W compare insert < limit] {
        %W mark set insert end
    }
}

# Evaluate everything between limit and end as a Tcl command

proc EvalTypein {} {
    global eval
    $eval(text) insert insert \n
    set command [$eval(text) get limit end]
    if [info complete $command] {
        $eval(text) mark set limit insert
        Eval $command
    }
}

# Echo the command and evaluate it

proc EvalEcho {command} {
    global eval
    $eval(text) mark set insert end
    $eval(text) insert insert $command\n
    Eval $command
}

# Evaluate a command and display its result

proc Eval {command} {
    global eval
    $eval(text) mark set insert end
    if [catch {$eval(slave) eval $command} result] {
        $eval(text) insert insert $result error
    } else {
        $eval(text) insert insert $result result
    }
    if {[$eval(text) compare insert != "insert linestart"]} {
```

```
        $eval(text) insert insert \n
    }
    $eval(text) insert insert $eval(prompt) prompt
    $eval(text) see insert
    $eval(text) mark set limit insert
    return
}

# Create and initialize the slave interpreter

proc SlaveInit {slave} {
    interp create $slave
    load {} Tk $slave
    interp alias $slave reset {} ResetAlias $slave
    interp alias $slave puts {} PutsAlias $slave
    return $slave
}

# The reset alias deletes the slave and starts a new one

proc ResetAlias {slave} {
    interp delete $slave
    SlaveInit $slave
}

# The puts alias puts stdout and stderr into the text widget

proc PutsAlias {slave args} {
    if {[llength $args] > 3} {
        error "invalid arguments"
    }
    set newline "\n"
    if {[string match "-nonewline" [lindex $args 0]]} {
        set newline ""
        set args [lreplace $args 0 0]
    }
    if {[llength $args] == 1} {
        set chan stdout
        set string [lindex $args 0]$newline
    } else {
        set chan [lindex $args 0]
        set string [lindex $args 1]$newline
    }
    if [regexp (stdout|stderr) $chan] {
        global eval
        $eval(text) mark gravity limit right
        $eval(text) insert limit $string output
        $eval(text) see limit
        $eval(text) mark gravity limit left
    } else {
        puts -nonewline $chan $string
    }
}
set eval(slave) [SlaveInit shell]
```

Text Marks, Tags, and Bindings

The shell uses a text *mark* and some extra bindings to ensure that users only type new text into the end of the text widget. A mark represents a position in the text that is updated as characters are inserted and deleted. The `limit` mark keeps track of the boundary between the read-only area and the editable area. The `insert` mark is where the cursor is displayed. The `end` mark is always the end of the text. The `EvalTypein` procedure looks at all the text between `limit` and `end` to see if it is a complete Tcl command. If it is, it evaluates the command in the slave interpreter.

The `<Key>` binding checks to see where the `insert` mark is and bounces it to the `end` if the user tries to input text before the `limit` mark. The `puts` alias sets `right` gravity on `limit`, so the mark is pushed along when program output is inserted right at `limit`. Otherwise, the `left` gravity on limit means that the mark does not move when the user inserts right at `limit`.

Text *tags* are used to give different regions of text difference appearances. A tag applies to a range of text. The tags are configured at the beginning of the script and they are applied when text is inserted.

Chapter 33 describes the text widget in more detail.

Multiple Interpreters

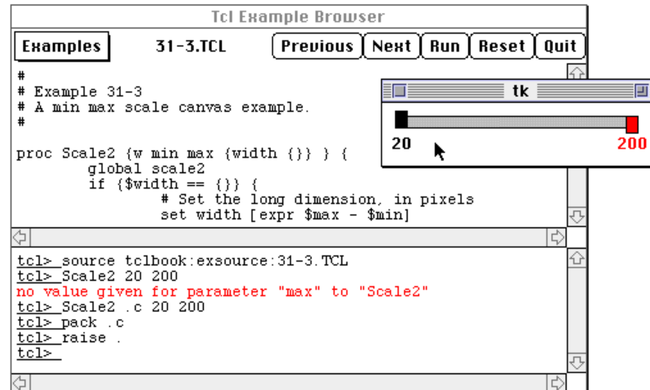
The `SlaveInit` procedure creates another interpreter to evaluate the commands. This prevents conflicts with the procedures and variables used to implement the shell. Initially, the slave interpreter only has access to Tcl commands. The `load` command installs the Tk commands, and it creates a new top-level window that is "." for the slave interpreter. Chapter 20 describes how you can embed the window of the slave within other frames.

The `shell` interpreter is not created with the `-safe` flag, so it can do anything. For example, if you type `exit`, it will exit the whole application. The `SlaveInit` procedure installs an alias, `reset`, that just deletes the slave interpreter and creates a new one. You can use this to clean up after working in the shell for a while. Chapter 19 describes the `interp` command in detail.

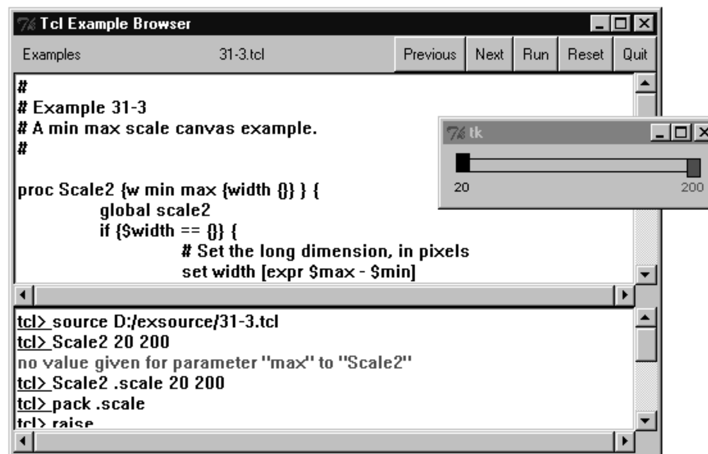
Native Look and Feel

When you run a Tk script on different platforms, it uses native buttons, menus, and scrollbars. The text and entry widgets are tuned to give the application the native look and feel. The following screen shots show the combined browser and shell as it looks on Macintosh, Windows, and UNIX.

Example 22–5 Macintosh look and feel.



Example 22–6 Windows look and feel.



III. Tk Basics

Example 22–7 UNIX look and feel.