

## **Part I**

# **Tcl Tk**



## Tcl Tk: Overview

Tcl (“tool command language”) is a scripting language and interpreter of that language. Originally developed in the late 80s by John Ousterhout as a “glue” to combine two or more complicated applications together, it evolved overtime to find use not just as middleware, but also as a standalone development tool.

Tk<sup>1</sup> is an extension of Tcl that provides GUI components through Tcl. This was first developed in 1990, again by John Ousterhout. Tk quickly found widespread usage, as it made programming GUIs for X11 easier and faster. Over the years, other graphical toolkits have evolved and surpassed this one, but Tk still has numerous users.

Tk has a large number of bindings available for it, e.g. Perl, Python, Ruby, and through the `tcltk` package, R. The `tcltk` package was developed by Peter Dalgaard, and included in R from version 1.1.0. Since then, the package has been used in a number of GUI projects for R, most notably, the `Rcmdr` GUI.

Tk had a major change between version 8.4 and 8.5, with the latter introducing themed widgets. Many widgets were rewritten, and their API dramatically simplified. In `tcltk` there can be two different functions to construct a similar widget. For example, `tklabel` or `ttklabel`. The latter, with the `ttk` prefix, would be for the newer themed widget. We assume the Tk version is 8.5 or higher, as this was a major step forward. As of version 2.7.0, R for windows has been bundled with this Tk version, so there are no installation issues for that platform. As of writing, some linux distributions and Mac OS X still come with 8.4 which would need to be upgraded for the following.

---

<sup>1</sup> Tk has a well documented API (?) ([www.tcl.tk/man/tcl8.5](http://www.tcl.tk/man/tcl8.5)). There are also several books to supplement. We consulted the one by Welch, Jones and Hobbs (?) often in the development of this material. In addition, the Tk Tutorial of Mark Roseman (?) ([www.tkdocs.com/tutorial](http://www.tkdocs.com/tutorial)) provides much detail. R specific documentation include two excellent R News articles and a proceedings paper (?), (?) and (?) by Peter Dalgaard, the package author. A set of examples due to James Wettenhall (?) are also quite instructive. A main use of `tcltk` is within the `Rcmdr` framework. Writing extensions for that is well documented in an R News article (?) by John Fox, the package author.

### 1.1 Interacting with Tcl

The basic syntax of Tcl is a bit unlike R. For example a simple string assignment would be made at `tclsh`, the Tcl shell with (using `%` as a prompt)

```
% set x {hello world}
hello world
```

Unlike R where braces are used to form blocks, this example shows how Tcl uses braces instead of quotes to group the words as a single string. The use of braces, instead of quotes, in this example is optional, but in general isn't, as expressions within braces are not evaluated.

The example above assigns to the variable `x` the value of `hello world`. Once assignment has been made, one can call commands on the value stored in `x` using the `$` prefix:

```
% puts $x
hello world
```

The `puts` command, in this usage, simply writes back its argument to the terminal. Had we used braces the argument would not have been substituted:

```
% puts {$x}
$x
```

More typical within the `tcltk` package is the idea of a subcommand. For example, the `string` command provides the subcommand `length` to return the number of characters in the string.

```
% string length $x
11
```

The `tcltk` package provides the low-level function `.Tcl` for direct access to the Tcl interpreter:

```
library(tcltk)
.Tcl("set x {some text}")           # assignment
```

```
<Tcl> some text
```

```
.Tcl("puts $x")                     # print
.Tcl("string length $x")            # call a command
```

```
<Tcl> 9
```

the `.Tcl` function simply sends a command as a text string to the Tcl interpreter and returns the result as an object of class `tclObj` (cf. `?Tcl`). These objects print with the leading `<Tcl>` (which we suppress here when there is no output). To coerce these values into characters, the `tclvalue`

function is used or the `as.character` function. They differ in how they treat spaces and new lines. Conversion to numeric values is also possible through `as.numeric`, but conversion to logical requires two steps.

The `.Tcl` function can be used to read in Tcl scripts as with `.Tcl("source filename")`. This can be used to run arbitrary Tcl scripts within an R session.

The Tk extensions to Tcl have a complicated command structure, and thankfully, `tcltk` provides some more conveniently named functions. To illustrate, the Tcl command to set the text value for a label object (`.label`) would look like

```
% .label configure -text "new text"
```

The `tcltk` provides a corresponding function `tkconfigure`. The above would be done as (assuming `l` is a label object):

```
tkconfigure(l, text="new text")
```

Although the Tcl statement appears to have the object oriented form of “object method arguments,” behind the scenes Tcl creates a command with the same name as the widget with `configure` as a subcommand. This is followed by options passed in using the form `-key value`. The Tk API for `ttklabel`’s `configure` subcommand is

*pathName* **configure** *?option? ?value option value ...?*

The *pathName* is the ID of the label widget. In the Tk documentation paired question marks indicate optional values. In this case, one can specify nothing, returning a list of all options; just an option, to query the configured value; the option with a value, to modify the option; and possibly do more than one at a time. For commands such as `configure`, if possible, there will correspond a function in R of the same name with a `tk` prefix, in this case `tkconfigure`. (The package `tcltk` was written before namespaces, so the “tk” prefix serves that role.) To make consulting the Tk manual pages easier in the text we would describe the `configure` subcommand as *ttklabel configure [options]*. (The R manual pages simply redirect you to the original Tk documentation, so understanding this is important for reading the API.) However, if such a function is present, we will use the R function equivalent when we illustrate code. Some subcommands have further subcommands. An example is to set the selection. In the R function, the second command is appended with a dot, as in `tkselection.set`. (There are just a few exceptions to this.)

**The `tcl` function** Within `tcltk`, the `tkconfigure` function is defined by

```
function(widget, ...) tcl(widget, "configure", ...)
```

The `tcl` function is the workhorse used to piece together Tcl commands. Behind the scenes it turns an R object, `widget`, into the *pathName* above (using

```
tcl(widget, subcommand, key=value, callback)
      /           |           |           \
widget$ID  subcommand  -key value  makeCallback
```

Figure 1.1: How the `tcl` function maps its arguments

its ID component), converts R `key=value` pairs into `-key value` options for Tcl, and adjusts any callback functions. The `tcl` function uses position to create its command, the order of the subcommands needs to match that of the Tk API.

Often, the R object is first, but this is not always the case. As named arguments are only for the `-key value` expansion, we follow the Tcl language and call the arguments “options” in the following. The `tcl` function returns an object of class `tclObj`.

## 1.2 Constructors

In this Chapter, we will stick to a few basic widgets: labels and buttons; and top-level containers to illustrate the basic usage of `tcltk`, leaving for later more detail on containers and widgets.

Unlike GTK+, say, the construction of widgets in `tcltk` is tightly linked to the widget heirarchy. Tk widgets are constructed as children of a parent container with the parent specified to the constructor. When the Tk shell, `wish`, is used or the Tk package is loaded through the Tcl command `package require Tk`, a top level window named “.” is created. In the variable name `.label`, from above, the dot refers to the top level window. In `tcltk` a top-level window is created separately through the `tktoplevel` constructor, as with

```
w <- tktoplevel()
```

Top-level windows will be explained in more detail in Section 2. For now, we just use one to construct a label widget. Like all constructors but a `toplevel` window one, the label constructor (`ttklabel`) requires a specification of the parent container (`w`) and any other options that are desired. A typical usage would look like:

```
l <- ttklabel(w, text="label text")
```

**Options** The first argument of a constructor is the parent container, subsequent arguments are used to specify the options for the constructor given as `key=value` pairs. The Tk API lists these options along with their description.

For a simple label, the following options are possible: `anchor`, `background`, `font`, `foreground`, `justify`, `padding`, `relief`, `text`, and `wraplength`. This is

in addition to the standard options `class`, `compound`, `cursor`, `image`, `state`, `style`, `takefocus`, `text`, `textvariable`, `underline`, and `width`. (Although clearly lengthy, this list is significantly reduced from the options for `tklabel` where options for the many style properties are also included.)

Many of the options are clear from their name. The `padding` argument allows the specification of space in pixels between the text of the label and the widget boundary. This may be set as four values `c(left, top, right, bottom)`, or fewer, with `bottom` defaulting to `top`, `right` to `left` and `top` to `left`. The `relief` argument specifies how a 3-d effect around the label should look if specified. Possible values are `"flat"`, `"groove"`, `"raised"`, `"ridge"`, `"solid"`, or `"sunken"`.

**The functions `tkcget`, `tkconfigure`** Option values may be set through the constructor, or adjusted afterwards by `tkconfigure`. A listing (in Tcl code) of possible options that can be adjusted may be seen by calling `tkconfigure` with just the widget as an argument.

```
head(as.character(tkconfigure(l)))           # first 6 only
```

```
[1] "-background frameColor FrameColor {} {}"
[2] "-foreground textColor TextColor {} {}"
[3] "-font font Font {} {}"
[4] "-borderwidth borderWidth BorderWidth {} {}"
[5] "-relief relief Relief {} {}"
[6] "-anchor anchor Anchor {} {}"
```

The `tkcget` function returns the value of an option (again as a `tclobj` object). The option can be specified two different ways. Either using the Tk style of a leading dash or using the convention that `NULL` values mean to return the value, and not set it.

```
tkcget(l, "-text")                           # retrieve text property
```

```
<Tcl> label text
```

```
tkcget(l, text=NULL)                         # alternate syntax
```

```
<Tcl> label text
```

**Coercion to character** The `tclobj` objects can be coerced to character class two ways. The conversion through `as.character` breaks the return value along whitespace:

```
as.character(tkcget(l, text=NULL))
```

```
[1] "label" "text"
```

## 1. TCL TK: OVERVIEW

---

The `tclvalue` function can also be used to extract the value from a `tclObj`, in this case not breaking along white space.

```
tclvalue(tkget(1, text=NULL))
```

```
[1] "label text"
```

**Buttons** Buttons are constructed using the `ttkbutton` constructor.

```
b <- ttkbutton(w, text="click me")
```

Buttons and labels share many of the same options. However, buttons have a `command` option to specify a callback for when it is clicked. Callbacks will be explained in Section 1.3. Furthermore, buttons have the option `default` to specify which button of a dialog, by default, will get the Return signal when the enter key is pressed. A callback can then be set to respond to this signal. This value for `default` may be "active", indicating the button will get the signal; "normal"; or "disabled", to draw the button without space to indicate it

**tkwidget** Constructors call the `tkwidget` function which returns an object of class `tkwin`. (In Tk the term "window" is used to refer to the drawn widget and not just a top-level window)

```
str(b)
```

```
List of 2
 $ ID : chr ".2.2"
 $ env:<environment: 0x2ab5f88>
- attr(*, "class")= chr "tkwin"
```

The returned widget objects are lists with two components an ID and an environment. The ID component keeps a unique ID of the constructed widget. This is a character string, such as ".1.2.1" coming from the widget heirarchy of the object. This value is generated behind the scenes by the `tcltk` package using numeric values to keep track of the heirarchy. The `env` component contains an environment that keeps track of subwindows, the parent window and any callback functions. This helps ensure that any copies of the widget refer to the same object (?). As the construction of a new widget requires the ID and environment of its parent, the first argument to `tkwidget`, `parent`, must be an R Tk object, not simply its character ID, as is possible for the `tcl` function. The latter is useful in a callback, as only the ID may be known to the callback function.

### Geometry managers

As with Qt, when a new widget is constructed it is not automatically mapped. Tk uses geometry managers to specify how the widget will be drawn within



the parent container. We will discuss two such geometry managers in Section 2, but for now, we note that the simplest way to view a widget in its parent window is through `tkpack`:

```
tkpack(l)
tkpack(b)
```

This command packs the widgets into the top-level window (the parent in this case) in a box-like manner. Unlike GTK+ more than one child can be packed into a top-level window, although we don't demonstrate this further, as later we will use an intermediate `ttkframe` box container so that themes are properly displayed.

### Tcl variables

For several Tk widgets, there is an option `textvariable` for a Tcl variable. These variables are dynamically bound to the widget, so that changes to the variable are propagated to the GUI. (The Tcl variable is a model and the widget a view of the model.) The basic functions involved are `tclVar` to create a Tcl variable, `tclvalue` to get the assigned value and `tclvalue<-` to modify the value.

```
textvar <- tclVar("another label")
l2 <- ttklabel(w, textvariable=textvar)
tkpack(l2)
tclvalue(textvar)
```

```
[1] "another label"
```

```
tclvalue(textvar) <- "new text"
```

The advantages of Tcl variables are like those of the MVC paradigm – a single data source can have its changes propagated to several widgets automatically. If the same text is to appear in different places, their usage is recommended. One disadvantage, is that in a callback, the variable is not passed to the callback and must be found through R's scoping rules.

### Colors and fonts

The label color can be set through its `foreground` property. Colors can be specified by name – for common colors – or by hex RGB values which are common in web programming.

```
tkconfigure(l, foreground="red")
tkconfigure(l, foreground="#00aa00")
```

To find the hex RGB value, one can use the `rgb` function to create RGB values from intensities in  $[0,1]$ . The R function `col2rgb` can translate a named color into RGB values. The `as.hexmode` function will display an integer in hexadecimal notation.

Table 1.1: Standard font names defined by a theme.

Standard font name	Description
TkDefaultFont	Default font for all GUI items not otherwise specified
TkTextFont	Font for text widgets
TkFixedFont	Fixed-width font
TkMenuFont	Menu bar fonts
TkHeadingFont	Font for column headings
TkCaptionFont	Caption font (dialogs)
TkSmallCaptionFont	Smaller caption font
TkIconFont	Icon and text font

**Fonts** Fonts are more involved than colors. Tk version 8.5 made it more difficult to change style properties of individual widgets. This following the practice of centralizing style options for consistency, ease of maintaining code and ease of theming. To set a font for a label, rather than specify the font properties, one configures the font attribute using a pre-defined font name, such as

```
tkconfigure(l, font="TkFixedFont")
```

The "TkFixedFont" value is one of the standard font names, in this case to use a fixed-width font. A complete list of the standard names is provided in Table 1.2. Each theme sets these defaults accordingly. The `tkfont.create` function can be used to create a new font, as with the following commands:

```
tkfont.create("ourFont", family="Helvetica", size=12,  
             weight="bold")
```

```
<Tcl> ourFont
```

```
tkconfigure(l, font="ourFont")
```

Available font families are system dependent. Only "Courier", "Times" and "Helvetica" are guaranteed to be there. A list of available font families is returned by the function `tkfont.families`. Figure 1.2 shows a display of some available font families on a Mac OS X machine. See Example 3.7 for details.

The arguments for `tkfont.create` are optional. The `size` argument specifies the pixel size. The `weight` argument can be used to specify "bold" or "normal". Additionally, a `slant` argument can be used to specify either "roman" (normal) or "italic". Finally, `underline` and `overstrike` can be set with a TRUE or FALSE value.

**Font metrics** The average character size is important in setting the width and height of some components. The can be found through the `tkfont.measure` and `tkfont.metrics` functions as follows:

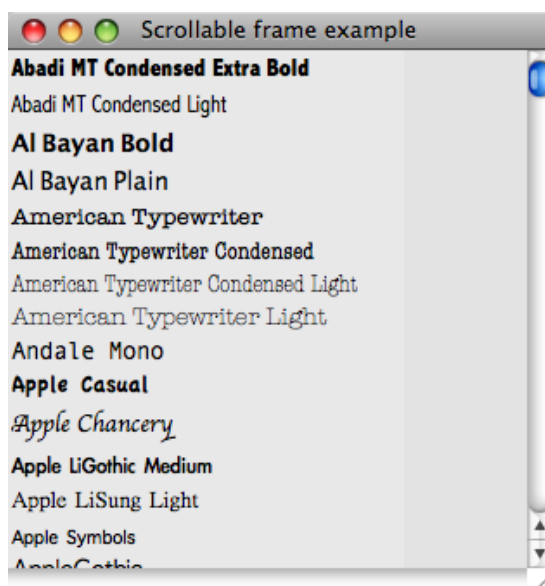


Figure 1.2: A scrollable frame widget (cf. Example 3.7) showing the available fonts on a system.

```
tmp <- tkfont.measure("TkTextFont",paste(c(0:9,LETTERS),collapse=""))
fontWidth <- ceiling(as.numeric(tclvalue(tmp))/36)
tmp <- tkfont.metrics("TkTextFont","linespace"=NULL)
fontHeight <- as.numeric(tclvalue(tmp))
c(width=fontWidth, height=fontHeight)
```

```
width height
      9    16
```

## Images

Many tcltk widgets, including both labels and buttons, can show images. In these cases, either with or without an accompanying text label. Constructing images to display is similar to constructing new fonts, in that a new image object is created and can be reused by various widgets. The `tkimage.create` function is used to create image objects. The following command shows how an image object can be made from the file `tclp.gif` in the current directory:

```
tkimage.create("photo", "::img::tclLogo", file = "tclp.gif")
```

```
<Tcl> ::img::tclLogo
```

## 1. TCL TK: OVERVIEW

---

The first argument, "photo" specifies that a full color image is being used. This option could also be "bitmap" but that is more a legacy option. The second argument specifies the name of the object. We follow the advice of the Tk manual and preface the name with `::img::` so that we don't inadvertently overwrite any existing Tcl commands. The third argument `file` specifies the graphic file. The basic Tk image command only can show only GIF and PPM/PNM images. Unfortunately, not many R devices output in these formats. (The GDD device driver can.) One may need system utilities to convert to the allowable formats or install add-on Tcl packages.

To use the image, one can specify the name for the image option.

```
l <- ttklabel(w, image="::img::tclLogo", text="logo text",
             compound = "top")
```

By default the text will not show. The `compound` argument takes a value of either "text", "image" (default), "center", "top", "left", "bottom", or "right" specifying where the label is in relation to the text.

**Image manipulation** Once an image is created, there are several options to manipulate the image. These are found in the Tk man page for photo, not image. For instance, to change the palette so that instead of fullcolor only 16 shades of gray are used to display the icon, one could issue the command

```
tkconfigure("::img::tclLogo", palette=16)
```

Another useful manipulation to draw attention to an image is to change the `gamma` value when something happens, such as a mouse-over event (cf. Example 2.4).

### Themes

The themed widgets have a style that determines how they are drawn. The separation of style properties from the widget, as opposed to having these set for each construction of a widget, makes it much easier to change the look of a GUI and easier to maintain the code. A collection of styles makes up a theme. The available themes depend on the system. The default theme should enable a GUI to have the native look and feel of the operating system. (This was definitely not the case for the older Tk widgets.) There is no built in command to return the theme, so we use `.Tcl` to call the appropriate Tcl command. The `names` sub command will return the available themes and the `use` sub command can be used to set the theme.

```
.Tcl("ttk::style theme names")
```

```
<Tcl> aqua clam alt default classic
```

```
.Tcl("ttk::style theme use classic")
```

The writing of themes will not be covered, but in Example 2.4 we show how to create a new style for a button.

The example we have shown so far, would not look quite right, as the toplevel window is not a themed widget. To work around that, a `ttkframe` widget is usually used to hold the child components of the top-level window. The following shows how to place a frame inside the window, with some arguments to be explained later that allow it to act reasonably if the window is resized.

```
w <- tktoplevel()
f <- ttkframe(w, padding=c(3,3,12,12)) # Some breathing room
tkpack(f, expand=TRUE, fill="both")    # For window expansion
l <- ttklabel(f, image="::img::tclLogo", text="label",
              compound="top")
tkpack(l)
```

### Window properties and state: `tkwininfo`

Widgets have options which can be set through `tkconfigure` and additionally, when mapped, the “window” they are rendered to has properties, such as a class or size. These properties are queried through the `tkwininfo` function. There are several such properties, and may take different forms. If the API is of the form

```
wininfo subcommand_name window
```

the specification to `tkwininfo` is in the same order (the widget is not the first argument). For instance, the class of a label is returned by the `class` subcommand as

```
tkwininfo("class", l)
```

```
<Tcl> TLabel
```

The window, in this example, `l`, can be specified as an R object, or by its character ID. This is useful, as the return value of some functions is the ID. For instance, the `children` subcommand returns IDs. Below the `as.character` function will coerce these into a vector of IDs.

```
(children <- tkwininfo("children",f))
```

```
<Tcl> .3.1.1
```

```
sapply(as.character(children), function(i) tkwininfo("class", i))
```

```
$ '.3.1.1'
```

```
<Tcl> TLabel
```

There are several possible subcommands, here we list a few. The *tkwininfo* geometry sub command returns the location and size of the widgets' window in the form `width x height + x + y`; the sub commands *tkwininfo* height, *tkwininfo* width, *tkwininfo* x, or *tkwininfo* y can be used to return just those parts. The *tkwininfo* exists command returns 1 (TRUE) if the window exists and 0 otherwise; the *tkwininfo* ismapped sub command returns 1 or 0 if the window is currently mapped (drawn); the *tkwininfo* viewable sub command is similar, only it checks that all parent windows are also mapped. For traversing the widget heirarchy, one has available the *tkwininfo* parent sub command which returns the immediate parent of the component, *tkwininfo* toplevel which returns the ID of the top-level window, and *tkwininfo* children which returns the IDs of all the immediate child components, if the object is a container, such as a top-level window.

### 1.3 Events and Callbacks

The button widget has the `command` option for assigning a callback for when the user clicks the mouse button on the button. In addition to this, one can specify callbacks for many other events that the user may initiate.

#### Callbacks

The `tcltk` package implements callbacks in a manner different from Tk, as the callback functions are R functions, not Tk procedures. This is much more convenient, but introduces some slight differences. In `tcltk` these callbacks can be expressions (unevaluated calls) or functions. We use only the latter, for more clarity. The basic callback function need not have any arguments. For instance, here we show how to print a message when the user clicks a button:

```
w <- tkoplevel()
callback <- function() print("hi")
b <- ttkbutton(w, text="Click me", command = callback)
tkpack(b)
```

The callback's return value is generally not important, although we shall see with the validation framework, discussed in Section 3.2, it can matter.<sup>2</sup> As well, in Tk callbacks are evaluated in the global environment, but this is not so in `tcltk`, which respects the callback's scope.

---

<sup>2</sup>The difference in processing of return values can make porting some Tk code to `tcltk` difficult

## Events

When a user interacts with a GUI, they initiate events. The `tcltk` package allows the programmer to bind callbacks to these events, through the `tkbind` function. This function is called as `tkbind(tag, events, command)`. The `command` is a callback, as described above.

The `tag` argument allows for quite a bit of flexibility. It can be:

- the name of a widget** , in which case the command will be bound to that widget;
- a top-level window** , in which case the command will be bound to the event for the window and all its internal widgets;
- a class of widget** , such as `"TButton"`, in which case all such widgets will get the binding; or
- the value "all"** , in which case all widgets in the application will get the binding.

The possible events (or sequences of events) vary from widget to widget. Events can be specified in a few ways. A single keypress event, can be assigned by specifying the ASCII character generated. For instance, to bind to `C` for the "Click me" button above using the same callback could be done with

```
tkbind(b, "C", callback)
tkfocus(b)
```

The `tkfocus` function is used to set the focus to the button so that it will receive the keypress.

**Events with modifiers** More complicated events can be described with the pattern

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

Examples of a type are `<KeyPress>`, `<ButtonPress>`. The event `<Control-c>` has the type `c` and modifier `Control`. Whereas `<Double-Button-1>` also has the detail `1`. The full list of modifiers and types are described in the man page for `bind`. Some familiar modifiers are `Control`, `Alt`, `Button1` (or its shortening `B1`), `Double` and `Triple`. The event types are the standard X event types along with some abbreviations. These are also listed in the `bind` man page. Some commonly used ones are `ButtonPress`, `ButtonRelease`, `KeyPress`, `KeyRelease`, `FocusIn`, and `FocusOut`.

**Window manager events** Some events are based on window manager events. The `<Configure>` event happens when a component is resized. The `<Map>` and `<Unmap>` events happen when a component is drawn or undrawn.

**Virtual events** Finally, the event may be a “virtual event.” These are represented with `<<EventName>>`. There are predefined virtual events listed in the event man page. These include `<<MenuSelect>>` when working with menus, `<<Modified>>` for text widgets, `<<Selection>>` for text widgets, and `<<Cut>>`, `<<Copy>>` and `<<Paste>>` for working with the clipboard. New virtual events can be produced with the `tkevent.add` function. This takes at least two arguments, an event name and a sequence that will initiate that event. The event man page has these examples coming from the Emacs world:

```
tkevent.add("<<Paste>>", "<Control-y>")
tkevent.add("<<Save>>", "<Control-x><Control-s>")
```

In addition to virtual events occurring when the sequence is performed, the `tkevent.generate` can be used to force an event for a widget. This function requires a widget (or its ID) and the event name. Other options can be used to specify substitution values, described below. To illustrate, this command will generate the `<<Save>>` event for the button `b`:

```
tkevent.generate(b, "<<Save>>")
```

In `tcltk` only one callback can be associated with a widget and event through the call `tkbind(widget,event,callback)`. (Although, callbacks for the widget associated with classes or toplevel windows can differ.) Calling `tkbind` another time will replace the callback. To remove a callback, simply assign a new callback which does nothing.<sup>3</sup>

### % Substitutions

Tk provides a mechanism called *percent substitution* to pass information about the event to callbacks bound to the event. The basic idea is that in the Tcl callback expressions of the type `%X`, for different characters `X`, will be replaced by values coming from the event. In `tcltk`, if the callback function has an argument `X`, then that variable will correspond to the value specified by `%X`. The complete list of substitutions is in the `bind` man page. Useful ones are `x` and `X` to specify the relative or absolute *x*-position of a mouse click (the difference can be found through the `rootx` property of a widget), `y` and `Y` for the *y*-position, `k` and `K` for the keycode (ASCII) and key symbol of a `<KeyPress>` event, and `W` to refer to the ID of the widget that signaled the event the callback is bound to. Example 1.1 will illustrate some of these.

**The after command** The Tcl command `after` will execute a command after a certain delay (specified in milliseconds as an integer) while not interrupting

---

<sup>3</sup>This event handling can prevent being able to port some Tk code into `tcltk`. In those cases, one may consider sourcing in Tcl code directly.



the control flow while it waits for its delay. The function is called in a manner like:

```
ID <- tcl("after", 1000, function() print("1 second passed"))
```

The ID returned by after may be used to cancel the command before it executes. To execute a command repeatedly, can be done along the lines of:

```
afterID <- ""; someFlag <- TRUE
repeatCall <- function(ms=10000, f, w) {
  afterID <- tcl("after", ms, function() {
    if(someFlag) {
      f()
      afterID <- repeatCall(ms, f, w)
    } else {
      tcl("after", "cancel", afterID)
    }
  })
}
repeatCall(100, function() print("running"), w)
```

The flag allows for the cancellation of the repeated call.

### Example 1.1: Drag and Drop

This long example shows how to implement drag and drop between two widgets. Steps are needed to make a widget a drop source, and other steps are needed to make a widget a drop target. The basic idea is that when a value is being dragged, virtual events are generated for the widget the cursor is over. If that widget has callbacks bound to these events, then the drag and drop can be processed. The idea for the code below originated with <http://wiki.tcl.tk/416>.

To begin, we create a simple GUI to hold three widgets. We use buttons for drag and drop, but only because we haven't yet discussed more natural widgets such as the text widgets.

```
w <- tkoplevel()
bDrag <- ttkbutton(w, text="Drag me")
bDrop <- ttkbutton(w, text="Drop here")
tkpack(bDrag)
tkpack(ttklabel(w, text="Drag over me"))
tkpack(bDrop)
```

Before beginning, we define three global variables that can be shared among drop sources to keep track of the drag and drop state. A more elegant example might store these in an environment.

```
.dragging <- FALSE           # currently dragging?
.lastWidgetID <- ""         # last widget dragged over
.dragValue <- ""            # value to transfer
```

## 1. TCL Tk: OVERVIEW

---

To set up a drag source, we bind to three events: a mouse button press, mouse motion, and a button release. For the button press, we set the values of the three global variables.

```
tkbind(bDrag, "<ButtonPress-1>", function(W) {  
    .dragging <- TRUE  
    .lastWidgetID <- as.character(W)  
    .dragValue <- as.character(tkcget(W, text=NULL))  
})
```

For mouse motion, we do several things. First we set the cursor to indicate a drag operation. The choice of cursor is a bit outdated. The commented code shows how one can put in a custom cursor from an xbm file, but this doesn't work for all platforms (e.g., OS X). After setting the cursor, we find the ID of the widget the cursor is over. This uses `tkwinfo` to find the widget containing the  $x, y$ -coordinates of the cursor position. We then generate the `<<DragOver>>` virtual event for this widget, and if this widget is different from the previous last widget, we generate the `<<DragLeave>>` virtual event.

```
tkbind(bDrag, "<B1-Motion>", function(W, X, Y) {  
    if(!.dragging) return()  
    ## see cursor help page in API for more options  
    ## For custom cursors cf. http://wiki.tcl.tk/8674.  
    tkconfigure(W, cursor="coffee_mug") # set cursor  
  
    w = tkwinfo("containing", X, Y) # widget mouse is over  
    if(as.logical(tkwinfo("exists", w))) # does widget exist?  
        tkevent.generate(w, "<<DragOver>>")  
  
    ## generate drag leave if we left last widget  
    if(as.logical(tkwinfo("exists", w)) &&  
        length(as.character(w)) > 0 &&  
        length(as.character(.lastWidgetID)) > 0  
    ) {  
        if(as.character(w)[1] != .lastWidgetID)  
            tkevent.generate(.lastWidgetID, "<<DragLeave>>")  
    }  
    .lastWidgetID <- as.character(w)  
})
```

Finally, if the button is released, we generate the virtual events `<<DragLeave>>` and most importantly `<<DragDrop>>` for the widget we are over.

```
tkbind(bDrag, "<ButtonRelease-1>", function(W, X, Y) {  
    if(!.dragging) return()  
    w = tkwinfo("containing", X, Y)  
  
    if(as.logical(tkwinfo("exists", w))) {
```

```
tkevent.generate(w, "<<DragLeave>>")
tkevent.generate(w, "<<DragDrop>>")
tkconfigure(w, cursor="")
}
.dragging <- FALSE
tkconfigure(W, cursor="")
})
```

To set up a drop target, we bind callbacks for the virtual events generated above to the widget. For the `<<DragOver>>` event we make the widget active so that it appears ready to receive a drag value.

```
tkbind(bDrop, "<<DragOver>>", function(W) {
  if(.dragging)
    tkconfigure(W, default="active")
})
```

If the drag event leaves the widget without dropping, we change the state back to normal.

```
tkbind(bDrop, "<<DragLeave>>", function(W) {
  if(.dragging) {
    tkconfigure(W, cursor="")
    tkconfigure(W, default="normal")
  }
})
```

Finally, if the `<<DragDrop>>` virtual event occurs, we set the widget value to that stored in the global variable `.dragValue`.

```
tkbind(bDrop, "<<DragDrop>>", function(W) {
  tkconfigure(W, text=.dragValue)
  .dragValue <- ""
})
```



## Tcl Tk: Containers and Layout

### 2.1 Top-level windows

Top level windows are created through the `tktoplevel` constructor. The arguments `width` and `height` may be specified to give a requested size. Negative values means the window will not request any size. Top-level windows can have a menubar specified through the `menu` argument. Menus will be covered in Section 3.4.

The `tkdestroy` function can be called to destroy the window and its child components.

The Tk command `wm` is used to interact with top-level windows. This command has several subcommands, leading to `tcltk` functions with names such as `tkwm.title`, the function used to set the window title. As with all such functions, either the top-level window object, or its ID must be the first argument. In this case, the new title is the second.

When a top-level window is constructed there is no option for it not to be shown. However, one can use the `tclServiceMode` function to suspend/resume drawing of any widget through Tk. This function takes a logical value indicating the updating of widgets should be suspended. One can set the value to `FALSE`, initiate the widgets, then set to `TRUE` to display the widgets. After a window is drawn. To iconify an already drawn window can be done through the `tkwm.withdraw` function and reversed with the `tkwm.deiconify` function. Together these can be useful to use in the construction of complicated GUIs, as the drawing of the widgets can seem slow. (The same can be done through the `tkwm.state` function with an option of `"withdraw"` or `"normal"`.)

**Window sizing** The preferred size of a top-level window can be configured through the `width` and `height` options. The absolute size and position of a top-level window in pixels can be queried or specified through the `tkwm.geometry` function. The geometry is specified as a string in the form `=w x h + x + y` (or `-`) where any of `=`, `wxh` or `+x+y` can be omitted. The value

for *x* (if using *+*) indicates how many pixels to the right from the left edge should the window be placed (if using *-* then the left side of the screen is used as a reference). For *y* the top (or bottom) of the screen is the reference.

The `ttksizegrip` widget can be used to add a visual area (usually the lower right corner) for the user to grab on to with their mouse for resizing the window. On some OSes (e.g., Mac OS X) these are added by the window manager automatically.

The `tkwm.resizable` function can be used to prohibit the resizing of a top-level window. The syntax allows either the width or height to be constrained. The following command would prevent resizing of both the width and height of the toplevel window *w*.

```
tkwm.resizable(w, FALSE, FALSE)    # width first
```

When a window is resized, you can constrain the minimum and maximum sizes with `tkwm.minsize` and `tkwm.maxsize`. The aspect ratio (width/height) can be set through `tkwm.aspect`.

For some uses it may be desirable to not have the window manager decorate the window with a title bar etc. Tooltips, for example, can be constructed using this approach. The command `tkoplevel wm overrideredirect logical` takes a logical value indicating if the window should be decorated. Though, not all window managers respect this.

**bindings** Bindings for top-level windows are propagated down to all of their child widgets. If a common binding is desired for all the children, then it need only be specified once for the top-level window.

The `tkwm.protocol` function (not `tkbind`) is used to assign commands to window manager events, most commonly, the delete event when the user clicks the close button on the windows decorations. A top-level window can be removed through the `tkdestroy` function, or through the user clicking on the correct window decorations. When the window decoration is clicked, the window manager issues a "WM\_DELETE\_WINDOW" event. To bind to this, a command of this form `tkwm.protocol(win,"WM_DELETE_WINDOW", callback)` is used.

To illustrate, if *w* is a top-level window, and *e* a text entry widget (cf. Section 3.2) then the following snippet of code would check to see if the text widget has been modified before closing the window. This uses a modal message box described in Section 3.6.

```
tkwm.protocol(w,"WM_DELETE_WINDOW", function() {
    modified <- tcl(e, "edit", "modified")
    if(as.logical(modified)) {
        response <-
            tkmessageBox(icon="question",
                        message="Really close?",
                        detail="Changes need to be saved",
```

```

        type="yesno",
        parent=w)
    if(as.character(response) == "no")
        return()
    }
    tkdestroy(w)                                # otherwise close
})

```

Sometimes, say with dialogs, a top-level window should be related to a different top-level window. The function `tkwm.transient` allows one to specify the master window as its second argument. The new window will mirror the state of the master window, including if the master is withdrawn.

A window can be made to always be the topmost window through the `attributes` subcommand of the `wm` command. However, there is no direct `tcltk` function, so if `w` was to be on top, one would use the `tcl` function as follows:

```
tcl("wm", "attributes", w, topmost=TRUE)
```

## 2.2 Frames

The `ttkframe` constructor produces a themable container that can be used to organize visible components within a GUI. It is often the first thing packed within a top-level window. (As in the example of Section 1.2.)

The options include `width` and `height` to set the requested size, `borderwidth` to specify a border around the frame of a given width, and `relief` to set the border style. The value of `relief` is chosen from the default "flat", "groove", "raised", "ridge", "solid", or "sunken". The `padding` option can be used to put space within the border between the border and subsequent children.

### Label Frames

The `ttklabelframe` constructor produces a frame with an optional label that can be used to set off and organize components of a GUI. The label is set through the option `text`. Its position is determined by the option `labelanchor` taking values labeled by compass headings (combinations of `n`, `e`, `w`, `s`). The default is theme dependent, although typically "nw" (upper left).

**Separators** To use a single line to separate out areas in a GUI, the `ttkseparator` widget can be used. The lone widget-specific option is `orient` which takes values of "horizontal" (the default) or "vertical". This widget must be told to stretch when added to a container, as described in the next section.

### 2.3 Geometry Managers

Tcl uses *geometry managers* to place child components within their parent windows. There are three such managers, but we describe only two, leaving the lower-level `place` command for the official documentation. The use of geometry managers, allows Tk to quickly reallocate space to a GUI's components when it is resized. The `tkpack` function will place children into their parent in a box-like manner. We have seen in several examples throughout the text, that through the use of nested boxes, one can construct quite flexible layouts, and Example 2.2 will illustrate that once again. When simultaneous horizontal and vertical alignment of child components is desired, the `tkgrid` function can be used to manage the components.

A GUI may use a mix of `pack` and `grid` to manage the child components, but all siblings in the widget hierarchy must be managed the same way. Mixing the two will typically result in a lockup of the R session.

#### Pack

We have illustrated how `tkpack` can be used to manage how child components are viewed within their parent. The basic usage `tkpack(child)` will pack in the child components from top to bottom. The `side` option can take a value of "left", "right", "top" (default), or "bottom" to adjust where the children are placed. These can be mixed and matched, but sticking to just one direction is typical, with nested frames to give additional flexibility.

**after, before** The `after` and `before` options can be used to place the child before or after another component. These are used as with `tkpack(child1, after=child2)`. The object `child2` can be an R object or its ID. The latter might be useful, say when all the children are listed using the command `tkwininfo("children",parent)` which returns the IDs of the immediate child components.

**padding** In addition to the `padding` option for a frame container, the `ipadx`, `ipady`, `padx`, and `pady` options can be used to add space around the child components. Figure 2.1 has an example. The `x` and `y` indicate left-right space or top-bottom space. The `i` stands for internal padding that is left on the sides or top and bottom of the child within the border, for `padx` the external padding added around the border of the child component. The value can be a single number or pair of numbers for asymmetric padding.

This sample code shows how one can easily add padding around all the children of the frame `f` using the `tkpack "configure"` subcommand.

**Cavity model** The packing algorithm, as described in the Tk documentation, is based on arranging where to place a slave into the rectangular unal-



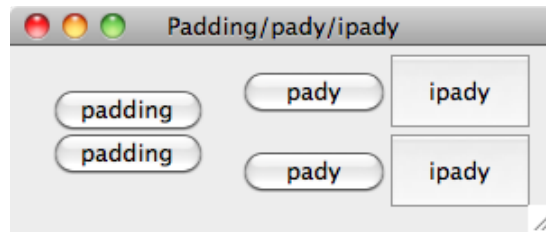


Figure 2.1: Various ways to put padding between widgets using a box container and `tkpack`. The `padding` option for the box container puts padding around the cavity for all the widgets. The `pady` option for `tkpack` puts padding around the top and bottom on the border of each widget. The `ipady` option for `tkpack` puts padding within the top and bottom of the border for each child (modifying the theme under Mac OS X).

located space called a cavity. We use the nicer terms child component and box to describe these. When a child is placed inside the box, say on the top, the space allocated to the child is the rectangular space with width given by the width of the box, and height the sum of the requested height of the child plus twice the `ipady` amount (or the sum if specified with two numbers). The packer then chooses the dimension of the child component, again from the requested size plus the `ipad` values for `x` and `y`. These two spaces may, of course, have different dimensions.

**anchor** By default, the child will be placed centered along the edge of the box within the allocated space and blank space, if any, on both sides. If there is not enough space for the child in the allocated space, the component can be drawn oddly. Enlarging the top-level window can adjust this. When there is more space in the box than requested by the child component, there are other options. The `anchor` option can be used to anchor the child to a place in the box by specifying one of the valid compass points (eg. `"n"` or `"se"`) leaving blank space around the child. External padding between the child and the box can be set through the `padx` and `pady` options.

**expand, fill** When there is more space in the original box than needed by the children the extra space will be left blank unless some children have the option `expand` set to `TRUE`. In this case, the extra space is allocated evenly to each child with this set. The `fill` option is often used when `expand` is set. The `fill` option is used to base the size of the child on the available cavity in the box – not on the requested size of the child. The `fill` option can be `"x"`, `"y"` or `"both"`. The first two expanding the child's size in just one direction, the latter in both.

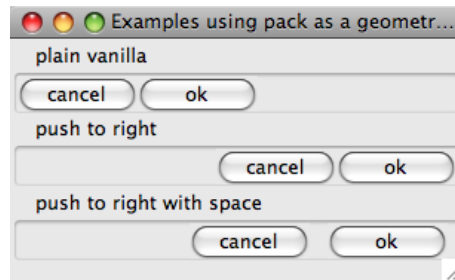


Figure 2.2: Demonstration of using `tkpack` options showing effects of using the `side` and `padx` options to create dialog buttons.

**forget** Child components can be forgotten by the window manager, unmaping them but not destroying them, with the `tkpack forget` subcommand, or the convenience function `tkpack.forget`. After a child component is removed this way, it can be re-placed in the GUI using a geometry manager. In `gWidgetstcltk` this is used to create a `gexpandgroup` container, as such a container is not provided by Tk.

**Introspection** The subcommand `tkpack slaves` will return a list of the child components packed into a frame. Coercing these return values to character via `as.character` will produce the IDs of the child components. The subcommand `tkpack info` will provide the packing info for a child.

### Example 2.1: Packing dialog buttons

This example shows how one can pack in action buttons, such as when a dialog is created.

The first example just uses `tkpack` without any arguments except the `side` to indicate the buttons are packed in left to right, not top to bottom.

```
f1 <- ttklabelframe(f, text="plain vanilla")
tkpack(f1, expand=TRUE, fill="x")
l <- function(f)
  list(ttkbutton(f, text="cancel"), ttkbutton(f, text="ok"))
QT <- sapply(l(f1), function(i) tkpack(i, side="left"))
```

Typically the buttons are right justified. One way to do this is to pack in using `side` with a value of `"right"`. This shows how to use a blank expanding label to take up the space on the left.

```
f2 <- ttklabelframe(f, text="push to right")
tkpack(f2, expand=TRUE, fill="x")
tkpack(ttklabel(f2, text=" "), expand=TRUE, fill="x", side="left")
QT <- sapply(l(f2), function(i) tkpack(i, side="left"))
```

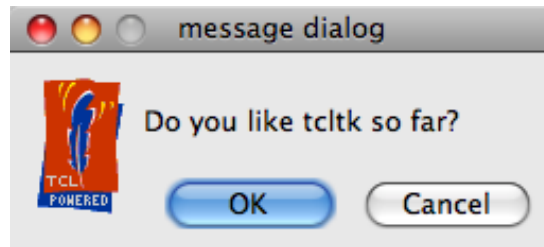


Figure 2.3: Example of a simple dialog

Finally, we add in some padding to conform to Apple's design specification that such buttons should have a 12 pixel separation.

```
f3 <- ttklabelframe(f, text="push to right with space")
tkpack(f3, expand=TRUE, fill="x")
tkpack(ttklabel(f3, text=" "), expand=TRUE, fill="x", side="left")
QT <- sapply(1(f3), function(i) tkpack(i, side="left", padx=6))
```

### Example 2.2: A non-modal dialog

This example shows how to use a window, frames, labels, buttons, icons, packing and bindings to create a non-modal dialog.

Although not written as a function, we set aside the values that would be passed in were it.

```
title <- "message dialog"
message <- "Do you like tcltk so far?"
parent <- NULL
QT <- tkimage.create("photo", "::img::tclLogo",
                     file = system.file("images", "tclp.gif",
                                         package="ProgGUIinR"))
```

The main top-level window is then given a title, then withdrawn while the GUI is created.

```
w <- tktoplevel(); tkwm.title(w, title)
tkwm.state(w, "withdrawn")
f <- ttkframe(w, padding=c(3, 3, 12, 12))
tkpack(f, expand=TRUE, fill="both")
```

As usual, we added a frame so that any themes are respected.

If the parent is non-null and is viewable, then the dialog is made transient to a parent. The parent need not be a top-level window, so `tkwinfo` is used to find the parent's top-level window. For Mac OS X, we use the `notify` attribute to bounce the dock icon until the mouse enters the window area.

```
if(!is.null(parent)) {
  parentWin <- tkwinfo("toplevel", parent)
```

## 2. TCL Tk: CONTAINERS AND LAYOUT

---

```
if(as.logical(tkwininfo("viewable", parentWin))) {
    tkwm.transient(w, parent)
    ## effects OS X only now
    tcl("wm","attributes",parentWin, notify=TRUE) # bounce
    tkbind(parentWin,"<Enter>", function()
        tcl("wm","attributes",parentWin, notify=FALSE)) #stop
    }
}
```

We will use a standard layout for our dialog with an icon on the left, a message and buttons on the right. We pack the icon into the left side of the frame,

```
l <- ttklabel(f, image="::img::tclLogo", padding=5) # recycle
tkpack(l,side="left")
```

A nested frame will be used to layout the message area and button area. Since the tkpack default is to pack in top to bottom, no side specification is made.

```
f1 <- ttkframe(f)
tkpack(f1, expand=TRUE, fill="both")
#
m <- ttklabel(f1, text=message)
tkpack(m, expand=TRUE, fill="both")
```

The buttons have their own frame, as they are layed out horizontally.

```
f2 <- ttkframe(f1)
tkpack(f2)
```

The callback function for the OK button prints a message then destroys the window.

```
okCB <- function() {
    print("That's great")
    tkdestroy(w)
}
okButton <- ttkbutton(f2, text="OK", default="active")
```

We bind the callback to both a left mouse click on the button, and if the user presses return when the button has the focus. The default="active" argument, makes this button the one that gets the Return event when the return key is pressed.

```
tkbind(okButton, "<Button-1>", okCB)
tkbind(okButton, "<Return>", okCB)
cancelButton <- ttkbutton(f2, text="Cancel",
    command=function() tkdestroy(w))
tkpack(okButton, side="left", padx=12) # give some space
tkpack(cancelButton)
```

Now we bring the dialog back from its withdrawn state, fix the size and set the focus on the OK button.

```
tkwm.state(w, "normal")
tkwm.resizable(w, FALSE, FALSE)
tkfocus(okButton)
```

Finally, the following bindings make the buttons look active when the keyboard focus is on them, generating a `FocusIn` event, then a `FocusOut` event. We make a binding for the top-level window, then within the callback check to see if the widget emitting the signal is of a themed button class.

```
isTButton <- function(W)
  as.character(tkwininfo("class",W)) == "TButton"
tkbind(w,"<FocusIn>", function(W) {
  if(isTButton(W)) tkconfigure(W,default="active")
})
tkbind(w,"<FocusOut>", function(W) {
  if(isTButton(W)) tkconfigure(W,default="normal")
})
```

## Grid

The `tkgrid` geometry manager is used to place child widgets in rows and columns. In its simplest usage, a command like

```
tkgrid(child1, child2, ..., childn)
```

will place the  $n$  children in a new row, in columns 1 through  $n$ . However, the specific row and column can be specified through the `row` and `column` options. Counting of rows and columns starts with 0. Spanning of multiple rows and columns can be specified with integers 2 or greater by the `rowspan` and `colspan` options. These options, and others can be adjusted through the `tkgrid.configure` function.

**The `tkgrid.rowconfigure`, `tkgrid.columnconfigure` commands** When the managed container is resized, the grid manager consults weights that are assigned to each row and column to see how to allocate the extra space. These weights are configured with the `tkgrid.rowconfigure` and `tkgrid.columnconfigure` functions through the option `weight`. The weight is a value between 0 and 1. If there are just two rows, and the first row has weight 1/2 and the second weight 1, then the extra space is allocated twice as much for the second row. The specific row or column must also be specified. Rows and columns are referenced starting with 0 not the usual R-like 1. So to specify a weight of 1 to the first row would be done with a command like:

```
tkgrid.rowconfigure(parent, 0, weight=1)
```

**The sticky option** When more space is available than requested by the child component, the `sticky` option can be used to place the widget into the grid. The value is a combination of the compass points "n", "e", "w", and "s". A specification "ns" will make the child component “stick” to the top and bottom of the cavity that is provided, similar to the `fill="y"` option for `tkpack`. A value of "news" will make the child component expand in all the direction like `fill="both"`.

**Padding** As with `tkpack`, `tkgrid` has options `ipadx`, `ipady`, `padx`, and `pady` to give internal and external space around a child.

**Size** The function `tkgrid.size` will return the number of columns and rows of the specified parent container that is managed by a grid. This can be useful when trying to position child components through the options `row` and `column`.

**Forget** To remove a child from the parent, the `tkgrid.forget` function can be used with the child object as its argument.

### Example 2.3: Using `tkgrid` and `tkpack` to draw some world flags

This example shows how the `tkpack` and `tkgrid` geometry managers can be used to draw some of the world flags. For these, we consulted <https://www.cia.gov/library/publications/the-world-factbook/docs/flagsoftheworld.html>.

We will make the dimensions of the flags true to the flag proportions. These we found at <http://flagspot.net/flags/xf-size.html>. Here we define the proportions for the flags of interest.

```
dims <- cbind(Benin=2:3, Cameroon=2:3,Guinea=2:3, Mali=2:3,
              Bolivia=2:3, Lithuania=1:2,Congo=2:3, Guyana=1:2,
              Togo= 2:3)
```

This is a convenience function to create `tkframes` with different background colors. We use `tkframe` here – not `ttkframe` – as it has a `background` property.

```
makeColors <- function(parent)
  list(green = tkframe(parent, background="green"),
       red   = tkframe(parent, background="red"),
       yellow = tkframe(parent, background="yellow"))
```

This convenience function packs a frame into a top-level window.

```
makeTopLevel <- function(country) {
  w <- tktoplevel()
  tkwm.title(w, country)
  f <- ttkframe(w, padding=c(3,3,3,12))
```



Figure 2.4: Example of world flags to illustrate `tkpack` and `tkgrid` usage. The Mali flag uses `expand=TRUE` to allocate space evenly, `fill="both"` to have the child fill the space and `side="left"` to place the children, whereas Lithuania uses `side="top"`. The Benin flag takes advantage of `tkgrid` to layout the colors in a grid. The left color has `rowspan=2` set. The Togo flag could be done using just `grid`, but a mix is demonstrated.

```
tkpack(f, expand=TRUE, fill="both")
return(list(w=w, f= f, country=country))
}
```

Our first flags are Cameroon (GRY), Guinea (RYG), and Mali (GYR). These are flags with 3 equal vertical strips of color. We use `tkpack` with `side="left"` to pack in the colors from left to right. The `expand=TRUE` option causes extra space to be allocated equally to the three children, preserving the equal sizes in this case.

```
win <- makeToplevel("Cameroon")
w <- win$w; f <- win$f
l <- makeColors(f)
tkpack(l$green, l$red, l$yellow, expand=TRUE,
       fill="both", side="left")
```

To create Guinea's flag we simply move the green strip to the end.

```
## Guinea just moves colors around
tkpack("forget", l$green)
tkpack(l$green, expand=TRUE, fill="both", side="left")
tkwm.title(win$w, "Guinea")
```

For Mali, we flip the position of green and red. We pack them in relative to the yellow strip using the before and after options to tkpack.

```
tkpack("forget", l$green)
tkpack("forget", l$red)
tkpack(l$green, before=l$yellow, expand=TRUE, fill="both", side="left")
tkpack(l$red, after=l$yellow, expand=TRUE, fill="both", side="left")
tkwm.title(win$w, "Mali")
```

Lithuania is similar, only the stripes run horizontally. We pack from top to bottom to achieve this.

```
win <- makeTopLevel("Lithuania")
l <- makeColors(win$f)
tkpack(l$yellow, l$green, l$red, expand=TRUE, fill="both", side="top")
```

Benin's flag is better suited for the grid geometry manager. We use a combination of rowspan and colspan to get the proper arrangement. In this case, the proportions of the colors are achieved through equal weights when we configure the row and columns.

```
## benin is better suited for grid
win <- makeTopLevel("Benin")
l <- makeColors(win$f)
tkgrid(l$green, row=0, column=0, rowspan=2, sticky="news")
tkgrid(l$yellow, row=0, column=1, colspan=2, sticky="news")
tkgrid(l$red, row=1, column=1, colspan=2, sticky="news")
## use grid in equal sizes to get spacing right
tkgrid.rowconfigure(win$f, 0:1, weight=1)
tkgrid.columnconfigure(win$f, 0:2, weight=1)
```

Togo is trickier. We could use grid, as above, with the proper combinations of row and colspan. Instead we do this less directly to illustrate the mixing of the tkgrid and tkpack geometry managers.

```
win <- makeTopLevel("Togo")
f <- win$f
l <- makeColors(f)
upperR <- ttkframe(f); bottom <- ttkframe(f)
## upper left red rectangle
tkgrid(l$red, row=0, column=0, sticky="news")
tkgrid(upperR, row=0, column=1, sticky="news")
tkgrid(bottom, row=1, column=0, colspan=2, sticky="news")
## top right stripes
l1 <- makeColors(upperR)
tkpack(l1$yellow, expand=TRUE, fill="both", side="top")
tkpack(l1$green, expand=TRUE, fill="both", side="top")
## bottom stripes
l2 <- makeColors(bottom)
tkpack(l2$yellow, expand=TRUE, fill="both", side="top")
```



```
tkpack(l2$green, expand=TRUE, fill="both", side="top")
tkgrid.rowconfigure(f, 0:1, weight=1)
tkgrid.columnconfigure(f, 0, weight=8)
tkgrid.columnconfigure(f, 1, weight=10) # not quite uniform
```

#### Example 2.4: Using tkgrid to create a toolbar

Tk does not have a toolbar widget. Here we use tkgrid to show how we can add one to a top-level window in a manner that is not affected by resizing. We begin by packing a frame into a top-level window.

```
w <- tktoplevel(); tkwm.title(w, "Toolbar example")
f <- ttkframe(w, padding=c(3,3,12,12))
tkpack(f, expand=TRUE, fill="both")
```

Our example has two main containers: one to hold the toolbar buttons and one to hold the main content.

```
tbFrame <- ttkframe(f, padding=0)
contentFrame <- ttkframe(f, padding=4)
```

The tkgrid geometry manager is used to place the toolbar at the top, and the content frame below. The choice of sticky and the weights ensure that the toolbar does not resize if the window does.

```
tkgrid(tbFrame, row=0, column=0, sticky="we")
tkgrid(contentFrame, row=1, column=0, sticky="news")
tkgrid.rowconfigure(f, 0, weight=0)
tkgrid.rowconfigure(f, 1, weight=1)
tkgrid.columnconfigure(f, 0, weight=1)
## some example to pack into the content area
tkpack(ttklabel(contentFrame, text="Some content"))
```

Now to add some buttons to the toolbar. We first show how to create a new style for a button, slightly modifying the themed button to set the font and padding, and eliminate the border if the OS allows.

```
tcl("ttk::style","configure","Toolbar.TButton",
    font="helvetica 12", padding=0, borderwidth=0)
```

This makeIcon function finds stock icons from the gWidgets package and adds them to a button.

```
makeIcon <- function(parent, stockName, command=NULL) {
  iconFile <- system.file("images",
                          paste(stockName,"gif",sep="."),
                          package="gWidgets")
  if(nchar(iconFile) == 0) {
    b <- ttkbutton(parent, text=stockName, width=6)
  } else {
```

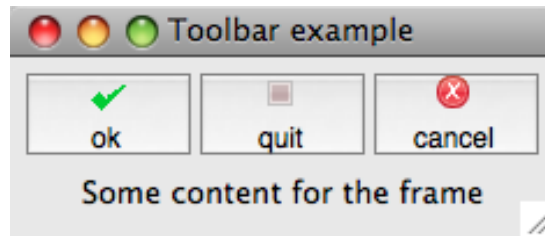


Figure 2.5: Illustration of using tkpack to make a toolbar.

```
iconName <- paste("::img::",stockName, sep="")
tkimage.create("photo", iconName, file = iconFile)
b <- ttkbutton(parent, image=iconName,
               style="Toolbar.TButton", text=stockName,
               compound="top", width=6)
if(!is.null(command))
    tkconfigure(b, command=command)
}
return(b)
}
```

To illustrate, we pack in some icons. Here we use tkpack. One does not use tkpack and tkgrid to manage children of the same parent, but these are children of tbFrame, not f.

```
tkpack(makeIcon(tbFrame, "ok"), side="left")
tkpack(makeIcon(tbFrame, "quit"), side="left")
tkpack(makeIcon(tbFrame, "cancel"), side="left")
```

These two bindings show how to slightly highlight the icon when the mouse is over that button, so that the user has some extra feedback.

```
changeGamma <- function(W, gamma=1.0) {
  if(as.character(tkwininfo("class",W)) == "TButton") {
    img <- tkcget(W,"image"=NULL)
    tkconfigure(img, gamma=gamma)
  }
}
tkbind(w,"<Enter>", function(W) changeGamma(W, gamma=0.5))
tkbind(w,"<Leave>", function(W) changeGamma(W, gamma=1.0))
```

### 2.4 Other containers

Tk provides just a few other basic containers, here we describe paned windows and notebooks.

## Paned Windows

A paned window is constructed by the function `ttkpanedwindow`. The primary option, outside of setting the requested width or height with `width` and `height`, is `orient`, which takes a value of "vertical" (the default) or "horizontal". This specifies how the children are stacked, and is opposite how the sash is drawn.

The returned object can be used as a parent container, although one does not use the geometry managers to manage them. Instead, the `add` command is used. For example:

```
w <- tkoplevel(); tkwm.title(w, "Paned window example")
pw <- ttkpanedwindow(w, orient="horizontal")
tkpack(pw, expand=TRUE, fill="both")
left <- ttklabel(pw, text="left")
right <- ttklabel(pw, text="right")
#
tkadd(pw, left, weight=1)
tkadd(pw, right, weight=2)
```

When resizing which child gets the space is determined by the associated weight, specified as an integer. The default uses even weights. Unlike GTK+ more than two children are allowed.

**Forget** The subcommand `ttkpanedwindow forget` can be used to unmanage a child component. For the paned window, we have no convenience function, so we call as follows:

```
tcl(pw, "forget", right)
tkadd(pw, right, weight=2) ## add back
```

**Sash position** The sash between two children can be adjusted through the subcommand `ttkpanedwindow sashpos`. The index of the sash needs specifying, as there can be more than one. Counting starts at 0. The value for `sashpos` is in terms of pixel width (or height) of the paned window. The width can be returned as follows:

```
tcl(pw, "sashpos", 0, 150)
```

```
<Tcl> 59
```

```
as.integer(tkwininfo("width", pw)) # or "height"
```

```
[1] 71
```

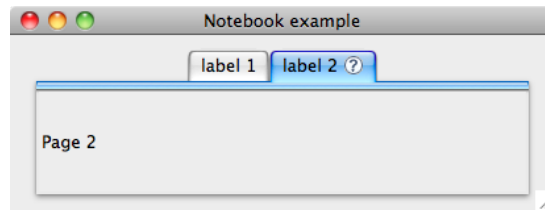


Figure 2.6: A basic notebook under Mac OS X

### Notebooks

The `ttknotebook` constructor returns a notebook object. In Tk the object itself, becomes a command with the subcommands being important. There are no convenience functions for these, so we will use the `tcl` function directly.

Notebook pages can be added through the `ttknotebook add` subcommand or inserted after a page through the `ttknotebook insert` subcommand. The latter requires a tab ID to be specified, as described below. The tab label is configured similarly to `ttklabel` through the options `text` and the optional `image`, which if given has its placement determined by `compound`. The placement of the child component within the notebook page is manipulated similarly as `tkgrid` through the `sticky` option with values specified through compass points. Extra padding around the child can be added with the `padding` option. Typically, the child components would be containers to hold more complicated layouts.

**Tab identifiers** Many of the commands for a notebook require a specification of a desired tab. This can be given by index, starting at 0; by the values "current" or "end"; by the child object added to the tab, either as an R object or an ID; or in terms of *x-y* coordinates in the form "@x,y" (likely found through a binding).

To illustrate, if `nb` is a `ttknotebook` object, then these commands would add pages (cf. Figure 2.6):

```
iconFile <- system.file("images",paste("help","gif",sep="."),
                        package="gWidgets")
iconName <- "::tcl::helpIcon"
QT <- tkimage.create("photo", iconName, file = iconFile)
#
l2 <- ttklabel(nb, text="Page 2")
tkadd(nb, l2, sticky="nswe", text="label 2",
      image=iconName, compound="right")
## put l1 first (tabID is 0), use tkinsert
l1 <- ttklabel(nb, text="Page 1")
tkinsert(nb, 0, l1, sticky="nswe", text="label 1")
```

There are several useful subcommands to extract information from the notebook object. For instance, `index` to return the page index (0-based), `tabs` to return the page IDs, `select` to select the displayed page, and `forget` to remove a page from the notebook. Except for `tabs`, these require a specification of a tab ID.

```
tcl(nb, "index", "current")          # current page for tabID

<Tcl> 1

length(as.character(tcl(nb,"tabs"))) # number of pages

[1] 2

tcl(nb, "select", 0)                 # select viewable page by index
tcl(nb, "forget", 11)                # forget removes page from notebook
tcl(nb, "add", 11)                   # can be managed again.
```

The notebook state can be manipulated through the keyboard, provided traversal is enabled. This can be done through

```
QT <- tcl("ttk::notebook::enableTraversal", nb)
```

If enabled, the shortcuts such as control-tab to move to the next tab are implemented. If new pages are added or inserted with the option `underline`, which takes an integer value (0-based) specifying which character in the label is underlined, then a keyboard accelerator is added for that letter.

**Bindings** Beyond the usual events, the notebook widget also generates a `<<NotebookTabChanged>>` virtual event after a new tab is selected.

The notebook container in Tk has a few limitations. For instance, there is no graceful management of too many tabs, as there is with GTK+, as well there is no easy way to implement close buttons as an icon, as in Qt.



## Tcl Tk: Widgets

Tk has widgets for the common GUI controls. As mentioned in Chapter 1 – where we illustrated both buttons and labels – the constructors for these widgets call the function `tkwidget` which calls the appropriate Tk command and adds in extra information including an ID and an environment. As with labels and buttons, one primarily uses `tkconfigure` and `tkcget` to set and get properties of the widget when a Tcl variable is not used to store the data for the widget.

### 3.1 Selection Widgets

This section covers the many different ways to present data for the user to select a value. The widgets can use Tcl variables to refer to the value that is displayed or that the user selects. Recall, these were constructed through `tclVar` and manipulated through `tclvalue`. For example, a logical value can be stored as

```
value <- tclVar(TRUE)
tclvalue(value) <- FALSE
tclvalue(value)
```

```
[1] "0"
```

As `tclvalue` coerces the logical into the character string "0" or "1", some coercion may be desired.

#### Checkbutton

The `ttkcheckbutton` constructor returns a check button object. The checkbutton's value (TRUE or FALSE) is linked to a Tcl variable which can be specified using a logical value. The checkbutton label can also be specified through a Tcl variable using the `textvariable` option. Alternately, as with the `ttklabel` constructor, the label can be specified through the `text` option.

### 3. TCL Tk: WIDGETS

---

This allows one to specify an image as well and arrange its display, as is done with `ttklabel`, using the `compound` option.

The `command` argument is used at construction time to specify a callback when the button is clicked. The callback is called when the state toggles, so often a callback considers the state of the widget before proceeding. To add a callback with `tkbind` use `<ButtonRelease-1>`, as the callback for the event `<Button-1>` is called before the variable is updated.

For example, if `f` is a frame, we can create a new check button with the following:

```
value <- tclVar(TRUE)
callback <- function() print(tclvalue(value))      # uses global
labelVar <- tclVar("check button label")
cb <- ttkcheckbutton(f, variable=value,
                    textvariable=labelVar, command=callback)
tkpack(cb)
```

To avoid using a global variable is not trivial here. There is no easy way to pass user data through to the callback, and there is no easy way to get the R object from the values passed through the `%` substitution values. The variable holding the value can be found through

```
tkcget(cb, "variable"=NULL)
```

```
<Tcl> ::RTcl3
```

But manipulating that is difficult. A more general strategy within R would be to use a function closure to encapsulate the variables or an environment to store the global values.

#### Radio Buttons

Radiobuttons are checkbuttons linked through a shared Tcl variable. Each button is constructed through the `ttkradiobutton` constructor. Each button has a value and a label, which need not be the same. The variable refers to the value. As with labels, the radio button labels may be specified through a text variable or the `text` option, in which case, as with a `ttklabel`, an image may also be incorporated through the `image` and `compound` options. In Tk the placement of the buttons is managed by the programmer.

This small example shows how radio buttons could be used for selection of an alternative hypothesis, assuming `f` is a parent container.

```
values <- c("less", "greater", "two.sided")
var <- tclVar(values[3])      # initial value
callback <- function() print(tclvalue(var))
sapply(values, function(i) {
  rb <- ttkradiobutton(f, text=i, variable=var,
                      value=i, command=callback)
```



```
tkpack(rb, side="top", anchor="w")
})
```

```
$less
```

```
$greater
```

```
$two.sided
```

## Comboboxes

The `ttkcombobox` constructor returns a combobox object to select from a list of values, or with the appropriate option, allowing the user to specify a value. Like radiobuttons and checkbuttons, the value of the combobox can be specified using a Tcl variable to the option `textvariable`, making the getting and setting of the displayed value straightforward. The possible values to select from are specified as a character vector through the `values` option. (This may require one to coerce the results to the desired class.) Unlike GTK+ and Qt there is no option to include images in the displayed text. One can adjust the alignment through the `justify` options. By default, a user can add in additional values through the entry widget part of the combobox. The `state` option controls this, with the default "normal" and the value "readonly" as an alternative.

To illustrate, again suppose `f` is a parent container. Then we begin by defining some values to choose from and a Tcl variable.

```
values <- rownames(mtcars)
var <- tclVar(values[1])           # initial value
```

The constructor call is as follows:

```
cb <- ttkcombobox(f,
                  values=values,
                  textvariable=var,
                  state="normal", # or "readonly"
                  justify="left")
tkpack(cb)
```

The possible values the user can select from can be configured after construction through the `values` option:

```
tkconfigure(cb, values=tolower(values))
```

**Setting the value** Setting values can be done through the Tcl variable, or by value or index (0-based) using the `ttkcombobox set` sub command through `tkset` or the `ttkcombobox current` sub command.

### 3. TCL Tk: WIDGETS

---

```
tclvalue(var) <- values[2]      # using tcl variable
tkset(cb, values[4])           # by value
tcl(cb, "current", 4)          # by index
```

**Getting the value** One can retrieve the selected object in various ways: from the Tcl variable. Additionally, the *ttkcombobox* *get* subcommand can be used through *tkget*.

```
tclvalue(var)                  # TCL variable
```

```
[1] "hornet sportabout"
```

```
tkget(cb)                      # get subcommand
```

```
<Tcl> hornet sportabout
```

```
tcl(cb, "current")            # 0-based index
```

```
<Tcl> 4
```

**Events** The virtual event `<<ComboboxSelected>>` occurs with selection. When the combobox may be edited, a user may expect some action when the return key is pressed. This triggers a `<Return>` event. To bind to this event, one can do something like the following:

```
tkbind(cb, "<Return>", function(W) {
  val <- tkget(W)
  cat(as.character(val), "\n")
})
```

For editable comboboxes, the widget also supports some of the *ttkentry* commands discussed in Section 3.2.

#### Scale widgets

The *ttkscale* constructor to produce a themable scale (slider) control is missing<sup>1</sup>. You can define your own:

```
ttkscale <- function(parent, ...) tkwidget(parent, "ttk::scale", ...)
```

The orientation is set through the option *orient* taking values of "horizontal" (the default) or "vertical". For sizing the slider, the *length* option is available. To set the range, the basic options are *from* and *to*. There is no *by* option as of Tk 8.5. The constructor *ttkscale*, for a non-themable slider, has the option *resolution* to set that. The *variable* option is used for specifying

---

<sup>1</sup>As of R 2.11.0

a Tcl variable to record the value of the slider. Otherwise the `value` option is available. The `tkget` and `tkset` function (using the `ttkscale` `get` and `ttkscale` `set` sub commands) can be used to get and set the value shown. They are used in the same manner as the same-named subcommands for a combobox. Again, the `command` option can be used to specify a callback for when the slider is manipulated by the user.

## Spinboxes

In Tk version 8.5 there is no themable spinbox widget. In Tk the `spinbox` command produces a non-themable spinbox. Again, there is no direct `tkspinbox` constructor, but one can be defined with:

```
tkspinbox <- function(parent, ...)
  tkwidget(parent, "tk::spinbox", ...)
```

The non-themable widgets have many more options than the themable ones, as style properties can be set on a per-widget basis. We won't discuss those here. The spinbox can be used to select from a sequence of numeric values or a vector of character values.

The basic options to set the range for a numeric spinbox are `from`, `to`, and `increment`. The `textvariable` option can be used to link the spinbox to a Tcl variable. As usual, this allows the user to easily get and set the value displayed. Otherwise, the `tkget` and `tkset` functions may be used for these tasks. The option `state` can be used to specify whether the user can enter values, the default of "normal"; not edit the value, but simply select one of the given values ("readonly"), or not select a value ("disabled").

In Tk, spinboxes can also be used to select from a list of text values. These are specified through the `values` option. For the latter, the `wrap` option, as in `wrap=TRUE`, can be used to wrap around the list of values when the end is reached. As with a combobox, when the Tk spinbox displays character data and is in the "normal" state, the widget can be controlled like the entry widget of Section 3.2.

### Example 3.1: A GUI for `t.test`

This example illustrates how the basic widgets can be combined to make a dialog for gathering information to run a *t*-test. A realization is shown in Figure 3.1.

We will use a data store to hold the values to be passed to `t.test`. For the data store, we use an environment to hold Tcl variables.

```
### Data model
e <- new.env()
e$x <- tclVar(""); e$f <- tclVar(""); e$data <- tclVar("")
e$mu <- tclVar(0); e$alternative <- tclVar("two.sided")
e$conf.level <- tclVar(95); e$var.equal <- tclVar(FALSE)
```

### 3. TCL Tk: WIDGETS

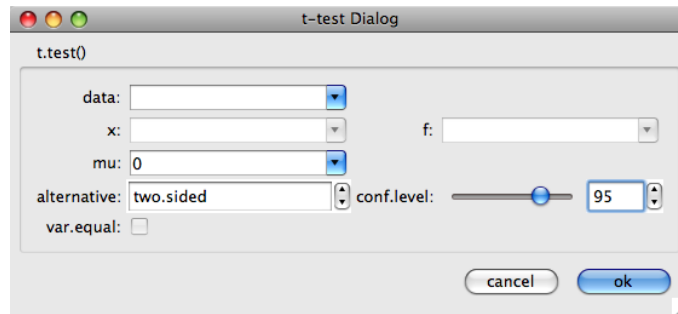


Figure 3.1: A dialog to collect values for a  $t$  test.

Our layout is basic. Here we pack a label frame into the window to give the dialog a nicer look. We will use the `tkgrid` geometry manager below.

```
lf <- ttklabelframe(f, text="t.test()", padding=10)
tkpack(lf, expand=TRUE, fill="both", padx=5, pady=5)
```

This next function simplifies the task of adding a label.

```
putLabel <- function(parent, text, row, column) {
  label <- ttklabel(parent, text=text)
  tkgrid(label, row=row, column=column, sticky="e")
}
```

Our first widget will be one to select a data frame. For this, a combobox is used, although if a large number of data frames are a possibility, a different widget may be better suited. The `getDfs` function is not shown, but simply returns the names of all data frames in the global environment. Also not shown are two similar calls to create comboboxes `xCombo` and `fCombo` which allow the user to specify parts of a formula.

```
putLabel(lf, "data:", 0, 0)
dataCombo <- ttkcombobox(lf, values=getDfs(), textvariable=e$data)
tkgrid(dataCombo, row=0, column=1, sticky="ew", padx=2)
tkfocus(dataCombo) # give focus
```

The combobox may not be the most natural widget to gather a numeric value for the mean when the data is continuous, but at this point we haven't quite yet discussed the `ttkentry` widget.

```
putLabel(lf, "mu:", 2, 0)
muCombo <- ttkcombobox(lf, values=c(""), textvariable=e$mu)
tkgrid(muCombo, row=2, column=1, sticky="ew", padx=2)
```

The selection of an alternative hypothesis is a natural choice for a combobox, but, as this alternative is available in `tcltk`, we use a spin box with `wrap=TRUE`.

```
putLabel(lf, "alternative:", 3, 0)
altCombo <- tkspinbox(lf, values=c("two.sided","less","greater"),
                      textvariable=e$alternative, wrap=TRUE)
tkgrid(altCombo, row=3, column=1, sticky="ew", padx=2)
```

Here we use two widgets to specify the confidence level. The slider is quicker to use, but less precise than the spinbox. By sharing a text variable, the widgets are automatically synchronized.

```
putLabel(lf, "conf.level:", 3, 2)
altFrame <- ttkframe(lf)
tkgrid(altFrame, row=3, column=3, columnspan=2,
       sticky="ew", padx=2)
altScale <- ttkscale(altFrame, from=75, to=100,
                    variable=e$conf.level)
tkpack(altScale, expand=TRUE, fill="y", side="left")
altSpin <- tkspinbox(altFrame, from=75, to=100, increment=1,
                    textvariable=e$conf.level, width=5)
tkpack(altSpin, side="left")
```

A checkbox is used to set the binary variable for `var.equal`

```
putLabel(lf, "var.equal:", 4, 0)
veCheck <- ttkcheckboxbutton(lf, variable=e$var.equal)
tkgrid(veCheck, row=4, column=1, stick="w", padx=2)
```

When assigning grid weights, we don't want the labels (columns 0 and 2) to expand the same way we want the other columns to do, so we assign different weights.

```
tkgrid.columnconfigure(lf, 0, weight=1)
tkgrid.columnconfigure(lf, 1, weight=10)
tkgrid.columnconfigure(lf, 2, weight=1)
tkgrid.columnconfigure(lf, 3, weight=10)
```

The dialog has two control buttons we wish to include.

```
bf <- ttkframe(f)
tkpack(bf, fill="x", padx=5, pady=5)
cancel <- ttkbutton(bf, text="cancel", command=function() {
  tcl("after","cancel",updateID$ID)
  tkdestroy(w)
})
ok <- ttkbutton(bf, text="ok")
tkpack(ttklabel(bf, text=" "), expand=TRUE, fill="y",
       side="left") # add a spring
tkpack(cancel, padx=6, side="left")
tkpack(ok, padx=6, side="left")
tkconfigure(ok, default="active")
```

The ok button is made to look active. As such we should bind to the button click and "Return" signals. First we define the callback. The `runTTest`

### 3. TCL Tk: WIDGETS

---

function is not shown, but is written to make good use of the structure of the data store.

```
okCallback <- function() {  
  l <- lapply(e, tclvalue)  
  runTTest(l)  
}  
tkbind(ok, "<Button-1>", okCallback)  
tkbind(w, "<Return>", okCallback)      # for active binding
```

At this point, our GUI is complete, but we would like to have it reflect any changes to the underlying R environment that effect its display. A such, we define a function, `updateUI`, which does two basic things: it searches for new data frames and it adjusts the controls depending on the current state.

```
updateUI <- function() {  
  tkconfigure(dataCombo, values=getDfs())  
  dfName <- tclvalue(e$data)  
  
  if(dfName == "") {  
    tkconfigure(xCombo, state="disabled")  
  } else {  
    df <- get(dfName, envir=.GlobalEnv)  
    tkconfigure(xCombo, state="normal", values=getNumericVars(df))  
    if(! tclvalue(e$x) %in% getNumericVars(df))  
      tclvalue(e$x) <- ""  
  
    tkconfigure(fCombo, values=getTwoLevelFactor(df))  
    if(!tclvalue(e$f) %in% getTwoLevelFactor(df))  
      tclvalue(e$f) <- ""  
  }  
  
  tkconfigure(fCombo, state=  
    ifelse(tclvalue(e$x) == "", "disabled", "normal"))  
  
  if(tclvalue(e$f) == "") {  
    tkconfigure(muCombo, state="normal")  
    tkconfigure(veCheck, state="disabled")  
  } else {  
    tclvalue(e$mu) <- 0  
    tkconfigure(muCombo, state="disabled")  
    tkconfigure(veCheck, state="normal")  
  }  
}
```

We use the `after` command to repeat a function call every so often. We also define a flag to stop the polling if desired. When polling, we make sure to test for existence of the parent window.

```
updateID <- new.env()
```

```

updateID$flag <- TRUE
updateID$ID <- NA
repeatFun <- function() {
  if(updateID$flag && as.logical(tkwininfo("exists",w))) {
    updateUI()
    updateID$ID <- tcl("after", 1000, repeatFun)
  }
}
repeatFun()

```

## 3.2 Text widgets

Tk provides both single- and multi-line text entry widgets. The section describes both and introduces scrollbars which are often desired for multi-line text entry.

### Entry Widgets

The `ttkentry` constructor provides a single line text entry widget. The widget can be associated with a Tcl variable at construction to facilitate getting and setting the displayed values through its argument `textvariable`. The width of the widget can be adjusted at construction time through the `width` argument. This takes a value for the number of characters to be displayed, assuming average-width characters. The text alignment can be set through the `justify` argument taking values of "left" (the default), "right" and "center". For gathering passwords, the argument `show` can be used, such as with `show="*"`, to show asterisks in place of all the characters.

The following constructs a basic example

```

eVar <- tclVar("initial value")
e <- ttkentry(w, textvariable=eVar)
tkpack(e)

```

We can get and set values using the Tcl variable.

```
tclvalue(eVar)
```

```
[1] "initial value"
```

```
tclvalue(eVar) <- "set value"
```

The `get` command can also be used.

```
tkget(e)
```

```
<Tcl> set value
```

### 3. Tcl Tk: WIDGETS

---

**Indices** The entry widget uses an index to record the different positions within the entry box. This index can be a number (0-based), an *x*-coordinate of the value (@x), the values "end" and "insert" to refer to the end of the current text and the insert as set through the keyboard or mouse. The mouse can also be used to make a selection. In this case the indices "sel.first" and "sel.last" describe the selection.

With indices, we can insert text with the *ttkentry* insert command

```
tkinsert(e, "end", "new text")
```

Or, we can delete a range of text, in this case the first 4 characters, using *ttkentry* delete. The first value is the left most index to delete (0-based), the second value the index to the right of the last value deleted.

```
tkdelete(e, 0, 4) # e.g., a b c d e f -text
```

The *ttkentry* icursor command can be used to set the cursor position to the specified index.

```
tkicursor(e, 0) # move to beginning
```

Finally, we note that the selection can be adjusted using the *ttkentry* selection range subcommand. This takes two indices. Like delete, the first index specifies the first character of the selection, the second indicates the character to the right of the selection boundary. The following example would select all the text.

```
tkselection.range(e, 0, "end")
```

The *ttkentry* selection clear subcommand clears the selection and *ttkentry* selection present signals if a selection is currently made.

**Events** Several useful events include <KeyPress> and <KeyRelease> for a key presses and <FocusIn> and <FocusOut> for focus events.

#### Example 3.2: Using validation for dates

There is no native calendar widget in *tcltk*. This example shows how one can use the validation framework for entry widgets to check that user-entered dates conform to an expected format.

Validation happens in a few steps. A validation command is assigned to some event. This call can come in two forms. Prevalidation is when a change is validated prior to being committed, for example when each key is pressed. Revalidation is when the value is checked after it is sent to be committed, say when the entry widget loses focus or the enter key is pressed.

When a validation command is called it should check whether the current state of the entry widget is valid or not. If valid, it returns a value of TRUE and FALSE otherwise. These need to be Tcl Boolean values, so in the following, the



command `tcl("eval","TRUE")` (or `tcl("eval", "FALSE")`) is used. If the validation command returns `FALSE`, then a subsequent call to the specified invalidation command is made.

For each callback, a number of substitution values are possible, in addition to the standard ones such as `W` to refer to the widget. These are: `d` for the type of validation being done: 1 for insert prevalidation, 0 for delete prevalidation, or -1 for revalidation; `i` for the index of the string to be inserted or deleted or -1; `P` for the new value if the edit is accepted (in prevalidation) or the current value in revalidation; `s` for the value prior to editing; `S` for the string being inserted or deleted, `v` for the current value of `validate` and `V` for the condition that triggered the callback.

In the following callback definition we use `W` so that we can change the entry text color to black and format the data in a standard manner and `P` to get the entry widget's value just prior to validations.

To begin, we define some patterns for acceptable date formats.

```
datePatterns <- c()
for(i in list(c("%m","%d","%Y"),      # U.S. style
              c("%m","%d","%y"))) {
  for(j in c("/", "- ", " "))
    datePatterns[length(datePatterns)+1] <-
      paste(i, sep=" ", collapse=j)
}
```

Our callbacks set the color to black or red, depending on whether we have a valid date. First our validation command.

```
isValidDate <- function(W, P) { # P is the current value
  for(i in datePatterns) {
    date <- try( as.Date(P, format=i), silent=TRUE)
    if(!inherits(date, "try-error") && !is.na(date)) {
      tkconfigure(W, foreground="black") # consult style?
      tkdelete(W, "0", "end")
      tkinsert(W, 0, format(date, format="%m/%d/%y"))
      return(tcl("expr", "TRUE"))
    }
  }
  return(tcl("expr", "FALSE"))
}
```

This is our invalid command.

```
indicateInvalidDate <- function(W) {
  tkconfigure(W, foreground="red")
  tcl("expr", "TRUE")
}
```

The `validate` argument is used to specify when the validation command should be called. This can be a value of `"none"` for validation when called

### 3. TCL Tk: WIDGETS

---

through the validation command; "key" for each key press; "focusin" for when the widget receives the focus; "focusout" for when it loses focus; "focus" for both of the previous; and "all" for any of the previous. We use "focusout" below, so also give a button widget so that the focus can be set elsewhere. (As usual, f is a parent frame.)

```
e <- ttkentry(f, validate="focusout",
              validatecommand=isValidDate,
              invalidcommand=indicateInvalidDate)
tkpack(e, side="left")
b <- ttkbutton(f, text="click")           # something to focus on
tkpack(b, side="bottom")
```

#### Scrollbars

Tk has several scrollable widgets – those that use scrollbars. Widgets which accept a scrollbar (without too many extra steps) have the options `xscrollcommand` and `yscrollcommand`. To use scrollbars in tcltk requires two steps: the scrollbars must be constructed and bound to some widget, and that widget must be told it has a scrollbar. This way changes to the widget can update the scrollbar and vice versa. Suppose, parent is a container and widget has these options, then the following will set up both horizontal and vertical scrollbars.

The scrollbars are defined as follows using the `orient` option and a command of the following form.

```
xscr <- ttkscrollbar(parent, orient="horizontal",
                    command=function(...) tkxview(widget, ...))
yscr <- ttkscrollbar(parent, orient="vertical",
                    command=function(...) tkxview(widget, ...))
```

The view commands set what part of the widget is being shown.

To link the widget back to the scrollbar, the `set` command is used in a callback to the scroll command. For this example we configure the options after the widget is constructed, but this can be done at the time of construction as well. Again, the command takes a standard form:

```
tkconfigure(widget,
            xscrollcommand=function(...) tkset(xscr,...),
            yscrollcommand=function(...) tkset(yscr,...))
```

Although scrollbars can appear anywhere, the conventional place is on the right and lower side of the parent. The following adds scrollbars using the grid manager. The combination of weights and stickiness below will have the scrollbars expand as expected if the window is resized.

```
tkgrid(widget, row=0, column=0, sticky="news")
tkgrid(yscr, row=0, column=1, sticky="ns")
tkgrid(xscr, row=1, column=0, sticky="ew")
```

```
tkgrid.columnconfigure(parent, 0, weight=1)
tkgrid.rowconfigure(parent, 0, weight=1)
```

Although a bit tedious, this gives the programmer some flexibility in arranging scrollbars. To avoid doing all this in the sequel, we turn the above into function `addScrollbars` (not shown).

## Multi-line Text Widgets

The `tktext` widget creates a multi-line text editing widget. If constructed with no options but a parent container, the widget can have text entered into it by the user.

The text widget is not a themed widget, hence has numerous arguments to adjust its appearance. We mention a few here and leave the rest to be discovered in the manual page (along with much else). The argument `width` and `height` are there to set the initial size, with values specifying number of characters and number of lines (not pixels). The actual size is font dependent, with the default for 80 by 24 characters. The `wrap` argument, with a value from "none", "char", or "word", indicates if wrapping is to occur and if so, does it happen at any character or only a word boundary. The argument `undo` takes a logical value indicating if the undo mechanism should be used. If so, the subcommand `tktext edit` can be used to undo a change (or the control-z keyboard combination).

**Indices** As with the entry widget, several commands take indices to specify position within the text buffer. Only for the multi-line widget both a line and character are needed in some instances. These indices may be specified in many ways. One can use row and character numbers separated by a period in the pattern `line.char`. The line is 1-based, the column 0-based (e.g., 1.0 says start on the 1st row and first character). In general, one can specify any line number and character on that line, with the keyword `end` used to refer to the last character on the line. Text buffers may carry transient marks, in which case the use of this mark indicates the next character after the mark. Predefined marks include `end`, to specify the end of the buffer, `insert`, to track the insertion point in the text buffer were the user to begin typing, and `current`, which follows the character closest to the mouse position. As well, pieces of text may be tagged. The format `tag.first` and `tag.last` index the range of the tag `tag`. Marks and tags are described below. If the *x-y* position of the spot is known (through percent substitutions say) the index can be specified by position, as *x,y*.

Indices can also be adjusted relative to the above specifications. This adjustment can be by a number of characters (`chars`), index positions (`indices`) or lines. For example, `insert + 1 lines` refers to 1 line under the insert point. The values `linestart`, `lineend`, `wordstart` and `wordend` are also avail-

### 3. TCL Tk: WIDGETS

---

able. For instance, `insert linestart` is the beginning of the line from the insert point, while `end -1 wordstart` and `end - 1 chars wordend` refer to the beginning and ending of the last word in the buffer. (The end index refers to the character just after the new line so we go back 2 steps.)

**Getting text** The *tktext* `get` subcommand is used to retrieve the text in the buffer. Coercion to character should be done with `tclvalue` and not `as.character` to preserve the distinction between spaces and line breaks.

```
value <- tkget(t, "1.0", "end")
as.character(value)                # wrong way
```

```
character(0)
```

```
tclvalue(value)
```

```
[1] "\n"
```

**Inserting text** Inserting text can be done through the *tktext* `insert` subcommand by specifying first the index then the text to add. One can use `\n` to add new lines.

```
tkinsert(t, "end", "more text\n new line")
```

Images and other windows can be added to a text buffer, but we do not discuss that here.

The buffer can have its contents cleared using `tkdelete`, as with `tkdelete(t, "0.0", "end")`.

**Panning the buffer: tksee** After text is inserted, the visible part of buffer may not be what is desired. The *tktext* `see` sub command is used to position the buffer on the specified index, its lone argument.

**tags** Tags are a means to assign a name to characters within the text buffer. Tags may be used to adjust the foreground, background and font properties of the tagged characters from those specified globally at the time of construction of the widget, or configured thereafter. Tags can be set when the text is inserted, as with

```
tkinsert(t, "end", "last words", "lastWords") # lastWords is tag
```

Tags can be set after the text is added through the *tktext* `tag add` subcommand using indices to specify location. The following marks the first word:

```
tktag.add(t, "firstWord", "1.0 wordstart", "1.0 wordend")
```

The *tktext* `tag configure` can be used to configure properties of the tagged characters, for example:

```
tktag.configure(t, "firstWord", foreground="red",
               font="helvetica 12 bold")
```

There are several other configuration options for a tag. A cryptic list can be produced by calling the subcommand *tktext tag configure* without a value for configuration.

**selection** The current selection, if any, is indicated by the `sel` tag, with `sel.first` and `sel.last` providing indices to refer to the selection. (Provided the option `exportSelection` was not modified.) These tags can be used with `tkget` to retrieve the currently selected text. An error will be thrown if there is no current selection. To check if there is a current selection, the following may be used:

```
hasSelection <- function(W) {
  ranges <- tclvalue(tcl(W, "tag", "ranges", "sel"))
  length(ranges) > 1 || ranges != ""
}
```

The cut, copy and paste commands are implemented through the functions `tk_textCut`, `tk_textCopy` and `tk_textPaste`. Their lone argument is the text widget. These work with the current selection and insert point. For example to cut the current selection, one has

```
tcl("tk_textCut", t)
```

**marks** Tags mark characters within a buffer, marks denote positions within a buffer that can be modified. For example, the marks `insert` and `current` refer to the position of the cursor and the current position of the mouse. Such information can be used to provide context-sensitive popup menus, as in this code example:

```
popupContext <- function(W, x, y) {
  ## or use sprintf("@%s,%s", x, y) for "current"
  cur <- tkget(W, "current wordstart", "current wordend")
  cur <- tclvalue(cur)
  popupContextMenuFor(cur, x, y)      # some function
}
```

To assign a new mark, one uses the *tktext* `mark set` subcommand specifying a name and a position through an index. Marks refer to spaces within characters. The gravity of the mark can be left or right. When right (the default), new text inserted is to the left of the mark. For instance, to keep track of an initial insert point and the current one, the initial point (marked `leftlimit` below) can be marked with

### 3. TCL Tk: WIDGETS

---

```
tkmark.set(t,"leftlimit","insert")
tkmark.gravity(t,"leftlimit","left")    # keep onleft
tkinsert(t,"insert","new text")
tkget(t, "leftlimit", "insert")
```

<Tcl> new text

The use of the subcommand *tktext* mark gravity is done so that the mark attaches to the left-most character at the insert point. The rightmost one changes as more text is inserted, so would make a poor choice.

**The edit command** The subcommand *tktext* edit can be used to undo text. As well, it can be used to test if the buffer has been modified, as follows:

```
tcl(t, "edit", "undo")                # no output
tcl(t, "edit", "modified")             # 1 = TRUE
```

<Tcl> 1

**Events** The text widget has a few important events. The widget defines virtual events <<Modified>> and <<Selection>> indicating when the buffer is modified or the selection is changed. Like the single-line text widget, the events <KeyPress> and <KeyRelease> indicate key activity. The %-substitution *k* gives the keycode and *K* the key symbol as a string (*N* is the decimal number).

#### Example 3.3: Displaying commands in a text buffer

This example shows how a text buffer can be used to display the output of R commands, using an approach modified from Sweave.

```
## create formatting tags
tktag.configure(t, "commandTag", foreground="blue",
                font="courier 12 italic")
tktag.configure(t, "outputTag", font="courier 12")
tktag.configure(t, "errorTag", foreground="red",
                font="courier 12 bold")
```

The following function does the work of evaluating a command chunk then inserting the values into the text buffer, using the different markup tags specified above to indicate commands from output.

```
evalCmdChunk <- function(t, cmds) {

  cmdChunks <- try(parse(text=cmds), silent=TRUE)
  if(inherits(cmdChunks,"try-error")) {
    tkinsert(t, "end", "Error", "errorTag") # add tag for markup
  }
}
```

```

for(cmd in cmdChunks) {
  dcmd <- deparse(cmd, width.cutoff = 0.75 * getOption("width"))
  command <-
    paste(getOption("prompt"),
          paste(dcmd, collapse=paste("\n", getOption("continue")),
                sep="")),
          sep="", collapse="")
  tkinsert(t, "end", command, "commandTag")
  tkinsert(t, "end", "\n")
  ## output, should check for errors in eval!
  output <- capture.output(eval(cmd, envir=.GlobalEnv))
  output <- paste(output, collapse="\n")
  tkinsert(t, "end", output, "outputTag")
  tkinsert(t, "end", "\n")
}
}

```

We envision this as a piece of a larger GUI which generates the commands to evaluate. For this example though, we make a simple GUI.

```

w <- tktoplevel(); tkwm.title(w, "Text buffer example")
f <- ttkframe(w, padding=c(3,3,3,12))
tkpack(f, expand=TRUE, fill="both")
t <- tktext(f, width=80, height = 24) # default size
addScrollbars(f, t)

```

This is how it can be used.

```
evalCmdChunk(t, "2 + 2; lm(mpg ~ wt, data=mtcars)")
```

### 3.3 Treeview widget

The themed treeview widget can be used to display rectangular data, like a data frame, or heirarchical data. The usage is similar for each beyond the need to indicate the heirarchical structure of a tree.

#### Rectangular data

Rectangular data has a row and column structure. In R, data frames are internally kept in terms of their columns which all must have the same type. The treeview widget is different, it stores all data as character data and one interacts with the data row by row.

The `ttktreeview` constructor creates the tree widget. There is no separate model for this widget, but there is a means to filter what is displayed. The argument `columns` is used to specify internal names for the columns and indicate the number of columns. A value of `1:n` will work here unless explicit names are desired. The argument `displaycolumns` is used to control which of

### 3. TCL Tk: WIDGETS

---

the columns are actually display. The default is "all", but a vector of indices or names can be given. The size of the widget is specified two different ways. The `height` argument is used to adjust the number of visible rows. The width of the widget is determined by the combined widths of each column, whose adjustments are mentioned later. The user may select one or more rows with the mouse, as controlled by the argument `selectmode`. Multiple rows may be selected with the default value of "extended", a restriction to a single row is specified with "browse", and no selection is possible if this is given as none. The treeview widget has an initial column for showing the tree-like aspect with the data. This column is referenced by #0. The `show` argument controls whether this column is shown. A value of "tree" leaves just this column shown, "headings" will show the other columns, but not the first, and the combined value of "tree headings" will display both (the default). Additionally, the treeview is a scrollable widget, so has the arguments `xscrollcommand` and `yscrollcommand` for specifying scrollbars.

If `f` is a frame, then the following call will create a widget with just one column showing 25 rows, like the older, non-themed, listbox widget of Tk.

```
tr <- ttktreeview(f,
                  columns=1,           # column identifier is "1"
                  show="headings",     # not "#0"
                  height=25)
addScrollbars(f, tr)                  # scrollbar function
```

**Column properties** Once the widget is constructed, its columns can be configured on a per-column basis. Columns can be referred to by the name specified through the `columns` argument or by number starting at 1 with "#0" referring to the tree column. The column headings can be set through the `ttktreeview` heading subcommand. The heading, similar to the button widget, can be text, an image or both. The text placement of the heading may be positioned through the `anchor` option. For example, this command will center the text heading of the first column:

```
tcl(tr, "heading", 1, text="Host", anchor="center")
```

The `ttktreeview` column subcommand can be used to adjust a column's properties including the size of the column. The option `width` is used to specify the pixel width of the column (the default is large); As the widget may be resized, one can specify the minimum column width through the option `minwidth`. When more space is allocated to the tree widget, than is requested by the columns, column with a TRUE value specified to the option `stretch` are resized to fill the available space. Within each column, the placement of each entry within a cell is controlled by the `anchor` option, using the compass points.

For example, this command will adjust properties of the lone column of `tr`:



```
tcl(tr, "column", 1, width=400, stretch=TRUE, anchor="w")
```

**Adding values** Values can be added to the widget through the *ttktreeview* *insert parent item [text] [values]* subcommand. This requires the specification of a parent (always "" for rectangular data) and an index for specifying the location of the new child amongst the previous children. The special value "end" indicates placement after all other children, as would a number larger than the number of children. A value of 0 or a negative value would put it at the beginning.

There are a number of options for each row. If column #0 is present, the text option is used to specify the text for the tree row and the option image can be given to specify an image to place to the left of the text value. For filling in the columns the values option is used. If there is a single column, like the current example, care needs to be taken that values separated by spaces are quoted (or in braces), otherwise, they will be split on spaces and treated like a vector of values truncated on the first one. Finally, we mention that tag option for insert that can be used to specify a tag for the inserted row. This allows the use of the subcommand *ttktreeview* tag configure to configure the foreground color, background color, font or image of an item.

In the example this is how we can add a list of possible CRAN mirrors to the treeview display.

```
x <- getCRANmirrors()
Host <- paste("'", x$Host, "'", sep="") # add quotes!
shade <- c("none", "gray") # tag names
for(i in 1:length(Host))
  ID <- tkinset(tr, "", "end", values=Host[i],
               tag=shade[i % 2]) # none or gray
tktag.configure(tr, "gray", background="gray95") # shade rows
```

**Item IDs** Each row has a unique item ID generated by the widget when a row is added. The base ID is "" (why this is specified for the value of parent for rectangular data). For rectangular displays, the list of all IDs may be found through the *ttktreeview* children sub command, which we will describe in the next section. Here we see it used to find the children of the root. As well, we show how the *ttktreeview* index command returns the row index.

```
children <- tcl(tr, "children", "")
(children <- head(as.character(children))) # as.character
```

```
[1] "I001" "I002" "I003" "I004" "I005" "I006"
```

```
sapply(children, function(i) tclvalue(tkindex(tr, i)))
```

### 3. TCL Tk: WIDGETS

---

```
I001 I002 I003 I004 I005 I006
"0"  "1"  "2"  "3"  "4"  "5"
```

**Retreiving values** The *ttktreeview* item subcommand can be used to get the values and other properties stored for each row. One specifies the item and the corresponding option:

```
x <- tcl(tr, "item", children[1], "-values") # no tkitem
as.character(x)
```

```
[1] "Patan.com.ar, Buenos Aires"
```

The value returned from the item command can be difficult to parse, as Tcl introduces braces for grouping. The coercion through `as.character` works much better at extracting the individual columns. A possible alternative to using the item command, is to instead keep the original data frame and use the index of the item to extract the value from the original.

**Moving and deleting items** The *ttktreeview* move subcommand can be used to replace a child. As with the insert command, a parent and an index for where the new child is to go among the existing children is given. The item to be moved is referred to by its ID. The *ttktreeview* delete and *ttktreeview* detach can be used to remove an item from the display, as specified by its ID. The latter command allows for the item to be reinserted at a later time.

**Events and callbacks** In addition to the keyboard events `<KeyPress>` and `<KeyRelease>` and the mouse events `<ButtonPress>`, `<ButtonRelease>` and `<Motion>`, the virtual event `<<TreeviewSelect>>` is generated when the selection changes. The current selection marks 0, 1 or more than 1 items if "extended" is given for the selectmode argument. The *ttktreeview* selection command will return the current selection. If converted to a string using `as.character` this will be a 0-length character vector, or a character vector of the selected item IDs. Further subcommands `set`, `add`, `remove`, and `toggle` can be used to adjust the selection programatically.

Within a key or mouse event callback, the selected column and row can be identified by position, as illustrated in this example callback.

```
callbackExample <- function(W, x, y) {
  col <- as.character(tkidentify(W, "column", x, y))
  row <- as.character(tkidentify(W, "row", x, y))
  ## do something ...
}
```

#### Example 3.4: Filtering a table

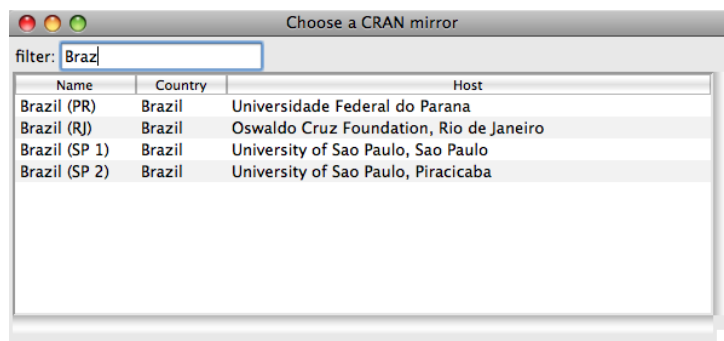


Figure 3.2: Using `ttktreeview` to show various CRAN sites. This illustration adds a search-like box to filter what repositories are displayed for selection.

We illustrate the above with a slightly improved GUI for selecting a CRAN mirror. This adds in a text box to filter the possibly large display of items to avoid scrolling through a long list.

```
df <- getCRANmirrors()[ , c(1,2,5,4)]
```

We use a text entry widget to allow the user to filter the values in the display as the user types.

```
f0 <- ttkframe(f); tkpack(f0, fill="x")
l <- ttklabel(f0, text="filter:"); tkpack(l, side="left")
filterVar <- tclVar("")
filterEntry <- ttkentry(f0, textvariable=filterVar)
tkpack(filterEntry, side="left")
```

The treeview will only show the first three columns of the data frame, although we store the fourth which contains the URL.

```
f1 <- ttkframe(f); tkpack(f1, expand=TRUE, fill="both")
tr <- ttktreeview(f1, columns=1:ncol(df),
                  displaycolumns = 1:(ncol(df) - 1),
                  show = "headings",      # not "tree"
                  selectmode = "browse") # single selection
addScrollbars(f1, tr)
```

We configure the column widths and titles as follows:

```
widths <- c(100, 75, 400)          # hard coded
nms <- names(df)
for(i in 1:3) {
  tcl(tr, "heading", i, text=nms[i])
  tcl(tr, "column", i, width=widths[i], stretch=TRUE, anchor="w")
}
```

### 3. TCL Tk: WIDGETS

---

This following helper function is used to fill in the widget with values from a data frame.

```
fillTable <- function(tr, df) {
  children <- as.character(tcl(tr, "children", ""))
  for(i in children) tcl(tr, "delete", i)
  shade <- c("none", "gray")
  for(i in seq_len(nrow(df)))
    tcl(tr, "insert", "", "end", tag=shade[i %% 2], text="",
        values=unlist(df[i,]))
  tktag.configure(tr, "gray", background="gray95")
}
```

The initial call populates the table from the entire data frame.

```
fillTable(tr, df)
```

The filter works by grepping the user input against the host value. We bind to `<KeyRelease>` (and not `<KeyPress>`) so we capture the last keystroke.

```
curInd <- 1:nrow(df)
tkbind(filterEntry, "<KeyRelease>", function(W, K) {
  val <- tclvalue(tkget(W))
  possVals <- apply(df, 1, function(...) paste(..., collapse=" "))
  ind <- grep(val, possVals)
  if(length(ind) == 0) ind <- 1:nrow(df)
  fillTable(tr, df[ind,])
})
```

This binding is for capturing a user's selection through a double-click event. In the callback, we set the CRAN option then withdraw the window.

```
tkbind(tr, "<Double-Button-1>", function(W, x, y) {
  sel <- as.character(tcl(W, "identify", "row", x, y))
  vals <- tcl(W, "item", sel, "-values")
  URL <- as.character(vals)[4] # not tclvalue
  repos <- getOption("repos")
  repos["CRAN"] <- gsub("/$", "", URL[1L])
  options(repos = repos)
  tkwm.withdraw(tkwininfo("toplevel", W))
})
```

**Editing cells of a table** There is no native widget for editing the cells of tabular data, as is provided by the `edit` method for data frames. The `tktable` widget (<http://tktable.sourceforge.net/>) provides such an add-on to the base Tk. We don't illustrate its usage here, as we keep to the core set of functions provided by Tk. However, we note that the `gdf` function of `gWidgetstcltk` provides an example of how it can be used.

## Heirarchical data

Specifying tree-like or heirarchical data is nearly identical to specifying rectangular data for the `ttktreeview` widget. The widget provides column #0 to display this extra structure. If an item, except the root, has children, a trigger icon to expand the tree is shown. This is in addition to any text and/or an icon that is specified. Children are displayed in an indented manner to indicate the level of ancestry they have relative to the root. To insert heirarchical data in to the widget the same `ttktreeview insert` subcommand is used, only instead of using the root item, "", as the parent item, one uses the item ID corresponding to the desired parent. If the option `open=TRUE` is specified to the `insert` subcommand, the children of the item will appear, if `FALSE`, the user can click the trigger icon to see the children. The programmer can use the `ttktreeview item` to configure this state. When the parent item is opened or closed, the virtual events `<<TreeviewOpen>>` and `<<TreeviewClose>>` will be signaled.

**Traversal** Once a tree is constructed, the programmer can traverse through the items using the subcommands `ttktreeview parent item` to get the ID for the parent of the item; `ttktreeview prev item` and `ttktreeview next item` to get the immediate siblings of the item; and `ttktreeview children item` to return the children of the item. Again, the latter one will produce a character vector of IDs for the children when coerced to character with `as.character`.

### Example 3.5: Using the treeview widget to show an XML file

This example shows how to display the heirarchical structure of an XML document using the tree widget.

We use the XML library to parse a document from the internet. This example uses just a few functions from this library: The `(htmlTreeParse)` (similar to `xmlInternalTreeParse`) to parse the file, `xmlRoot` to find the base node, `xmlName` to get the name of a node, `xmlValue` to get an associated value, and `xmlChildren` to return any child nodes of a node.

```
library(XML)
fileName <- "http://www.omegahat.org/RFXML/shortIntro.html"
QT <- function(...) {} # quiet next call
doc <- htmlTreeParse(fileName, useInternalNodes=TRUE, error=QT)
root <- xmlRoot(doc)
```

Our GUI is primitive, with just a treeview instance added.

```
tr <- ttktreeview(f, displaycolumns="#all", columns=1)
addScrollbars(f, tr)
```

We configure our columns headers and set a minimum width below. Recall, the tree column is designated "#0".

```
tcl(tr, "heading", "#0", text="Name")
```

### 3. TCL Tk: WIDGETS

---

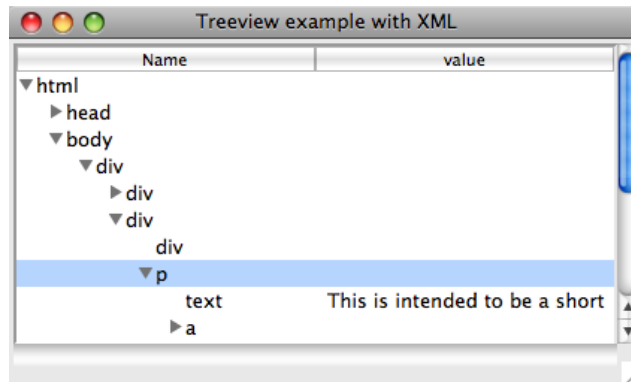


Figure 3.3: Illustration of using `ttktreeview` widget to show heirarchical data returned from parsing an HTML document with the XML package.

```
tcl(tr, "column", "#0", minwidth=20)
tcl(tr, "heading", 1, text="value")
tcl(tr, "column", 1, minwidth=20)
```

To map the tree-like structure of the XML document into the widget, we define the following function to recursively add to the treeview instance. We only add to the value column (through the `values` option) when the node does not have children. We use `do.call`, as a convenience, to avoid constructing two different calls to the `insert` subcommand. The `quoteIt` function used is not shown, but similar to `shQuote` only escaping with double quotes, as single quotes are treated differently by Tcl. (Otherwise the `ttktreeview` widget will split values on spaces.)

```
insertChild <- function(tr, node, parent="") {
  l <- list(tr, "insert", parent, "end", text=xmlName(node))
  children <- xmlChildren(node)
  if(length(children) == 0) {
    # add in values
    values <- paste(xmlValue(node), sep=" ", collapse=" ")
    values <- gsub("\n", " ", values) # treeview doesn't like
    values <- quoteIt(values)        # \n and spaces
    l$values <- values
  }
  treePath <- do.call("tcl", l)

  if(length(children)) # recurse
    for(i in children) insertChild(tr, i, treePath)
}
insertChild(tr, root)
```

At this point, the GUI will allow one to explore the structure of the XML file. We continue this example to show two things of general interest, but are

really artificial for this example.

**Drag and drop** First, we show how one might introduce drag and drop to rearrange the rows. We begin by defining two global variables that store the row that is being dragged and a flag to indicate if a drag event is ongoing.

```
.selectedID <- ""                                # globals
.dragging <- FALSE
```

We provide callbacks for three events: a mouse click, mouse motion and mouse release. This first callback sets the selected row on a mouse click.

```
tkbind(tr, "<Button-1>", function(W,x,y) {
    .selectedID <<- as.character(tcl(W, "identify","row", x, y))
})
```

The motion callback configures the cursor to indicate a drag event and sets the dragging flag. One might also put in code to highlight any drop areas.

```
tkbind(tr, "<B1-Motion>", function(W, x, y, X, Y) {
    tkconfigure(W, cursor="diamond_cross")
    .dragging <<-TRUE
})
```

When the mouse button is released we check that the widget we are over is indeed the tree widget. If so, we then move the rows. One can't move a parent to be a child of its own children, so we wrap the *ttktreeview* move subcommand within try. The move command places the new value as the first child of the item it is being dropped on. If a different action is desired, the "0" below would need to be modified.

```
tkbind(tr, "<ButtonRelease-1>", function(W, x, y, X, Y) {
    if(.dragging && .selectedID != "") {
        w = tkwinfo("containing", X, Y)
        if(as.character(w) == as.character(W)) {
            dropID <- as.character(tcl(W, "identify","row", x, y))
            try(tkmove(W, .selectedID, dropID, "0"), silent=TRUE)
        }
    }
    .dragging <<- FALSE; .selectedID <<- "" # reset
})
```

**Walking the tree** Our last item of general interest is a function that shows one way to walk the structure of the treeview widget to generate a list representing the structure of the data. A potential use of this might be to allow a user to rearrange an XML document through drag and drop. The subcommand *ttktreeview* children proves useful here, as it is used to identify the heirarchical structure. When there are children a recursive call is made.

```
treeToList <- function(tr) {
  l <- list()
  walkTree <- function(child, l) {
    l$name <- tclvalue(tcl(tr,"item", child, "-text"))
    l$value <- tclvalue(tcl(tr,"item", child, "-values"))
    children <- as.character(tcl(tr, "children", child))
    if(length(children)) {
      l$children <- list()
      for(i in children)
        l$children[[i]] <- walkTree(i, list()) # recurse
    }
    return(l)
  }
  l <- walkTree("",l)
  return(l)
}
```

#### 3.4 Menus

Menu bars and popup menus in Tk are constructed with `tkmenu`. The parent argument depends on what the menu is to do. A toplevel menu bar, such as appears at the top of a window has a toplevel window as its parent; a sub-menu of a menu bar uses the parent menu; and a popup menu uses a widget. The menu widget in Tk has an option to be “torn off.” This features was at one time common in GUIs, but now is rarely seen so it is recommended that this option be disabled. The `tearoff` option can be used at construction time to override the default behaviour. Otherwise, the following command will do so globally:

```
tcl("option","add","*tearOff", 0) # disable tearoff menus
```

A toplevel menu bar is attached to a top-level window using `tkconfigure` to set the menu property of the window. For the aqua Tk libraries for Mac OS X, this menu will appear on the top menu bar when the window has the focus. For other operating systems, it appears at the top of the window. For Mac OS X, a default menu bar with no relationship to your application will be shown if a menu is not provided for a toplevel window. Testing for native Mac OS X may be done via the following function:

```
usingMac <- function()
  as.character(tcl("tk", "windowingsystem")) == "aqua"
```

The `tkpopup` function facilitates the creation of a popup menu. This function has arguments for the menu bar, and the position where the menu should be popped up. For example, the following code will bind a popup menu, `pmb` (yet to be defined), to the right click event for a button `b`. As Mac OS X may



not have a third mouse button, and when it does it refers to it differently, the callback is bound conditionally to different events.

```
doPopup <- function(X, Y) tkpopup(pmb, X, Y) # define call back
if (usingMac()) {
  tkbind(b, "<Button-2>", doPopup)      # right click
  tkbind(b, "<Control-1>", doPopup)    # Control + click
} else {
  tkbind(b, "<Button-3>", doPopup)
}
```

**Adding submenus and action items** Menus shows a heirarchical view of action items. Items are added to a menu through the *tkmenu* add subcommand. The nested structure of menus is achieved by specifying a *tkmenu* object as an item. The *tkmenu* add cascade subcommand is used for this. The option *label* is used to label the menu and the menu option to specify the sub-menu.

Grouping of similar items can be done through nesting, or on occasion through visual separation. The latter is implemented with the *tkmenu* add separator subcommand.

There are a few different types of action items that can be added.

An action item is one associated with a command. The simplest case is a label in the menu that activates a command when selected through the mouse. The *tkmenu* add command (through *tkadd*(widget, "command", ...)) allows one to specify a label, a command and optionally an image with a value for *compound* to adjust its layout. (Images are not shown in Mac OS X.) Action commands may possibly be called for different widgets, so the use of percent substitution is discouraged here. One can also specify that a keyboard accelerator be displayed through the option *accelerator*, but a separate callback must listen for this combination.

Action items may also be checkboxes. To create one, the subcommand *tkmenu* add checkbutton is used. The available arguments include *label* to specify the text, *variable* to specify a tcl variable to store the state, *onvalue* and *offvalue* to specify the state to the tcl variable, and *command* to specify a call back when the checked state is toggled. The initial state is set by the value in the Tcl variable.

Additionally, action items may be radiobutton groups. These are specified with the subcommand *tkmenu* add radiobutton. The *label* option is used to identify the entry, *variable* to set a text variable and to group the buttons that are added, and *command* to specify a command when that entry is selected.

Action items can also be placed after an item, rather than at the end using the *tkmenu* insert command *index* subcommand. The *index* may be specified numerically with 0 being the first item for a menu. More conveniently

### 3. TCL Tk: WIDGETS

---

the index can be determined by specifying a pattern to match the menu's labels.

**Set state** The state option is used to retrieve and set the current state of the a menu item. This value is typically normal or disabled, the latter to indicate the item is not available. The state can be set when the item is added or configured after that fact through the *tkmenu* entryconfigure command. This function needs the menu bar specified and the item specified as an index or pattern to match the labels.

#### Example 3.6: Simple menu example

This example shows how one might make a very simple code editor using a text-entry widget. We use the *svMisc* package, as it defines a few GUI helpers which we use.

```
library(svMisc)                                # for some helpers
showCmd <- function(cmd) writeLine(captureAll(Parse(cmd)))
```

We create a simple GUI with a top-level window containing the text entry widget.

```
w <- tktoplevel()
tkwm.title(w, "Simple code editor")
f <- ttkframe(w, padding=c(3,3,3,12));
tkpack(f, expand=TRUE, fill="both")
tb <- tktext(f, undo=TRUE)
addScrollbars(f, tb)
```

We create a toplevel menu bar, *mb*, and attach it to our toplevel window. Then we create a file and edit submenu.

```
mb <- tkmenu(w); tkconfigure(w, menu=mb)
fileMenu <- tkmenu(mb)
tkadd(mb, "cascade", label="File", menu=fileMenu)
editMenu <- tkmenu(mb)
tkadd(mb, "cascade", label="Edit", menu=editMenu)
```

To these sub menu bars, we add action items. First a command to evaluate the contents of the buffer.

```
tkadd(fileMenu, "command", label="Evaluate buffer",
      command = function() {
        curVal <- tclvalue(tkget(tb, "1.0", "end"))
        showCmd(curVal)
      })
```

Then a command to evaluate just the current selection

```
tkadd(fileMenu, "command", label="Evaluate selection",
      state="disabled",
```

```

        command = function() {
            curSel <- tclvalue(tkget(tb, "sel.first", "sel.last"))
            showCmd(curSel)
        })

```

Finally, we end the file menu with a quit action.

```

tkadd(fileMenu, "separator")
tkadd(fileMenu, "command", label="Quit",
      command=function() tkdestroy(w))

```

The edit menu has an undo and redo item. For illustration purposes we add an icon to the undo item.

```

img <- system.file("images/up.gif", package="gWidgets")
QT <- tkimage.create("photo", "::img::undo",
                    file = img)
tkadd(editMenu, "command", label="Undo",
      image="::img::undo", compound="left",
      command = function() tcl(tb, "edit", "undo"))
tkadd(editMenu, "command", label="Redo",
      command = function() tcl(tb, "edit", "redo"))

```

We now define a function to update the user interface to reflect any changes.

```

updateUI <- function() {
    states <- c("disabled", "normal")
    ## selection
    hasSelection <- function(W) {
        ranges <- tclvalue(tcl(W, "tag", "ranges", "sel"))
        length(ranges) > 1 || ranges != ""
    }
    ## by index
    tkentryconfigure(fileMenu, 1, state=states[hasSelection(tb) + 1])
    ## undo — if buffer modified, assume undo stack possible
    ## redo — no good check for redo
    canUndo <- function(W) as.logical(tcl(W, "edit", "modified"))
    tkentryconfigure(editMenu, "Undo", state=states[canUndo(tb) + 1])
    tkentryconfigure(editMenu, "Redo", state=states[canUndo(tb) + 1])
}

```

We now add an accelerator entry to the menubar and a binding to the top-level window for the keyboard shortcut.

```

if(usingMac()) {
    tkentryconfigure(editMenu, "Undo", accelerator="Cmd-z")
    tkbind(w, "<Option-z>", function() tcl(tb, "edit", "undo"))
} else {
    tkentryconfigure(editMenu, "Undo", accelerator="Control-u")
    tkbind(w, "<Control-u>", function() tcl(tb, "edit", "undo"))
}

```

To illustrate popup menus, we define one within our text widget that will grab all functions that complete the current word, using the `CompletePlus` function from the `svMisc` package to find the completions. The use of `current wordstart` and `current wordend` to find the word at the insertion point isn't quite right for R, as it stops at periods.

```
doPopup <- function(W, X, Y) {
  cur <- tclvalue(tkget(W, "current wordstart",
                        "current wordend"))
  tcl(W, "tag", "add", "popup", "current wordstart",
                        "current wordend")
  posVals <- head(CompletePlus(cur)[,1, drop=TRUE], n=20)
  if(length(posVals) > 1) {
    popup <- tkmenu(tb) # create menu for popup
    sapply(posVals, function(i) {
      tkadd(popup, "command", label=i, command = function() {
        tcl(W,"replace", "popup.first", "popup.last", i)
      })
    })
    tkpopup(popup, X, Y)
  }
}
```

For a popup, we set the appropriate binding for the underlying windowing system. For the second mouse button binding in OS X, we clear the clipboard. Otherwise the text will be pasted in, as this mouse action already has a default binding for the text widget.

```
if (!usingMac()) {
  tkbind(tb, "<Button-3>", doPopup)
} else {
  tkbind(tb, "<Button-2>", function(W,X,Y) {
    ## UNIX legacy re mouse-2 click for selection copy
    tcl("clipboard", "clear", displayof=W)
    doPopup(W,X,Y)
  }) # right click
  tkbind(tb, "<Control-1>", doPopup) # Control + click
}
```

## 3.5 Canvas Widget

The canvas widget provides an area to display lines, shapes, images and widgets. Methods exist to create, move and delete these objects, allowing the canvas widget to be the basis for creating interactive GUIs. The constructor `tkcanvas` for the widget, being a non-themable widget, has many arguments. We mention the standard ones `width`, `height`, and `background`. Additionally, the canvas is a scrollable widget, so has the corresponding arguments `xscrollcommand` and `yscrollcommand`.

**The create command** The subcommand *tkcanvas create type [options]* is used to add new items to the canvas. The options vary with the type of the item. The basic shape types that one can add are "line", "arc", "polygon", "rectangle", and "oval". Their options specify the size using *x* and *y* coordinates. Other options allow one to specify colors, etc. The complete list is covered in the canvas manual page, which we refer the reader to, as the description is lengthy. In the examples, we show how to use the "line" type to display a graph and how to use the "oval" type to add a point to a canvas. Additionally, one can add text items through the "text" type. The first options are the *x* and *y* coordinates and the *text* option specifies the text. Other standard text options are possible (e.g., *font*, *justify*, *anchor*).

The type can also be an image object or a widget (a window object). Images are added by specifying an *x* and *y* position, possibly an anchor position, and a value for the "image" option and optionally, for state dependent display, specifying "activeimage" and "disabledimage" values. The "state" option is used to specify the current state. Window objects are added similarly in terms of their positioning, along with options for "width" and "height". The window itself is added through the "window" option. An example shows how to add a frame widget.

Once created, a screenshot of the canvas can be created through the *tkcanvas postscript* subcommand, as in `tcl(canvas, "postscript", file="filename")`. To store the widget so that it can be recreated is not supported directly. Tcl code to do so can be found at <http://wiki.tcl.tk/9168>.

**Items and tags** The *tkcanvas.create* function returns an item ID. This can be used to refer to the item at a later stage. Optionally, tags can be used to group items into common groups. The "tags" option can be used with *tkcreate* when the item is created, or the *tkcanvas addtag* subcommand can be used. The call `tkaddtag(canvas, tagName, "withtag", item)` would add the tag "tagName" to the item returned by *tkcreate*. (The "withtag" is one of several search specifications.) As well, if one is adding a tag through a mouse click, the call `tkaddtag(W, "tagName", "closest", x, y)` could be used with *W*, *x* and *y* coming from percent substitutions. Tags can be deleted through the *tkcanvas dtag tag* subcommand.

There are several subcommands that can be called on items as specified by a tag or item ID. For example, the *tkcanvas itemcget* and *tkcanvas itemconfigure* subcommands allow one to get and set options for a given item. The *tkcanvas delete tag\_or\_ID* subcommand can be used to delete an item. Items can be moved through the *tkcanvas move tag\_or\_ID x y* subcommand, where *x* and *y* specify the horizontal and vertical shift in pixels. The subcommand *tkcanvas coords tag\_or\_ID [coordinates]* allows one to respecify the coordinates for which the item was defined, thereby allowing the possibility of moving or resizing the object. Additionally, the *tkcanvas scale* can

### 3. TCL Tk: WIDGETS

---

be used to rescale items. If items overlap each other, except for windows, an item can be raised to the top through the *tkcanvas* *raise item\_or\_ID* subcommand.

**Bindings** Bindings can be specified overall for the canvas, as usual, through *tkbind*. However, bindings can also be set on specific items through the subcommand *tkcanvas bind tag\_or\_ID event function* which is aliased to *tkitembind*. This allows bindings to be placed on items sharing a tag name, without having the binding on all items. Only mouse, keyboard or virtual events can have such bindings.

#### Example 3.7: Using a canvas to make a scrollable frame

This example shows how to use a canvas widget to create a box container that scrolls when more items are added than will fit in the display area. The basic idea is that a frame is added to the canvas equipped with scrollbars using the *tkcanvas* *create window* subcommand. The binding to the *<Configure>* event updates the scrollregion of the canvas widget to include the entire canvas. This grows, as items are added to the frame. This is modified from an example found at <http://mail.python.org/pipermail/python-list/1999-June/005180.html>.

This constructor returns a box container that scrolls as more items are added. The parent passed in must use the grid manager for its children.

```
scrollableFrame <- function(parent, width= 300, height=300) {  
  canvasWidget <-  
    tkcanvas(parent,  
             borderwidth=0, highlightthickness=0,  
             background="#e3e3e3", # match themed widgets  
             width=width, height=height)  
  addScrollbars(parent, canvasWidget)  
  
  gp <- ttkframe(canvasWidget, padding=c(0,0,0,0))  
  gpID <- tkcreate(canvasWidget, "window", 0, 0, anchor="nw",  
                  window=gp)  
  
  tkbind(gp,"<Configure>",function() { # updates scrollregion  
    bbox <- tcl(canvasWidget, "bbox", "all")  
    tcl(canvasWidget,"config", scrollregion=bbox)  
  })  
  
  return(gp)  
}
```

To use it, we create a simple GUI as follows:

```
w <- tktoplevel()  
tkwm.title(w,"Scrollable frame example")
```

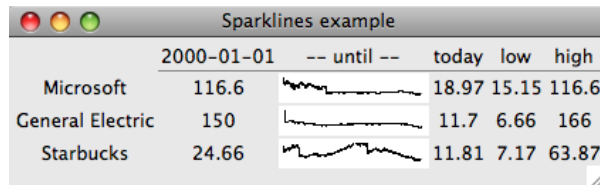


Figure 3.4: Example of embedding sparklines in a display organized using tkgrid. A tkcanvas widget is used to display the graph.

```
g <- ttkframe(w); tkpack(g, expand=TRUE, fill="both")
gp <- scrollableFrame(g, 300, 300)
```

To display a collection of available fonts requires a widget or container that could possibly show hundreds of similar values. The scrollable frame serves this purpose well (cf. Figure 1.2). The following shows how a label can be added to the frame whose font is the same as the label text. The available fonts are found from `tkfont.families` and the useful coercion to character by `as.character`.

```
fontFamilies <- as.character(tkfont.families())
## skip odd named ones
fontFamilies <- fontFamilies[grepl("^[[:alpha:]]", fontFamilies)]
for(i in 1:length(fontFamilies)) {
  fontName <- paste("tmp", i, sep="")
  try(tkfont.create(fontName, family=fontFamilies[i], size=14),
      silent=TRUE)
  l <- ttklabel(gp, text=fontFamilies[i], font=fontName)
  tkpack(l, side="top", anchor="w")
}
```

### Example 3.8: Using canvas objects to show sparklines

Edward Tufte, in his book *Beautiful Evidence*?, advocates for the use of *sparklines* – small, intense, simple datawords – to show substantial amounts of data in a small visual space. This example shows how to use a `ttkcanvas` object to display a sparkline graph using a line object. The example also uses `tkgrid` to layout the information in a table. We could have spent more time on the formatting of the numeric values and factoring out the data download, but leave improvements as an exercise.

This function simply shortens our call to `ttklabel`. We use the global `f` (a `ttkframe`) as the parent.

```
mL <- function(label) {
  if(is.numeric(label))
    label <- format(label, digits=4)
  ttklabel(f, text=label) # save some typing
}
```

### 3. TCL Tk: WIDGETS

---

```
}
```

We begin by making the table header along with a toprule.

```
tkgrid(mL(""), mL("2000-01-01"), mL("-- until --"),
      mL("today"), mL("low"), mL("high"))
tkgrid(ttkseparator(f), row=1, column=1, columnspan=5, sticky="we")
```

This function adds a sparkline to the table. We use financial data in this example, as we can conveniently employ the `get.hist.quote` function from the `tseries` package to get interesting data.

```
addSparkLine <- function(label, symbol="MSFT") {
  width <- 100; height=15 # fix width, height
  y <- get.hist.quote(instrument=symbol, start="2000-01-01",
                    quote="C", provider="yahoo",
                    retclass="zoo")$Close
  min <- min(y); max <- max(y)
  start <- y[1]; end <- tail(y,n=1)
  rng <- range(y)

  sparkLineCanvas <- tkcanvas(f, width=width, height=height)
  x <- 0:(length(y)-1) * width/length(y)
  if(diff(rng) != 0) {
    y1 <- (y - rng[1])/diff(rng) * height
    y1 <- height - y1 # adjust to canvas coordinates
  } else {
    y1 <- height/2 + 0 * y
  }
  ## make line with: pathName create line x1 y1... xn yn
  l <- list(sparkLineCanvas,"create","line")
  sapply(1:length(x), function(i) {
    l[[2*i + 2]] <- x[i]
    l[[2*i + 3]] <- y1[i]
  })
  do.call("tcl",l)

  tkgrid(mL(label),mL(start), sparkLineCanvas,
        mL(end), mL(min), mL(max), pady=1)
}
```

We can then add some rows to the table as follows:

```
addSparkLine("Microsoft","MSFT")
addSparkLine("General Electric", "GE")
addSparkLine("Starbucks","SBUX")
```

#### Example 3.9: Capturing mouse movements

This example is a stripped-down version of the `tkcanvas.R` demo that accompanies the `tcltk` package. That example shows a scatterplot with regression



line. The user can move the points around and see the effect this has on the scatterplot. Here we focus on the moving of an object on a canvas widget. We assume we have such a widget in the variable `canvas`.

This following adds a single point to the canvas using an oval object. We add the "point" tag to this item, for later use. Clearly, this code could be modified to add more points.

```
x <- 200; y <- 150; r <- 6
item <- tkcreate(canvas, "oval", x - r, y - r, x + r, y + r,
                    width=1, outline="black",
                    fill="SkyBlue2")
tkaddtag(canvas, "point", "withtag", item)
```

In order to indicate to the user that a point is active, in some sense, the following changes the fill color of the point when the mouse is over the point. We add this binding using `tkitembind` so that it will apply to all point items and only the point items.

```
tkitembind(canvas, "point", "<Any-Enter>", function()
    tkitemconfigure(canvas, "current", fill="red"))
tkitembind(canvas, "point", "<Any-Leave>", function()
    tkitemconfigure(canvas, "current", fill="SkyBlue2"))
```

There are two key bindings needed for movement of an object. First, we tag the point item that gets selected when a mouse clicks on a point and update the last position of the currently selected point.

```
lastPos <- numeric(2) # global to track position
tagSelected <- function(W, x, y) {
  tkaddtag(W, "selected", "withtag", "current")
  tkitemraise(W, "current")
  lastPos <- as.numeric(c(x, y))
}
tkitembind(canvas, "point", "<Button-1>", tagSelected)
```

When the mouse moves, we use `tkmove` to have the currently selected point move too. This is done by tracking the differences between the last position recorded and the current position and moving accordingly.

```
moveSelected <- function(W, x, y) {
  pos <- as.numeric(c(x,y))
  tkmove(W, "selected", pos[1] - lastPos[1],
        pos[2] - lastPos[2])

  lastPos <- pos
}
tkbind(canvas, "<B1-Motion>", moveSelected)
```

A further binding, for the `<ButtonRelease-1>` event, would be added to do something after the point is released. In the original example, the old regression line is deleted, and a new one drawn. Here we simply delete the "selected" tag.

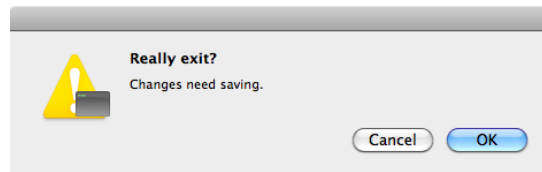


Figure 3.5: A basic modal dialog constructed by `tkmessageBox`.

```
tkbind(canvas, "<ButtonRelease-1>",  
       function(W) tkdtag(W, "selected"))
```

## 3.6 Dialogs

### Modal dialogs

The `tkmessageBox` constructor can be used to create simple modal dialogs allowing a user to confirm an action. This replaces the older `tkdialog` dialogs. The `tkmessageBox` dialogs use the native toolkit if possible. The arguments `title`, `message` and `detail` are used to set the text for the dialog. The title may not appear for all operating systems. A `messageBox` dialog has an `icon` argument. The default icon is "info" but could also be one of "error", "question" or "warning". The buttons used are specified by the `type` argument with values of "ok", "okcancel", "retrycancel", "yesno", or "yesnocancel". When a button is clicked the dialog is destroyed and the button label returned as a value. The argument `parent` can be given to specify which window the dialog belongs to. Depending on the operating system this may be used when drawing the dialog.

A sample usage is:

```
tkmessageBox(title="Confirm", message="Really exit?",  
             detail="Changes need saving.",  
             icon="question", type="okcancel")
```

If the default modal dialog is not enough – for instance there is no means to gather user input – then a `toplevel` window can be made modal. The `tkwait.window` will cause a top-level window to be modal and `tkgrab.release` will return the interactivity for the window.

### File and directory selection

Tk provides constructors for selecting a file, for selecting a directory or for specifying a filename when saving. These are implemented by `tkgetOpenFile`, `tkchooseDirectory`, and `tkgetSaveFile` respectively. Each of these can be called with no argument, and returns a `tclobj` that can be converted to a

character string with `tclvalue`. The value is empty when there is no selection made.

The dialog will appear related to a toplevel window if the argument `parent` is specified. The `initialdir` and `initialfile` can be used to specify the initial values in the dialog. The `defaulttextextension` argument can be used to specify a default extension for the file.

When browsing for files, it can be convenient to filter the available file types that can be selected. The `filetypes` argument is used for this task. However, the file types are specified using Tcl brace-notation, not R code. For example, to filter out various image types, one could have

```
tkgetOpenFile(filetypes = paste(
    "{{jpeg files} {.jpg .jpeg} }",
    "{{png files} {.png}}",
    "{{All files} {*}}", sep=" ") # needs space
```

Extending this is hopefully clear from the pattern above.

### Example 3.10: A “File” menu

To illustrate, a simple example for a file menu could be:

```
w <- tktoplevel(); tkwm.title(w, "File menu example")
mb <- tkmenu(w); tkconfigure(w, menu=mb)
fileMenu <- tkmenu(mb)
tkadd(mb, "cascade", label="File", menu=fileMenu)
tkadd(fileMenu, "command", label="Source file...",
      command= function() {
        fname <- tkgetOpenFile(fileTypes=
                               "{{R files} {.R}} {{All files} *}")
        source(tclvalue(fname))
      })
tkadd(fileMenu, "command", label="Save workspace as...",
      command=function() {
        fname <- tkgetSaveFile(defaulttextextension="Rsave")
        save.image(file=tclvalue(fname))
      })
tkadd(fileMenu, "command", label="Set working directory...",
      command=function() {
        fname <- tkchooseDirectory()
        setwd(tclvalue(fname))
      })
```

### Choosing a color

Tk provides the command `tk_chooseColor` to construct a dialog for selection of a color by RGB value. There are three optional arguments `initialcolor` to specify an initial color such as `"#efefef"`, `parent` to make the dialog a child

### 3. TCL Tk: WIDGETS

---

of a specified window and title to specify a title for the dialog. The return value is in hex-coded RGB quantiles. There is no constructor in `tcltk`, but one can use the dialog as follows:

```
w <- tktoplevel(); tkwm.title(w, "Select a color")
f <- ttkframe(w, padding=c(3,3,3,12))
tkpack(f, expand=TRUE, fill="both")
colorWell <- tkcanvas(f, width=40, height=16,
                      background="#ee11aa",
                      highlightbackground="#ababab")

tkpack(colorWell)
tkbind(colorWell, "<Button-1>", function(W) {
  color <- tcl("tk_chooseColor", parent=w, title="Set box color")
  color <- tclvalue(color)
  if(nchar(color))
    tkconfigure(W, background = color)
})
```