This tutorial will quickly get you up and running with the latest Tk from Tcl, Ruby, Perl or Python on Mac, Windows or Linux. It provides all the essentials about core Tk concepts, the various widgets, layout, events and more that you need for your application.

Table of Contents

Introduction

This tutorial is designed to help people get up to speed quickly with building mainstream desktop graphical user interfaces with Tk, and in particular Tk 8.5, which is an incredibly significant milestone release and a significant departure from the older versions of Tk which most people know and ~~love~~ recognize.

The downside is that unless you know one or two particular things, it's actually not that significant a release; For backwards compatibility reasons, unless existing programs make a few simple changes, they won't look all that much different. So while this tutorial will certainly benefit newcomers to Tk, it will also help existing Tk developers bring their knowledge right up to date. It's a cliche, but I can't believe how much I've learned in writing this tutorial, and I've been using Tk for over fifteen years.

The general state of Tk documentation (outside the Tcl-oriented reference documentation, which is excellent) is unfortunately not at a high point these days. This is particularly true for developers using Tk from languages other than Tcl, and developers working on multiple platforms.

So this tutorial will, as much as possible, target developers on the three main platforms (Windows, Mac, Linux), and also be language-neutral. Initially, the tutorial will cover Tcl, Ruby, Perl and Python. Over time, additional languages may be added. Even if your own language isn't included, the chances are you'll still benefit; since all the languages use the same underlying Tk library, there's obviously a lot of overlap.

This is also not a reference guide, it's not going to cover everything, just the essentials you need in 95% of applications. The rest you can find in reference documentation.

Who this Tutorial is for

This tutorial is designed for developers building tools and applications in Tk. It's also concerned with fairly mainstream graphical user interfaces, with buttons, lists, checkboxes, richtext editing, 2D graphics and so on. So if you're either looking to hack on Tk's internal C code, or build the next great 3D immersive game interface, this is probably not the material for you.

This tutorial also doesn't teach you the underlying programming language (Tcl, Ruby, Perl, Python, etc.), so you should have a basic grasp on that already. Similarly, you should have a basic familiarity with desktop applications in general, and while you don't have to be a user interface designer, some appreciation of GUI design is always helpful.

Modern Best Practices

This tutorial is all about building modern Tk user interfaces using the current tools Tk has to offer. It's all about the best practices you need to know to do this.

For most tools, you wouldn't think you'd have to say something like that, but for Tk that's not the case. Tk has had a very long evolution (see Tk Backgrounder), and any evolution tends to leave you with a bit of cruft; couple that with how much graphical user interface platforms and standards have evolved in that time, and you can see where keeping something as large and complex as a GUI library up to date as well as backwards compatible may be challenging.

Tk has, in recent years, gotten a bad rap, to put it mildly. Some of this has been well deserved, most of it not so much. Like any GUI tool, it can be used to create absolutely terrible looking and outdated user interfaces, but with the proper care and attention, it can also be used to create spectacularly good ones as well. Most people know about the crappy ones; most of the good ones people don't even know are done in Tk. In this tutorial, we're going to focus on what you need to build good user interfaces, which isn't nearly as hard as it used to be before Tk 8.5.

So modern desktop graphical user interfaces, using modern conventions and design sense, using the modern tools

Essentials

- Tk Backgrounder
- Installing Tk
- Tutorial
- Widget Roundup
- Languages Using Tk
- Official Tk Command Reference (Tcl-oriented; at www.tcl.tk)

Tutorial

Show: Tcl

provided by Tk 8.5.

Tk Extensions

When it comes to modern best practices, Tk extensions deserve a special word of note. Over the years, a number of groups have provided all kinds of add-ons to Tk, for example adding new widgets not available in the core (or at least not at the time). Some well-known and quite popular Tk extensions include BLT, Tix, iWidgets, BWidgets; there are many, many others.

Many of these extensions were created years ago. Because core Tk has always been highly backwards compatible, these extensions generally continue to work with newer versions. However, many have not been updated, or not been significantly updated, in a long time. They may not reflect current platform conventions or styles, and so while they "work", they can make your application appear extremely dated or out of place.

If you do decide to use Tk extensions, it's highly recommended that you investigate and review your choices carefully.

The Better Way Forward

Tk also gives you a lot of choices. There are at least six different ways to layout widgets on the screen, often multiple different widgets that could accomplish the same thing, especially if you count the huge assortment of Tk extensions like Tix, BLT, BWidgets, Itk and others. Most of these also are older, most not updated and therefore crappy looking, and in many cases, the facilities they provide have been obsoleted by newer and more modern facilities recently built into Tk itself. But for backwards compatibility reasons, most of these old ways of doing things still keep working, year after year. That doesn't necessarily mean people should still be using some of them.

So there are a lot of choices in Tk, but frankly, all that choice gets in the way. If you want to learn and use Tk, you don't need all the choices, you need the *right* choice, so you don't have to do all the research and make that choice yourself. That's what this tutorial will give you. Think of it as the documentation equivalent of opinionated software . So we'll often use different ways of doing things than in other documentation or examples; often, it's because when those were written, the better ways didn't even exist yet. Later on, once you're an expert, and you're encountering some wacky situation where the normal choice doesn't fit, you can go hunt around for alternatives.

How to Use

While the tutorial is designed to be used linearly, feel free to jump around as you see fit. We'll often provide links where you can go for more information, whether links to other documentation on this site, such as our "widget roundup" providing usage info on each Tk widget, or to external documentation, such as the full reference for a particular command.

The tutorial also lets you select what language (Tcl, Ruby, Perl or Python) to show. You can change this by the "Show:" popup menu which is located in the sidebar, near the top right of each page in the tutorial. But it also lets you see how Tk is used by all the different languages, which can itself be quite interesting and useful.

Conventions

As is typically done, code listings, interpreter or shell commands and responses will be indicated with a `fixed-width font`. When showing an interactive session with the interpreter, the parts that you enter will additionally be in **`bold fixed-width`**.

When describing procedure or method calls, the literal parts (e.g. the method name) will be in plain fixed-width font, parameters where you should fill in the actual value will add italics, and optional parameters will be surrounded by '?', e.g. "`set` *`variable ?value?`*".

A number of icons appearing to the left of text are used, as follows:

This paragraph consists of material that is specific to the Tcl binding to Tk.

> *This paragraph will help point out common mistakes that people make, or suggest useful but not necessarily obvious solutions related to the topic.*

> *This indicates a new way of doing things in Tk 8.5 that is very different from the way things would have been done previously. People familiar with older versions of Tk, or working on programs developed with older versions of Tk, should pay close attention.*

> *This paragraph provides some additional background information, not strictly necessary to understanding the topic at hand, but that might help you understand a bit more about how or why things are done the way they are.*

> *This indicates some error in the tutorial itself which hasn't yet been corrected, or a section that has been deleted but not yet replaced.*

> *This indicates an area in Tk that could most charitably be described as a "rough edge". It may indicate a bad or missing API requiring you to use a workaround in your code. Because these things tend to get fixed up over time, it's worth marking them in your code with a "TODO" so you can remember to go back later and see if a newer API resolves the problem cleanly.*

Installing Tk

In this chapter, you'll get Tk installed on your machine, verify it works, and then see a quick example of what a Tk program looks like.

Though pretty much all Mac OS X and Linux machines come with Tk installed already, it's usually an older version (typically 8.4.x). You want to make sure you've got at least version 8.5 to use the new widget set, so if that's not already there, you'll want to install the newer version.

Though there are lots of ways to install Tk, the easiest is to download and install one of the versions provided by ActiveState (www.activestate.com).

> *ActiveState is a company that sells professional developer tools for dynamic languages. They also provide (for free) quality-controlled distributions of some of these languages, and happen to employ a number of core developers of these languages.*

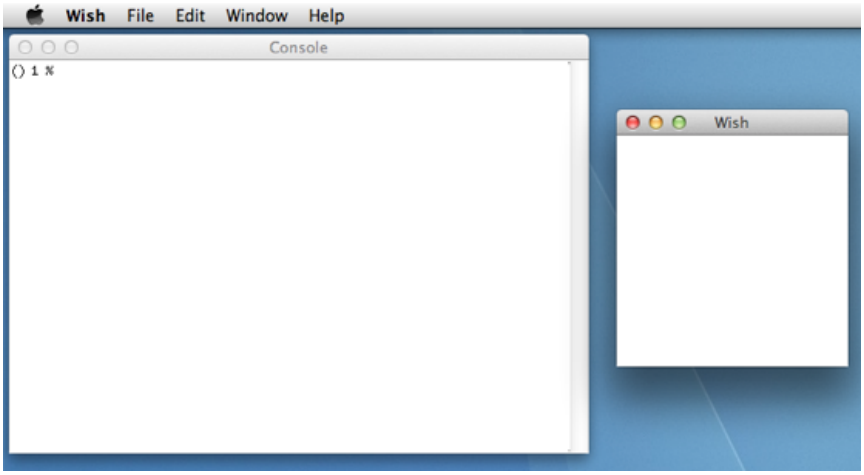
Installing Tk on Mac OS X

On Mac OS X, the easiest way to get Tk is to install the "ActiveTcl" distribution from ActiveState, which includes Tcl, Tk, plus a number of other extension libraries.

In your web browser, go to www.activestate.com , and follow along the links to download the Community Edition of ActiveTcl, available as a universal binary. Make sure you're downloading an 8.5.x version, not an older 8.4.x version.

Run the installer to get everything loaded onto your machine. When you're done, you'll find a shiny new application called "Wish 8.5" inside the Utilities folder of your Applications folder. This is the "wish" shell, an application that includes both Tcl and Tk.

If you launch that application, you'll see two windows popup (see below), one titled "Wish" which will contain your application, and the second titled "Console" which is where you can type in Tcl/Tk commands.



The Wish application running on Mac OS X.

For convenient use from the Unix command line, you'll also find a script installed as /usr/local/bin/wish8.5 which will launch the same application.

To verify the exact version of Tcl/Tk that you are running, from the Wish console type the following:

```
% info patchlevel
```

We want this to be returning something like '8.5.10'.

> *Verified install using ActiveTcl 8.5.10.1 on Mac OS X 10.7.1.*

Installing Tk on Windows

On Windows, the easiest way to get Tcl/Tk onto your machine is to install the "ActiveTcl" distribution from ActiveState, which includes Tcl, Tk, plus a number of other extension libraries.

In your web browser, go to www.activestate.com , and follow along the links to download the Community Edition of ActiveTcl for Windows. Make sure you're downloading an 8.5.x version, not an older 8.4.x version.

Run the installer, and follow along. You'll end up with a fresh install of ActiveTcl, usually located in C:\Tcl. From a DOS command prompt, or the Start Menu's "Run..." command, you should then be able to run a Tcl/Tk 8.5 shell via:

```
% C:\Tcl\bin\wish85
```

This should pop up a small window titled "wish85", which will contain your application. A second, larger window titled "Console" is where you can type in Tcl/Tk commands. To verify the exact version of Tcl/Tk that you are running, type the following:

```
% info patchlevel
```

We want this to be returning something like '8.5.10'.

Type "exit" in the console window to exit. You may also want to add C:\Tcl\bin to your PATH environment variable.

> *Verified install using ActiveTcl 8.5.10.1 on Windows 7.*

Installing Tk on Linux

While Linux distributions pretty much all come with Tcl/Tk installed, most include Tk 8.4.x, and we want to make sure to get an 8.5.x version. The easiest way to do this is to install the "ActiveTcl" distribution from ActiveState, which includes Tcl, Tk, plus a number of other extension libraries.

In your web browser, go to www.activestate.com , and follow along the links to download the Community Edition of ActiveTcl for Linux. Make sure you're downloading an 8.5.x version, not an older 8.4.x version.

Unpack it, and run the installer (install.sh), and follow along. You'll end up with a fresh install of ActiveTcl, located in /opt/ActiveTcl-8.5. You should then be able to run a Tcl/Tk 8.5 shell via:

```
% /opt/ActiveTcl-8.5/bin/wish8.5
```

This should pop up a window titled "wish8.5". To verify the exact version of Tcl/Tk that you are running, from the Wish prompt (in the terminal window) type the following:

```
% info patchlevel
```

We want this to be returning something like '8.5.10'. Type a control-D at the prompt in the terminal window to exit. You may also want to add /opt/ActiveTcl-8.5/bin to your Unix path.

> *Verified install using ActiveTcl 8.5.10.1 on Ubuntu 11.04.*

The Obligatory First Program

To make sure that everything actually did work, let's try to run a "Hello World" program in Tk. While for something this short you could just type it in directly to the interpreter, instead use your favorite text editor to put it in a file.

```
package require Tk
grid [ttk::button .b -text "Hello World"]
```

Save this to a file named 'hello.tcl'. From the wish shell, type:

```
% source hello.tcl
```

> *Couldn't find hello.tcl? You might be looking in the wrong directory. You can either give the full path to hello.tcl, or use Tcl's "pwd" and "cd" commands to see what directory you're in, and change to a different one.*

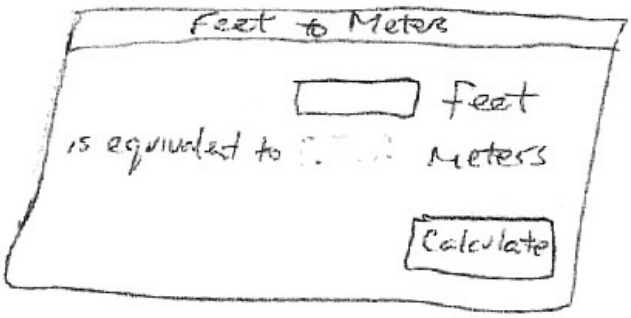
Our First Program. Some work left to do before the IPO.

A First (Real) Example

With that out of the way, let's try a slightly more useful example, which will give you an initial feel for what the code behind a Tk program looks like.

Design

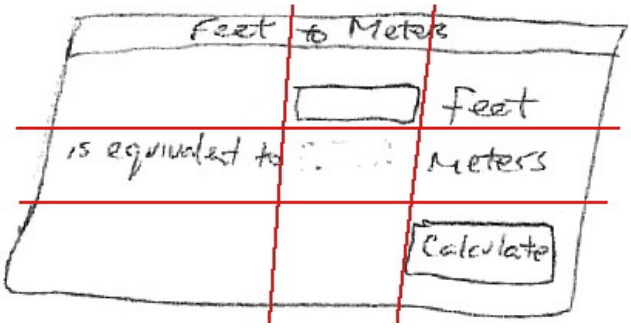
The example we'll use is a simple GUI tool that will convert a number of feet to the equivalent number of meters. If we were to sketch this out, it might look something like this:



A sketch of our feet to meters conversion program.

So it looks like we have a short text entry widget that will let us type in the number of feet, and a 'Calculate' button that will get the value out of that entry, perform the calculation, and then put the resulting number of meters on the screen just below where the entry is. We've also got three static labels ("feet", "is equivalent to", and "meters") which help our user figure out how to use the interface.

In terms of layout, things seem to naturally divide into three columns and three rows:



The layout of our user interface, which follows a 3 x 3 grid.

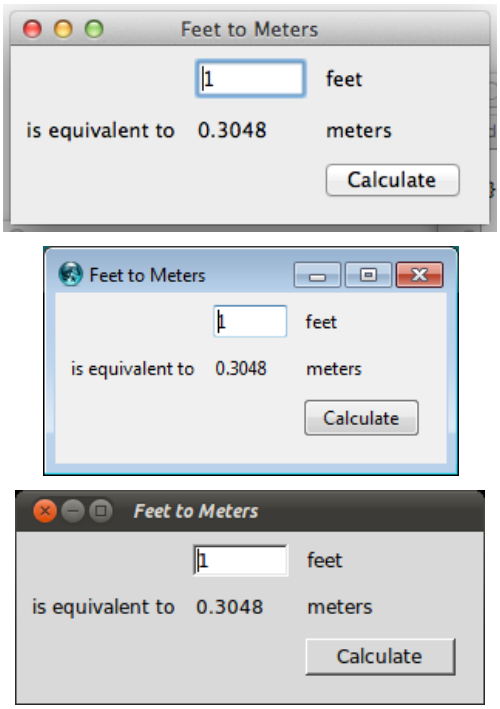
Code

Now here is the Tcl/Tk code to create this program.

```
package require Tk

wm title . "Feet to Meters"
grid [ttk::frame .c -padding "3 3 12 12"] -column 0 -row 0 -sticky nwes
grid columnconfigure . 0 -weight 1; grid rowconfigure . 0 -weight 1

grid [ttk::entry .c.feet -width 7 -textvariable feet] -column 2 -row 1 -sticky we
grid [ttk::label .c.meters -textvariable meters] -column 2 -row 2 -sticky we
grid [ttk::button .c.calc -text "Calculate" -command calculate] -column 3 -row 3 -sticky w
```

```
grid [ttk::label .c.flbl -text "feet"] -column 3 -row 1 -sticky w
grid [ttk::label .c.islbl -text "is equivalent to"] -column 1 -row 2 -sticky e
grid [ttk::label .c.mlbl -text "meters"] -column 3 -row 2 -sticky w

foreach w [winfo children .c] {grid configure $w -padx 5 -pady 5}
focus .c.feet
bind . <Return> {calculate}

proc calculate {} {
   if {[catch {
       set ::meters [expr {round($::feet*0.3048*10000.0)/10000.0}]
   }]!=0} {
       set ::meters ""
   }
}
```

And the resulting user interface:



Screenshot of our completed feet to meters user interface (on Mac OS X, Windows and Linux).

A Note on Coding Style

Each of the languages included in this tutorial has a variety of coding styles and conventions available to choose from, which help determine conventions for variable and function naming, procedural, functional or object-oriented styles, and so on.

Because the focus on this tutorial is Tk, this tutorial will keep things as simple as possible, generally using a very direct coding style, rather than wrapping up most of our code in procedures, modules, objects, classes and so on. As much as possible, you'll also see the same names for objects, variables, etc. used across the languages for each example.

Step-by-Step Walkthrough

Let's take a closer look at that code, piece by piece. For now, all we're trying to do is get a basic understanding of the types of things we need to do to create a user interface in Tk, and roughly what those things look like. We'll go into details later.

```
package require Tk
```

First thing we do is tell Tcl that our program needs Tk. Though not strictly necessary, it's considered good form to include this line. It can also be used to specify exactly what version of Tk is needed.

```
wm title . "Feet to Meters"
grid [ttk::frame .c -padding "3 3 12 12"] -column 0 -row 0 -sticky nwes
grid columnconfigure . 0 -weight 1; grid rowconfigure . 0 -weight 1
```

Next, the above lines set up the main window, giving it the title "Feet to Meters". Next, we create a frame widget, which will hold all the content of our user interface, and place that in our main window. The

"columnconfigure"/"rowconfigure" bits just tell Tk that if the main window is resized, the frame should expand to take up the extra space.

> *Strictly speaking, we could just put the other parts of our interface directly into the main root window, without the intervening content frame. However, the main window isn't itself part of the "themed" widgets, so its background color wouldn't match the themed widgets we will put inside it. Using a "themed" frame widget to hold the content ensures that the background is correct.*

```
grid [ttk::entry .c.feet -width 7 -textvariable feet] -column 2 -row 1 -sticky we
grid [ttk::label .c.meters -textvariable meters] -column 2 -row 2 -sticky we
grid [ttk::button .c.calc -text "Calculate" -command calculate] -column 3 -row 3 -sticky w
```

The preceding lines create the three main widgets in our program: the entry where we type the number of feet in, a label where we put the resulting number of meters, and the calculate button that we press to perform the calculation.

For each of the three widgets, we need to do two things: create the widget itself, and then place it onscreen. All three widgets, which are 'children' of our content window are created as instances of one of Tk's themed widget classes. At the same time as we create them, we give them certain options, such as how wide the entry is, the text to put inside the Button, etc. The entry and label each are assigned a mysterious "textvariable"; we'll see what that does shortly.

If the widgets are just created, they won't automatically show up on screen, because Tk doesn't know how you want them to be placed relative to other widgets. That's what the "grid" part does. Remembering the layout grid for our application, we place each widget in the appropriate column (1, 2 or 3), and row (also 1, 2 or 3). The "sticky" option says how the widget would line up within the grid cell, using compass directions. So "w" (west) means anchor the widget to the left side of the cell, "we" (west-east) means anchor it to both the left and right sides, and so on.

```
grid [ttk::label .c.flbl -text "feet"] -column 3 -row 1 -sticky w
grid [ttk::label .c.islbl -text "is equivalent to"] -column 1 -row 2 -sticky e
grid [ttk::label .c.mlbl -text "meters"] -column 3 -row 2 -sticky w
```

The above three lines do exactly the same thing for the three static text labels in our user interface; create each one, and place it onscreen in the appropriate cell in the grid.

```
foreach w [winfo children .c] {grid configure $w -padx 5 -pady 5}
focus .c.feet
bind . <Return> {calculate}
```

The preceeding three lines help put some nice finishing touches on our user interface.

The first line walks through all of the widgets that are children of our content frame, and adds a little bit of padding around each, so they aren't so scrunched together. We could have added these options to each "grid" call when we first put the widgets onscreen, but this is a nice shortcut.

The second line tells Tk to put the focus on our entry widget. That way the cursor will start in that field, so the user doesn't have to click in it before starting to type.

The third line tells Tk that if the user presses the Return key (Enter on Windows) anywhere within the root window, that it should call our calculate routine, the same as if the user pressed the Calculate button.

```
proc calculate {} {
   if {[catch {
       set ::meters [expr {round($::feet*0.3048*10000.0)/10000.0}]
   }]!=0} {
       set ::meters ""
   }
}
```

Here we define our calculate procedure, which is called either when the user presses the Calculate button, or hits the Return key. It performs the feet to meters calculation, taking the number of feet from our entry widget, and placing the result in our label widget.

Say what? It doesn't look like we're doing anything with those widgets! Here's where the magic "textvariable" options we specified when creating the widgets come into play. We specified the global variable "feet" as the textvariable for the entry, which means that anytime the entry changes, Tk will *automatically* update the global variable feet. Similarly, if we explicitly change the value of a textvariable associated with a widget (as we're doing for "meters" which is attached to our label), the widget will automatically be updated with the current contents of the variable. Slick.

What's Missing

It's also worth examing what we *didn't* have to include in our Tk program to make it work. For example:

- we didn't have to worry about redrawing the screen as things changed
- we didn't have to worry about parsing and dispatching events, hit detection, or handling events on each widget
- we didn't have to provide a lot of options when we created widgets; the defaults seemed to take care of most things, and so we only had to change things like the text the button displayed
- we didn't have to write complex code to get and set the values of simple widgets; we just attached them to variables
- we didn't have to worry about what happens when the user closes the window or resizes it

- we didn't need to write extra code to get this all to work cross-platform

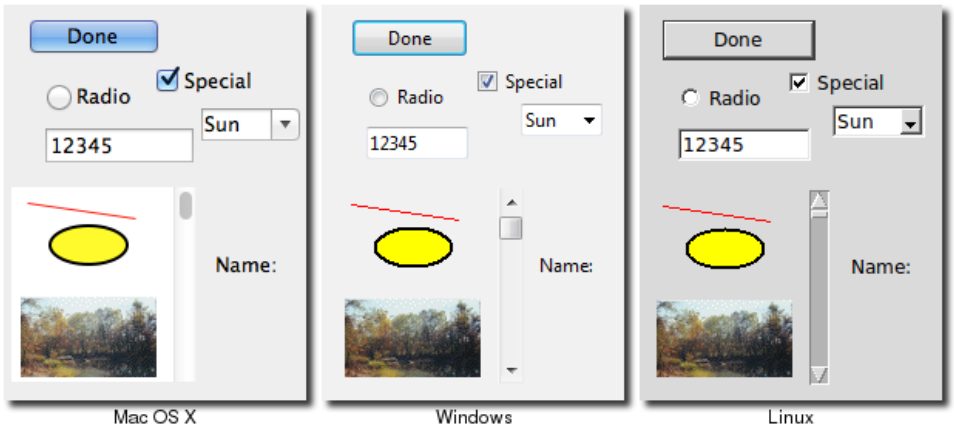
Tk Concepts

With a first example behind you, you should have a basic idea of what a Tk program might look like and the types of tasks it needs to accomplish. We'll step back and look at three broad concepts that you need to know to understand Tk: widgets, geometry management, and event handling.

Widgets

Widgets are all the things that you see onscreen. In our example, we had a button, an entry, a few labels, and a frame. Others are things like checkboxes, tree views, scrollbars, text areas, and so on. Widgets are what are often referred to as "controls"; you'll also often see them referred to as "windows", particularly in Tk's documentation, a holdover from its X11 roots (so under that terminology, both a toplevel window and things like a button would be called windows).

Here is an example showing some of Tk's widgets, which we'll cover individually shortly.



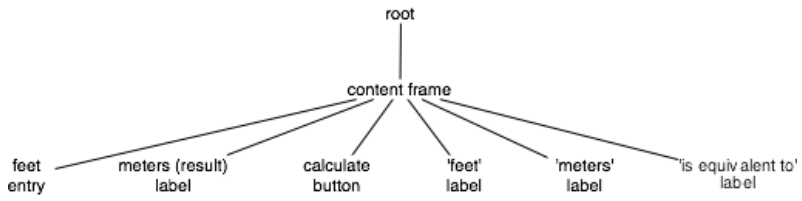
Widget Classes

Widgets are objects, instances of classes that represent buttons, frames, and so on. So the first thing you'll need to do is identify the specific class of the widget you'd like to instantiate. This tutorial and the widget roundup will help with that.

Window Hierarchy

The other thing you'll need to know is the *parent* of the widget instance you'd like to create. In Tk, all widgets are part of a *window hierarchy*, with a single root at the top of the hierarchy. This hierarchy can be arbitrarily deep; so you might have a button in a frame in another frame within the root window. Even a new toplevel window is part of that same hierarchy, with it and all its contents forming a subtree of the overall window hierarchy.

In our metric conversion example, we had a single frame that was created as a child of the root window, and that frame had all the other controls as children. The root window was a *container* for the frame, and was therefore the frame's *parent*. The complete hierarchy for the example looked like this:



The window hierarchy of the metric conversion example.

Creating and Using Widgets

In Tcl, each widget is given an explicit *pathname*, which both differentiates it from other widgets, and also indicates its place in the window hierarchy. The root of the hierarchy, the toplevel widget that Tk automatically creates, is named simply `"."` (dot).

The frame, which was a child of the root, was named `".c"`. We could have put pretty much anything in place of the "c", naming it for example `".content"`. This name is purely for use by your program, so it's best to choose something meaningful. The controls that were children of the frame were given names like `".c.feet"`, `".c.meters"`, `".c.flbl"`, and so on. If there were any widgets at a deeper level of the hierarchy, we'd add another `"."` and then a unique identifier.

So to create a widget, we need to provide the widget class, and the pathname. The pathname is used to indicate the widget's parent (which must of course exist also), and hence its position in the window hierarchy. For example:

```
ttk::button .b
ttk::frame .f
ttk::entry .f.entry
```

This also creates a new *object command* with the same name as the widget's pathname, which will let us communicate with the widget. So the above code would produce new Tcl commands named `".b"`, `".f"`, `".f.entry"`, and so on. You can then use that object command to communicate further with the widget, calling e.g. `".b invoke"`, or `".f.entry state disabled"`. Because of the obvious parallels with many object-oriented systems, we'll often refer to the object commands as *objects*, and calls on those objects (like the `"invoke"`) as *method* calls. For example, you'll see below the use of the `"configure"` and `"cget"` methods.

Configuration Options

All widgets also have a number of different *configuration options*, which generally control how they are displayed or how they behave.

The options that are available depend upon the widget class of course. There is a lot of consistency between different widget classes, so options that do pretty much the same thing tend to be named the same. So both a button and a label have a `"text"` option to adjust the text the widget displays, while a scrollbar for example would not have a `"text"` option since it's not needed. In the same way, the button has a `"command"` option telling it what to do when pushed, while a label, which holds just static text, does not.

Configuration options can be set when the widget is first created, by passing along the names and values of the options as optional parameters. You can later check what the value of those options are, and with a very small number of exceptions, change them at any time. If you're not sure what all the different options are for a widget, you can ask the widget to provide it. This gives you a long list of all the options, and for each option, you can see the name of the option and its current value (along with three other attributes which you won't normally need to worry about).

This is all best illustrated with the following interactive dialog with the interpreter.

```
% wish8.5
create a button, passing two options:
% grid [ttk::button .b -text "Hello" -command {button_pressed}]
check the current value of the text option:
% .b cget -text
Hello
check the current value of the command option:
% .b cget -command
button_pressed
change the value of the text option:
% .b configure -text Goodbye
check the current value of the text option:
% .b cget -text
Goodbye
get all information about the text option:
% .b configure -text
-text text Text {} Goodbye
get information on all options for this widget:
% .b configure
{-takefocus takeFocus TakeFocus ttk::takefocus ttk::takefocus}
{-command command Command {} button_pressed} {-default default Default normal normal}
{-text text Text {} Goodbye} {-textvariable textVariable Variable {} {}}
{-underline underline Underline -1 -1} {-width width Width {} {}} {-image image Image {} {}}
{-compound compound Compound none none} {-padding padding Pad {} {}}
{-state state State normal normal} {-takefocus takeFocus TakeFocus {} ttk::takefocus}
{-cursor cursor Cursor {} {}} {-style style Style {} {}} {-class {} {} {} {}}
```

Geometry Management

If you've been playing around creating widgets, you've probably noticed that just by creating them they didn't end up showing up onscreen. Having things actually put in the onscreen window, and precisely *where* in the window they show up is a separate step called *geometry management*.

In our example, this positioning was accomplished by the `"grid"` command, where we passed along the column and row we wanted each widget to go in, how things were to be aligned within the grid, and so on. Grid is an example of a *geometry manager* (of which there are several in Tk, grid being the most useful). We'll talk about grid in detail in a later chapter, but for now we'll look at geometry management in general.

A geometry manager's job is to figure out exactly where those widgets are going to be put. This turns out to be a very difficult optimization problem, and a good geometry manager relies on quite complex algorithms. A good geometry manager provides the flexibility, power and ease of use that makes programmers happy, and Tk's `"grid"` is without a doubt one of the absolute best. A poor geometry manager... well, all the Java programmers who have suffered through "GridBagLayout" please raise their hands.

The Problem

The problem for a geometry manager is to take all the different widgets the program creates, plus the instructions for where in the window the program would like things to go (explicitly, or more often, relative to other widgets), and then actually put them in the window.

In doing so, the geometry manager has to balance a number of different constraints:

- The widgets may have a "natural" size (e.g. the natural width of a label would normally be determined by the text and font in it), but the toplevel all these different widgets are trying to fit into isn't big enough to accommodate them; the geometry manager must decide which widgets to shrink to fit, by how much, etc.
- If the toplevel window is bigger than the natural size of all the widgets, how is the extra space used? Is it just used for extra space between widgets, and if so, how is that space distributed? Is it used to make certain widgets bigger than they normally want to be?
- If the toplevel window is resized, how does the size and position of the widgets in it change? Will certain areas (e.g. a text entry area) expand or shrink, while other parts stay the same size, or is the area distributed differently? Do certain widgets have a minimum (or maximum) size that you want to avoid going under (over)?
- How can widgets in different parts of the user interface be aligned with each other, to present a clean layout and match platform guidelines to do with inter-widget spacing?
- For a complex user interface, which may have many frames nested in other frames nested in the window (etc.), how can all the above be accomplished, trading off the conflicting demands of different parts of the entire user interface?

How it Works

Geometry management in Tk relies on the concept of *master* and *slave* widgets. A master is a widget, typically a toplevel window or a frame, which will contain other widgets, which are called slaves. You can think of a geometry manager as taking control of the master widget, and deciding what will be displayed within.

The geometry manager will ask each slave widget for its natural size, or how large it would ideally like to be displayed. It then takes that information and combines it with any parameters provided by the program when it asks the geometry manager to manage that particular slave widget. In our example, we passed grid a `"column"` and `"row"` number for each widget, which indicated the relative position of the widget with respect to others, and also a `"sticky"` parameter to suggest how the widget should be aligned or possibly stretched. We also used `"columnconfigure"` and `"rowconfigure"` to indicate the columns and rows we'd like to have expand if there is extra space available in the window. Of course, all these parameters are specific to grid; other geometry managers would use different ones.

The geometry manager takes all the information about the slaves, as well as the information about how large the master is, and uses its internal algorithms to determine the area each slave will be allocated (if any!). The slave is then responsible for drawing etc. within that particular rectangle. And of course, any time the size of the master changes (e.g. because the toplevel window was resized), the natural size of a slave changes (e.g. because we've changed the text in a label), or any of the geometry manager parameters change (e.g. like `"row"`, `"column"`, or `"sticky"`) we repeat the whole thing.

This all works recursively as well. In our example, we had a content frame inside the toplevel window, and then a number of other controls in the content frame. We therefore had a geometry manager working on two different masters. At the outer level, the toplevel window was the master, and the content frame was the slave. At the inner level, the content frame was the master, with each of the other widgets being slaves. So the same widget can be both a master and a slave. This hierarchy can of course also be nested much more deeply.

> *While each master can have only one geometry manager (e.g. grid), it's entirely possible for different masters to have different geometry managers; while grid is generally used, others may make sense for a particular layout used in one part of your user interface. Also, we've been making the assumption that slave widgets are the immediate children of their master in the widget hierarchy. While this is usually the case, and mostly there's no good reason to do it any other way, it's also possible (with some restrictions) to get around this.*

Event Handling

In Tk, as in most other user interface toolkits, there is an *event loop* which receives events from the operating system. These are things like button presses, keystrokes, mouse movement, window resizing, and so on.

Generally, Tk takes care of managing this event loop for you. It will figure out what widget the event applies to (did the user click on this button? if a key was pressed, which textbox had the focus?), and dispatch it accordingly. Individual widgets know how to respond to events, so for example a button might change color when the mouse moves over it, and revert back when the mouse leaves.

Command Callbacks

Often though you want your program to handle particular events, for example doing something when a button is pushed. For those events that are pretty much essential to customize (what good is a button without something happening when you press it?), the widget will provide a *callback* as a widget configuration option. We saw this in the example with the `"command"` option of the button.

Callbacks in Tk tend to be simpler than in toolkits used with compiled languages (where a callback must generally be

directed at a procedure with a certain set of parameters or an object method with a certain signature). Instead, the callback is just a normal bit of code that the interpreter evaluates. While it can be as complex as you want to make it, most times though you'll just want your callback to call some other procedure.

Event Bindings

For events that don't have a command callback associated with them, you can use Tk's `"bind"` to capture any event, and then (like with callbacks) execute an arbitrary piece of code.

Here is a (silly) example that shows how a label can have bindings set up for it to respond to different events, which it does so by just changing what is displayed in the label.

```
package require Tk
grid [ttk::label .l -text "Starting..."]
bind .l <Enter> {.l configure -text "Moved mouse inside"}
bind .l <Leave> {.l configure -text "Moved mouse outside"}
bind .l <1> {.l configure -text "Clicked left mouse button"}
bind .l <Double-1> {.l configure -text "Double clicked"}
bind .l <B3-Motion> {.l configure -text "right button drag to %x %y"}
```

> *Note that the `bind` command lives in the global namespace; there is not a `ttk::bind` command.*

The first three event bindings are pretty straightforward, just looking at simple events. The double click binding introduces the idea of an *event modifier*; in this case we want to trigger the event on a left mouse click (the `"1"`), but only when it's a double click (the `"Double-"`).

The last binding also uses a modifier: capture mouse movement (`"Motion"`), but only when the right mouse button (`"B3"`) is held down. This binding also shows an example of how to use *event parameters*, through the use of *percent substitutions*. Many events, such as mouse clicks or movement have as parameters additional information like the current position of the mouse. These percent substitutions let you capture them so they can be used in your script.

For a complete description of all the different event names, modifiers, and the different event parameters that are available with each, the best place to look is the "bind" command reference .

Virtual Events

Beyond the low-level operating system events like mouse clicks and window resizes, many widgets generate higher level events called *virtual events*. For example, a listbox widget will generate a `"ListboxSelect"` virtual event anytime the selection changes, regardless of whether that was because the user clicked on an item, moved to it with the arrow keys, or whatever. This avoids the problem of setting up multiple, possibly platform-specific event bindings to capture the change. Virtual events for a widget, if any, will be listed in the widget's documentation.

Multiple Bindings

Widgets can actually have a number of different event bindings trigger for a single event. Normally, events can be set up for: the individual widget itself, all widgets of a certain class (e.g. buttons), the toplevel window containing the widget, and all widgets in the application. Each of these will fire in sequence.

We saw this in our example when we set up a binding for the Return key on the toplevel window, and that applied to every widget within that window.

> *The default behavior of each widget class in Tk is itself defined with script-level event bindings, and so can be introspected and modified to alter the behavior of all widgets of a certain class. You can even completely modify the handling of this multiple sequence of events for each widget; see the "bindtags" command reference if you're curious.*

Basic Widgets

This chapter introduces you to the basic Tk widgets that you'll find in just about any user interface: frames, labels, buttons, checkbuttons, radiobuttons, entries and comboboxes. By the end, you'll know how to use all the widgets you'd ever need for a typical fill-in form type of user interface.

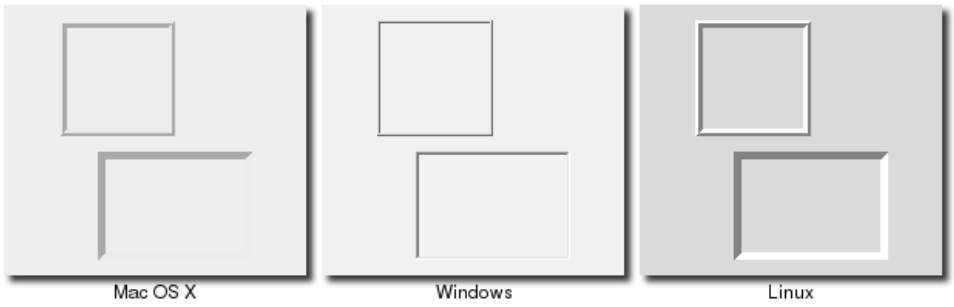
This chapter (and those following that discuss more widgets) are meant to be read in order. Because there is so much commonality between many widgets, we'll introduce certain concepts in an earlier widget that will also apply to a later one. Rather than going over the same ground multiple times, we'll just refer back to when the concept was first introduced.

At the same time, each widget will also refer to the widget roundup page for the specific widget, as well as the reference manual page , so feel free to jump around a bit too.

Frame

A **frame** is a widget that displays just as a simple rectangle. Frames are primarily used as a container for other widgets, which are under the control of a geometry manager such as grid.

- Widget Roundup
- Reference Manual



Frame Widgets

Frames are created using the **ttk::frame** command:

```
ttk::frame .frame
```

Frames can take several different configuration options which can alter how they are displayed.

Requested Size

Like any other widget, after creation it is added to the user interface via a (parent) geometry manager. Normally, the size that the frame will request from the geometry manager will be determined by the size and layout of any widgets that are contained in it (which are under the control of the geometry manager that manages the contents of the frame itself).

If for some reason you want an empty frame that does not contain other widgets, you should instead explicitly set the size that the frame will request from its parent geometry manager using the `"width"` and/or `"height"` configuration options (otherwise you'll end up with a very small frame indeed).

> *Normally, distances such as width and height are specified just as a number of pixels on the screen. You can also specify them via one of a number of suffixes. For example, `"350"` means 350 pixels, `"350c"` means 350 centimeters, `"350i"` means 350 inches, and `"350p"` means 350 printer's points (1/72 inch).*

Padding

The `"padding"` configuration option is used to request extra space around the inside of the widget; this way if you're putting other widgets inside the frame, there will be a bit of a margin all the way around. A single number specifies the same padding all the way around, a list of two numbers lets you specify the horizontal then the vertical padding, and a list of four numbers lets you specify the left, top, right and bottom padding, in that order.

```
.frame configure -padding "5 10"
```

Borders

You can display a border around the frame widget; you see this a lot where you might have a part of the user interface looking "sunken" or "raised" in relation to its surroundings. To do this, you need to set the `"borderwidth"` configuration option (which defaults to 0, so no border), as well as the `"relief"` option, which specifies the visual appearance of the border: `"flat"` (default), `"raised"`, `"sunken"`, `"solid"`, `"ridge"`, or `"groove"`.

```
.frame configure -borderwidth 2 -relief sunken
```

Changing Styles

There is also a `"style"` configuration option, which is common to all of the themed widgets, which can let you control just about any aspect of their appearance or behavior. This is a bit more advanced, so we won't go into it right now.

> *Styles mark a sharp departure from the way most aspects of a widget's visual appearance are changed in the "classic" Tk widgets. While in classic Tk you could provide a wide range of options to finely control every aspect of behavior (foreground color, background color, font, highlight thickness, selected foreground color, padding, etc.), in the new themed widgets these changes are done by changing styles, not adding options to each widget.*
>
> *As such, many of the options you may be familiar with in certain widgets are not present in their themed version. Given that overuse of such options was a key factor undermining the appearance of Tk applications, especially when moved across platforms, transitioning to themed widgets provides an opportune time to review and refine if and how such appearance changes are made.*

Label

A **label** is a widget that displays text or images, typically that the user will just view but not otherwise interact with. Labels are used for such things as identifying controls or other parts of the user interface, providing textual feedback or results, etc.

- Widget Roundup
- Reference Manual

Label Widgets

Labels are created using the **`ttk::label`** command, and typically their contents are set up at the same time:

```
ttk::label .label -text {Full name:}
```

Like frames, labels can take several different configuration options which can alter how they are displayed.

Displaying Text

The `"text"` configuration option shown above when creating the label is the most commonly used, particularly when the label is purely decorative or explanatory. You can of course change this option at any time, not only when first creating the label.

You can also have the widget monitor a variable in your script, so that anytime the variable changes, the label will display the new value of the variable; this is done with the `"textvariable"` option:

```
.label configure -textvariable resultContents
set resultContents "New value to display"
```

Variables must be global, or the fully qualified name given for those within a namespace.

Displaying Images

You can also display an image in a label instead of text; if you just want an image sitting in your interface, this is normally the way to do it. We'll go into images in more detail in a later chapter, but for now, let's assume you want to display a GIF image that is sitting in a file on disk. This is a two-step process, first creating an image "object", and then telling the label to use that object via its `"image"` configuration option:

```
image create photo imgobj -file "myimage.gif"
.label configure -image imgobj
```

You can use both an image and text, as you'll often see in toolbar buttons, via the `"compound"` configuration option. The default value is `"none"`, meaning display only the image if present, otherwise the text specified by the `"text"` or `"textvariable"` options. Other options are `"text"` (text only), `"image"` (image only), `"center"` (text in center of image), `"top"` (image above text), `"left"`, `"bottom"`, and `"right"`.

Layout

While the overall layout of the label (i.e. where it is positioned within the user interface, and how large it is) is determined by the geometry manager, there are several options that can help you control how the label will be displayed within the box the geometry manager gives it.

If the box given to the label is larger than the label requires for its contents, you can use the `"anchor"` option to specify what edge or corner the label should be attached to, which would leave any empty space in the opposite edge or corner. Possible values are specified as compass directions: `"n"` (north, or top edge), `"ne"`, (north-east, or top right corner), `"e"`, `"se"`, `"s"`, `"sw"`, `"w"`, `"nw"` or `"center"`.

Labels can be used to display more than one line of text. This can be done by embedding carriage returns (`"\n"`) in the `"text"`/`"textvariable"` string. You can also let the label wrap the string into multiple lines that are no longer than a given length (with the size specified as pixels, centimeters, etc.), by using the `"wraplength"` option.

> *Multi-line labels are a replacement for the older `"message"` widgets in classic Tk.*

You can also control how the text is justified, by using the `"justify"` option, which can have the values `"left"`, `"center"` or `"right"`. If you only have a single line of text, this is pretty much the same as just using the `"anchor"` option, but is more useful with multiple lines of text.

Fonts, Colors and More

Like with frames, normally you don't want to touch things like the font and colors directly, but if you need to change them

(e.g. to create a special type of label), this would be done via creating a new style, which is then used by the widget with the `"style"` option.

Unliked most themed widgets, the label widget also provides explicit widget-specific options as an alternative; again, you'd use this only in special one-off cases, when using a style didn't necessarily make sense.

You can specify the font used to display the label's text using the `"font"` configuration option. While we'll go into fonts in more detail in a later chapter, here are the names of some predefined fonts you can use:

`TkDefaultFont`	The default for all GUI items not otherwise specified.
`TkTextFont`	Used for entry widgets, listboxes, etc.
`TkFixedFont`	A standard fixed-width font.
`TkMenuFont`	The font used for menu items.
`TkHeadingFont`	The font typically used for column headings in lists and tables.
`TkCaptionFont`	A font for window and dialog caption bars.
`TkSmallCaptionFont`	A smaller caption font for subwindows or tool dialogs
`TkIconFont`	A font for icon captions.
`TkTooltipFont`	A font for tooltips.

> *Because the choice of fonts is so platform specific, be careful of hardcoding them (font families, sizes, etc.); this is something else you'll see in a lot of older Tk programs that can make them look ugly.*

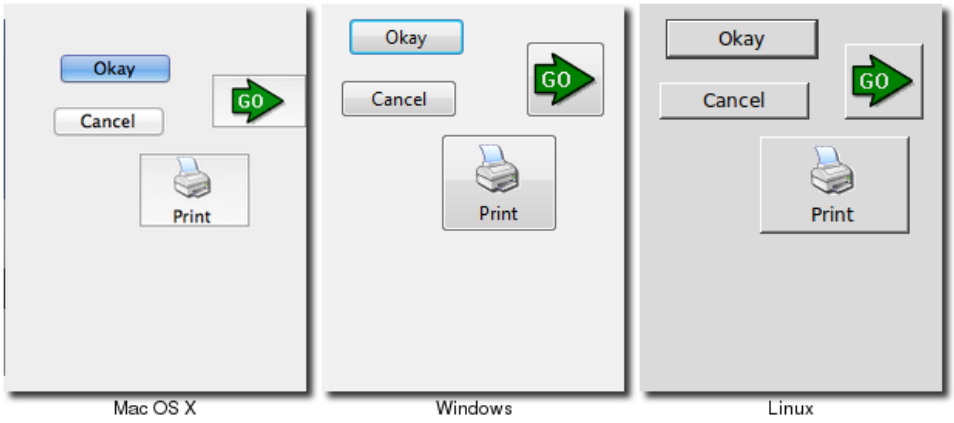
The foreground (text) and background color can also be changed via the `"foreground"` and `"background"` options. Colors are covered in detail later, but you can specify these as either color names (e.g. `"red"`) or hex RGB codes (e.g. `"#ff340a"`).

Labels also accept the *"relief"* option that was discussed for frames.

Button

A **button**, unlike a frame or label, is very much designed for the user to interact with, and in particular, press to perform some action. Like labels, they can display text or images, but also have a whole range of new options used to control their behavior.

- Widget Roundup
- Reference Manual



Button Widgets

Buttons are created using the **`ttk::button`** command, and typically their contents and command callback are set up at the same time:

```
ttk::button .button -text "Okay" -command "submitForm"
```

As with other widgets, buttons can take several different configuration options which can alter their appearance and behavior.

Text or Image

Buttons take the same `"text"`, `"textvariable"` (rarely used), `"image"` and `"compound"` configuration options as labels, which control whether the button displays text and/or an image.

Buttons have a `"default"` option, which tells Tk that the button is the default button in the user interface (i.e. the one that will be invoked if the user hits Enter or Return). Some platforms and styles will draw this with a different border or highlight. Set the option to `"active"` to specify this is a default button; the regular state is `"normal"`. Note that setting this option doesn't create an event binding that will make the Return or Enter key activate the button; that you have to do

yourself.

The Command Callback

The `"command"` option is used to provide an interface between the button's action and your application. When the user clicks the button, the script provided by the option is evaluated by the interpreter.

You can also ask the button to invoke the command callback from your application. This is useful so that you don't need to repeat the command to be invoked several times in your program; so you know if you change the option on the button, you don't need to change it elsewhere too.

```
.button invoke
```

Button State

Buttons and many other widgets can be in a normal state where they can be pressed, but can also be put into a disabled state, where the button is greyed out and cannot be pressed. This is done when the button's command is not applicable at a given point in time.

All themed widgets carry with them an internal state, which is a series of binary flags. You can set or clear these different flags, as well as check the current setting using the `"state"` and `"instate"` methods. Buttons make use of the `"disabled"` flag to control whether or not the user can press the button. For example:

```
.button state disabled            ;# set the disabled flag, disabling the button
.button state !disabled           ;# clear the disabled flag
.button instate disabled          ;# return 1 if the button is disabled, else 0
.button instate !disabled         ;# return 1 if the button is not disabled, else 0
.button instate !disabled {mycmd} ;# execute 'mycmd' if the button is not disabled
```

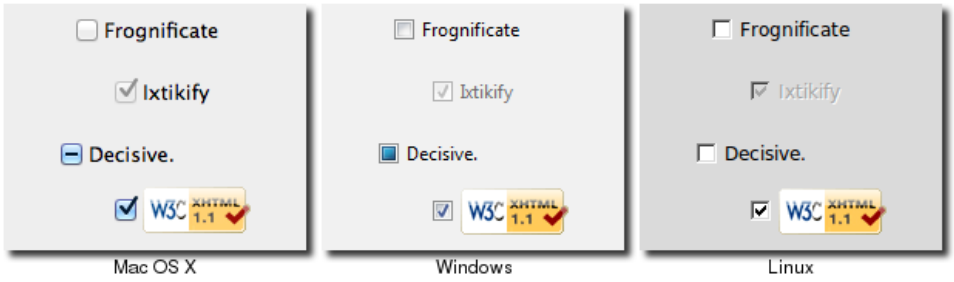
> *Using `"state"`/`"instate"` replaces the older `"state"` configuration option (which took the values `"normal"` or `"disabled"`). This configuration option is actually still available in Tk 8.5, but "write-only", which means that changing the option calls the appropriate `"state"` command, but other changes made using the `"state"` command are not reflected in the option. This is only for compatibility reasons; you should change your code to use the new state vector.*

> *The full list of state flags available to themed widgets is: `"active"`, `"disabled"`, `"focus"`, `"pressed"`, `"selected"`, `"background"`, `"readonly"`, `"alternate"`, and `"invalid"`. These are described in the themed widget reference ; not all states are meaningful for all widgets. It's also possible to get fancy in the `"state"` and `"instate"` methods and specify multiple state flags at the same time.*

Checkbutton

- Widget Roundup
- Reference Manual

A **checkbutton** is like a regular button, except that not only can the user press it, which will invoke a command callback, but it also holds a binary value of some kind (i.e. a toggle). Checkbuttons are used all the time when a user is asked to choose between e.g. two different values for an option.



Checkbutton Widgets

Checkbuttons are created using the **`ttk::checkbutton`** command, and typically set up at the same time:

```
ttk::checkbutton .check -text "Use Metric" -command "metricChanged"
            -variable measuresystem -onvalue metric -offvalue imperial
```

Checkbuttons use many of the same options as regular buttons, but add a few more. The `"text"`, `"textvariable"`, `"image"`, and `"compound"` options control the display of the label (next to the check box itself), and the `"state"` and `"instate"` methods allow you to manipulate the `"disabled"` state flag to enable or disable the checkbutton. Similarly, the `"command"` option lets you specify a script to be called everytime the user toggles the checkbutton, and the `"invoke"` method will also execute the same callback.

Widget Value

Unlike buttons, checkbuttons also hold a value. We've seen before how the `"textvariable"` option can be used to tie the label of a widget to a variable in your program; the `"variable"` option for checkbuttons behaves similarly, except it is used to read or change the current value of the widget, and updates whenever the widget is toggled. By default, checkbuttons use a value of `"1"` when the widget is checked, and `"0"` when not checked, but these can be changed to just about anything using the `"onvalue"` and `"offvalue"` options.

What happens when the linked variable contains neither the on value or the off value (or even doesn't exist)? In that case, the checkbutton is put into a special "tristate" or indeterminate mode; you'll sometimes see this in user interfaces where the checkbox holds a single dash rather than being empty or holding a check mark. When in this state, the state flag `"alternate"` is set, so you can check for it with the `"instate"` method:

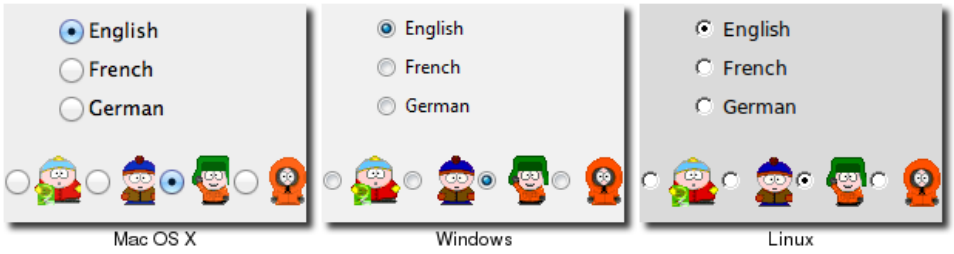
```
.check instate alternate
```

Because the checkbutton won't automatically set (or create) the linked variable, your program needs to make sure it sets the variable to the appropriate starting value.

Radiobutton

A **radiobutton** lets you choose between one of a number of mutually exclusive choices; unlike a checkbutton, it is not limited to just two choices. Radiobuttons are always used together in a set, and are a good option when the number of choices is fairly small, e.g. 3-5.

- Widget Roundup
- Reference Manual



Radiobutton Widgets

Radiobuttons are created using the **`ttk::radiobutton`** command, typically as a set:

```
ttk::radiobutton .home -text "Home" -variable phone -value home
ttk::radiobutton .office -text "Office" -variable phone -value office
ttk::radiobutton .cell -text "Mobile" -variable phone -value cell
```

Radiobuttons share most of the same configuration options as checkbuttons. One exception is that the `"onvalue"` and `"offvalue"` options are replaced with a single `"value"` option. Each of the radiobuttons of the set will have the same linked variable, but a different value; when the variable has the given value, the radiobutton will be selected, otherwise unselected. When the linked variable does not exist, radiobuttons also display a "tristate" or indeterminate, which can be checked via the `"alternate"` state flag.

Entry

An **entry** presents the user with a single line text field that they can use to type in a string value. These can be just about anything: their name, a city, a password, social security number, and so on.

- Widget Roundup
- Reference Manual

Entry Widgets

Entries are created using the **`ttk::entry`** command:

```
ttk::entry .name -textvariable username
```

A `"width"` configuration option may be specified to provide the number of characters wide the entry should be, allowing you for example to provide a shorter entry for a zip or postal code.

We've seen how checkbutton and radiobutton widgets have a value associated with them. Entries do as well, and that value is normally accessed through a linked variable specified by the `"textvariable"` configuration option. Note that unlike the various buttons, entries don't have a separate text or image beside them to identify them; use a separate label widget for that.

You can also get or change the value of the entry widget directly, without going through the linked variable. The `"get"` method returns the current value, and the `"delete"` and `"insert"` methods let you change the contents, e.g.

```
puts "current value is [.name get]"
.name delete 0 end           ; # delete between two indices, 0-based
.name insert 0 "your name"   ; # insert new text at a given index
```

Note that entry widgets do not have a `"command"` option which will invoke a callback whenever the entry is changed. To watch for changes, you should watch for changes on the linked variable. See also "Validation", below.

Passwords

Entries can be used for passwords, where the actual contents are displayed as a bullet or other symbol. To do this, set the `"show"` configuration option to the character you'd like to display, e.g. `"*"`.

Widget States

Like the various buttons, entries can also be put into a disabled state via the `"state"` command (and queried with `"instate"`). Entries can also use the state flag `"readonly"`; if set, users cannot change the entry, though they can still select the text in it (and copy it to the clipboard). There is also an `"invalid"` state, set if the entry widget fails validation, which leads us to...

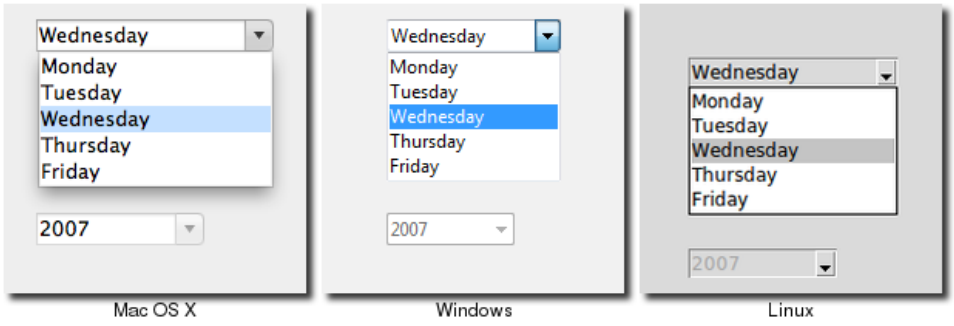
Validation

> *validate (controls overall validation behavior) - none (default), key (on each keystroke, runs before - prevalidation), focus/focusin/focusout (runs after.. revalidation), all*
> ** validatecommand script (script must return 1 or 0)*
> ** invalidcommand script (runs when validate command returns 0)*
> *- various substitutions in scripts.. most useful %P (new value of entry), %s (value of entry prior to editing)*
> *- the callbacks can also modify the entry using insert/delete, or modify -textvariable, which means the in progress edit is rejected in any case (since it would overwrite what we just set)*
> ** .e validate to force validation now*

Combobox

A **combobox** combines an entry with a list of choices available to the user. This lets them either choose from a set of values you've provided (e.g. typical settings), but also put in their own value (e.g. for less common cases you don't want to include in the list).

- Widget Roundup
- Reference Manual



Combobox Widgets

Comboboxes are created using the **`ttk::combobox`** command:

```
ttk::combobox .country -textvariable country
```

Like entries, the `"textvariable"` option links a variable in your program to the current value of the combobox. As with other widgets, the linked variable will not be set automatically. You can also get the current value using the `"get"` method, and change the current value using the `"set"` method (which takes a single argument, the new value).

A combobox will generate a `"<ComboboxSelected>"` virtual event that you can bind to whenever its value changes.

```
bind .country <<ComboboxSelected>> { script }
```

Predefined Values

You can provide a list of values the user can choose from using the `"values"` configuration option:

```
.country configure -values [list USA Canada Australia]
```

If set, the `"readonly"` state flag will restrict the user to making choices only from the list of predefined values, but not be able to enter their own (though if the current value of the combobox is not in the list, it won't be changed).

As a complement to the `"get"` and `"set"` methods, you can also use the `"current"` method to determine which item in the predefined values list is selected (call `"current"` with no arguments, it will return a 0-based index into the list, or -1 if the current value is not in the list), or select one of the items in the list (call `"current"` with a single 0-based index argument).

> *Want to associate some other value with each item in the list, so that your program can refer to some actual meaningful value, but it gets displayed in the combobox as something else? You'll want to have a look at the section entitled "Keeping Extra Item Data" when we get to the discussion of listboxes in a couple of chapters from now.*

The Grid Geometry Manager

We'll take a bit of a break from talking about different widgets (what to put onscreen), and focus instead on geometry management (where to put it). We introduced the general idea of geometry management in the "Tk Concepts" chapter; here, we focus on one specific geometry manager: grid.

As you've seen, grid lets you layout widgets in columns and rows. If you're familiar with using HTML tables to do layout, you'll feel right at home here. This chapter describes the various ways you can tweak grid to give you all the control you need for your user interface.

Grid is one of several geometry managers available in Tk, but it's mix of power, flexibility and ease of use, along with its natural fit with today's layouts (that rely on alignment of widgets) make it the best choice for general use. There are other geometry managers: "pack" is also quite powerful, but harder to use and understand; "place" gives you complete control of positioning each element; we'll see even widgets like paned windows, notebooks, canvas and text can act as geometry managers.

> *Grid was first introduced to Tk in 1996, several years after Tk became popular, and took a while to catch on. Before that, developers had always used "pack" to do constraint-based geometry management. When grid came out, many developers kept using pack, and you'll still find it used in many Tk programs and documentation. While there's nothing technically wrong with it, the algorithm's behavior is often hard to understand. More importantly, because the order that widgets are packed is significant in determining layout, modifying existing layouts can be more difficult.*
>
> *Grid has all the power of pack, generally produces nicer layouts (because it makes it easy to align widgets both horizontally and vertically), and is easier to learn and use. Because of that, we think grid is the right choice for most developers most of the time. Start your new programs using grid, and switch old ones to grid as you're making changes to an existing user interface.*

The reference documentation for grid provides an exhaustive description of grid, its behaviors and all options.

Columns and Rows

Using grid, widgets are assigned a `"column"` number and a `"row"` number, which indicates their relative position to each other. All widgets in the same column will therefore be above or below each other, while those in the same row will be to the left or right of each other.

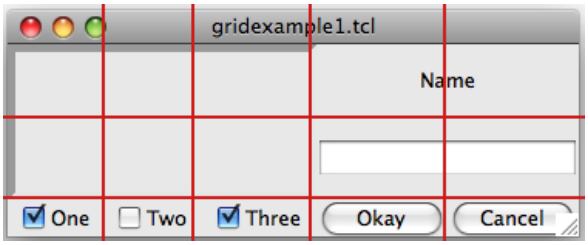
Column and row numbers must be integers, with the first column and row starting at 0. You can leave gaps in column and row numbers (e.g. column 0, 1, 2, 10, 11, 12, 20, 21), which is handy if you plan to add more widgets in the middle of the user interface at a later time.

The width of each column (or height of each row) depends on the width or height of the widgets contained within the column or row. This means when sketching out your user interface, and dividing it into rows and columns, you don't need to worry about each column or row being equal width.

Spanning Multiple Cells

Widgets can take up more than a single cell in the grid; to do this, you'll use the `"columnspan"` and `"rowspan"` options when gridding the widget. These are analogous to the "colspan" and "rowspan" attribute of HTML tables.

Here is an example of creating a user interface that has multiple widgets, some that take up more than a single cell.



Gridding multiple w idgets

```
ttk::frame .c
ttk::frame .c.f -borderwidth 5 -relief sunken -width 200 -height 100
ttk::label .c.namelbl -text Name
ttk::entry .c.name
ttk::checkbutton .c.one -text One -variable one -onvalue 1; set one 1
ttk::checkbutton .c.two -text Two -variable two -onvalue 1; set two 0
ttk::checkbutton .c.three -text Three -variable three -onvalue 1; set three 1
ttk::button .c.ok -text Okay
ttk::button .c.cancel -text Cancel

grid .c -column 0 -row 0
grid .c.f -column 0 -row 0 -columnspan 3 -rowspan 2
grid .c.namelbl -column 3 -row 0 -columnspan 2
grid .c.name -column 3 -row 1 -columnspan 2
grid .c.one -column 0 -row 3
grid .c.two -column 1 -row 3
grid .c.three -column 2 -row 3
grid .c.ok -column 3 -row 3
grid .c.cancel -column 4 -row 3
```

Layout within the Cell

Because the width of a column (and height of a row) depends on all the widgets that have been added to it, the odds are that at least some widgets will have a smaller width or height than has been allocated for the cell its been placed in. So the question becomes, where exactly should it be put within the cell?

By default, if a cell is larger than the widget contained in it, the widget will be centered within it, both horizontally and vertically, with the master's background showing in the empty space around it. The `"sticky"` option can be used to change this default behavior.

The value of the `"sticky"` option is a string of 0 or more of the compass directions "nsew", specifying which edges of the cell the widget should be "stuck" to. So for example, a value of `"n"` (north) will jam the widget up against the top side, with any extra vertical space on the bottom; the widget will still be centered horizontally. A value of `"nw"` (north-west) means the widget will be stuck to the top left corner, with extra space on the bottom and right.

Specifying two opposite edges, such as `"we"` (west, east) means that the widget will be stretched, in this case so it is stuck both to the left and right edge. So the widget will then be wider than its "ideal" size. Most widgets have options that can control how they are displayed if they are larger than needed. For example, a label widget has an `"anchor"` option which controls where the text of the label will be positioned.

If you want the widget to expand to fill up the entire cell, grid it with a sticky value of `"nsew"` (north, south, east, west) meaning it will stick to every side.

Handling Resize

If you've taken a peek below and added the extra `"sticky"` options to our example, when you try it out you'll notice things still don't look quite right (the entry is lower on the screen then we'd want), and things are even worse if you try to resize the window — nothing moves at all!

It looks like "sticky" may tell Tk *how* to react if the cell's row or column does resize, but doesn't actually say that the row or columns *should* resize if extra room becomes available. Let's fix that.

Every column and row has a `"weight"` grid option associated with it, which tells it how much it should grow if there is extra room in the master to fill. By default, the weight of each column or row is 0, meaning don't expand to fill space.

For the user interface to resize then, we'll need to give a positive weight to the columns we'd like to expand. This is done using the `"columnconfigure"` and `"rowconfigure"` methods of grid. If two columns have the same weight, they'll expand at the same rate; if one has a weight of 1, another of 3, the latter one will expand three pixels for every one pixel added to the first.

Both `"columnconfigure"` and `"rowconfigure"` also take a `"minsize"` grid option, which specifies a minimum size which you really don't want the column or row to shrink beyond.

Padding

Normally, each column or row will be directly adjacent to the next, so that widgets will be right next to each other. This is sometimes what you want (think of a listbox and its scrollbar), but often you want some space between widgets. In Tk, this is called padding, and there are several ways you can choose to add it.

We've already actually seen one way, and that is using a widget's own options to add the extra space around it. Not all widgets have this, but one that does is a frame; this is useful because frames are most often used as the master to grid other widgets. The frame's `"padding"` option lets you specify a bit of extra padding inside the frame, whether the same amount for each of the four sides, or even different for each.

A second way is using the `"padx"` and `"pady"` grid options when adding the widget. As you'd expect, `"padx"` puts a bit of extra space to the left and right of the widget, while `"pady"` adds extra space top and bottom. A single value for the option puts the same padding on both left and right (or top and bottom), while a two value list lets you put different amounts on left and right (or top and bottom). Note that this extra padding is within the grid cell containing the widget.

If you want to add padding around an entire row or column, the `"columnconfigure"` and `"rowconfigure"` methods accept a `"pad"` option, which will do this for you.

Let's add the extra sticky, resizing, and padding behavior to our example (additions in bold).

```
ttk::frame .c -padding "3 3 12 12"
ttk::frame .c.f -borderwidth 5 -relief sunken -width 200 -height 100
ttk::label .c.namelbl -text Name
ttk::entry .c.name
ttk::checkbutton .c.one -text One -variable one -onvalue 1; set one 1
ttk::checkbutton .c.two -text Two -variable two -onvalue 1; set two 0
ttk::checkbutton .c.three -text Three -variable three -onvalue 1; set three 1
ttk::button .c.ok -text Okay
ttk::button .c.cancel -text Cancel

grid .c -column 0 -row 0 -sticky nsew
grid .c.f -column 0 -row 0 -columnspan 3 -rowspan 2 -sticky nsew
grid .c.namelbl -column 3 -row 0 -columnspan 2 -sticky nw -padx 5
grid .c.name -column 3 -row 1 -columnspan 2 -sticky new -pady 5 -padx 5
grid .c.one -column 0 -row 3
grid .c.two -column 1 -row 3
grid .c.three -column 2 -row 3
grid .c.ok -column 3 -row 3
grid .c.cancel -column 4 -row 3

grid columnconfigure . 0 -weight 1
grid rowconfigure . 0 -weight 1
grid columnconfigure .c 0 -weight 3
grid columnconfigure .c 1 -weight 3
grid columnconfigure .c 2 -weight 3
grid columnconfigure .c 3 -weight 1
grid columnconfigure .c 4 -weight 1
grid rowconfigure .c 1 -weight 1
```

This looks more promising. Play around with the example to get a feel for the resize behavior.

Grid example, handling in-cell layout and resize.

> *You'll notice the little resize gadget at the very bottom right of the window; while we're just taking the easy route and avoiding it with the extra padding, we'll see later how to better take it into account using a "sizegrip" widget.*

Additional Grid Features

As you could see from the grid reference , there are lots of other things you can do with grid. Here are a few of the more useful ones.

Querying and Changing Grid Options

Like widgets themselves, it's easy to introspect the various grid options, as well as change them; setting options when you first grid the widget is just a convenience, and you can certainly change them anytime you'd like.

The `"slaves"` method will tell you all the widgets that have been gridded inside a master, or optionally those within just a certain column or row. The `"info"` method will give you a list of all the grid options for a widget and their values. Finally, the `"configure"` method lets you change one or more grid options on a widget.

These are illustrated in this interactive session:

```
% grid slaves .c
.c.cancel .c.ok .c.three .c.two .c.one .c.name .c.namelbl .c.f
% grid slaves .c -row 3
.c.cancel .c.ok .c.three .c.two .c.one
% grid slaves .c -column 0
.c.one .c.f
% grid info .c.namelbl
-in .c -column 3 -row 0 -columnspan 2 -rowspan 1 -ipadx 0 -ipady 0 -padx 5 -pady 0 -sticky nw
% grid configure .c.namelbl -sticky ew
% grid info .c.namelbl
-in .c -column 3 -row 0 -columnspan 2 -rowspan 1 -ipadx 0 -ipady 0 -padx 5 -pady 0 -sticky ew
```

Internal Padding

You saw how the `"padx"` and `"pady"` grid options added extra space around the outside of a widget. There's also a less used type of padding called "internal padding", which is controlled by the grid options `"ipadx"` and `"ipady"`.

The difference can be subtle. Let's say you have a frame that's 20x20, and specify normal (external) padding of 5 pixels on each side. The frame will request a 20x20 rectangle (its natural size) from the geometry manager. Normally, that's what it will be granted, so it'll get a 20x20 rectangle for the frame, surrounded by a 5 pixel border.

With internal padding, the geometry manager will effectively add the extra padding to the widget when figuring out its natural size, as if the widget has requested a 30x30 rectangle. If the frame is centered, or attached to a single side or corner (using `"sticky"`), you'll end up with a 20x20 frame with extra space around it. If however the frame is set to stretch (i.e. a `"sticky"` value of `"we"`, `"ns"`, or `"nwes"`) it will fill the extra space, resulting in a 30x30 frame, with no border.

Forget and Remove

The `"forget"` method of grid, taking as arguments a list of one or more slave widgets, can be used to remove slaves from the grid they're currently part of. This does not destroy the widget altogether, but takes it off the screen, as if it had not been gridded in the first place. You can grid it again later, though any grid options you'd originally assigned will have been lost.

The `"remove"` method of grid works the same, except that the grid options will be remembered.

Nested Layouts

As your user interface gets more complicated, the grid that you're using to organize all your widgets can get more and more complicated, and more fine-grained. This can make changing and maintaining your program very difficult.

Luckily, you don't have to manage your entire user interface with a single grid. If you have one area of your user interface that is fairly independent of others, create a new frame to hold that area, and grid the widgets that are part of that area within that frame. If you had a graphics program of some kind with multiple palletes, toolbars, and so on, each one of those areas might be a candidate for putting in its own frame.

In theory, these frames, each with its own grid, can be nested arbitrarily deep, though in practice this usually doesn't go beyond a few levels. This can be a big help in modularizing your program. If for example you have a pallete of drawing tools, you can create the whole thing in a separate procedure, including creating all the component widgets, gridding them together, setting up event bindings, and so on. From the point of view of your main program, all it needs to see is the single frame widget containing it all.

Our examples have shown just a hint of this, where a content frame was gridded into the main window, and then all the other widgets gridded into the content frame.

More Widgets

This chapter carries on introducing several more widgets: listbox, scrollbar, text, progressbar, scale and spinbox. Some of these are starting to be a bit more powerful than the basic ones we looked at before. Here we'll also see a few instances of using the classic Tk widgets, in instances where there isn't (or there isn't a need for) a themed counterpart.

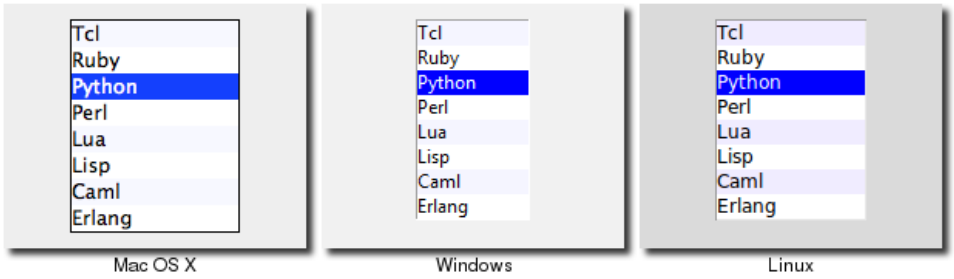
Listbox

- Widget Roundup
- Reference Manual

A **listbox** displays a list of single-line text items, usually lengthy, and allows the user to browse through the list, selecting one or more.

Listboxes are part of the classic Tk widgets; there is not presently a listbox in the themed Tk widget set.

> *Tk's treeview widget (which is themed) can also be used as a listbox (a one level deep tree), allowing you to use icons and styles with the list. It's also likely that a multi-column (table) list widget will make it into Tk at some point, based on one of the available extensions.*



Listbox Widgets

Listboxes are created using the **`tk::listbox`** command:

```
tk::listbox .l -height 10
```

Populating the Listbox Items

There's an easy way and a hard way to populate and manage all the items that are contained in the listbox.

Here is the easy way. Each listbox has a `"listvariable"` configuration option, which allows you to link a variable (which must hold a list) to the listbox. Each element of this list is a string representing one item in the listbox. So to add, remove, or rearrange items in the listbox, you can simply manipulate this variable as you would any other list. Similarly, to find out which item is on the third line of the listbox, just look at the third element of the list variable.

> *The reason there is a hard way at all is because the `"listvariable"` option was only introduced in Tk 8.3. Before that, you were stuck with the hard way. Because using the list variable lets you use all the standard list operations, it provides a much simpler API, and is certainly an upgrade worth considering if you have listboxes doing things the older way.*

The older, harder way to do things is use a set of methods that are part of the listbox widget itself that operate on the (internal) list of items:

- The `"insert` *`idx item ?item... ?`*`"` method is used to add one or more items to the list; `"`*`idx`*`"` is a 0-based index indicating the position of the item before which the item(s) should be added; specify `"end"` to put the new items at the end of the list.
- Use the `"delete` *`first ?last?`*`"` method to delete one or more items from the list; `"`*`first`*`"` and `"`*`last`*`"` are indices as per the `"insert"` method.
- Use the `"get` *`first ?last?`*`"` method to return the contents of a single item at the given position, or a list of the items between `"`*`first`*`"` and `"`*`last`*`"`.
- The `"size"` method returns the number of items in the list.

Selecting Items

The first thing you need to decide is whether it is possible for the user to select only a single item at a time, or if multiple items can simultaneously be selected. This is controlled by the `"selectmode"` option: the default is only being able to select a single item (`"browse"`), while a `selectmode` of `"extended"` allows the user to select multiple items.

> *The names `"browse"` and `"extended"`, again for backwards compatibility reasons, are truly awful. This is made worse by the fact that there are two other modes, `"single"` and `"multiple"` which you* ***should not use*** *(they use an old interaction style that is inconsistent with modern user interface and platform conventions).*

To find out which item or items in the listbox the user has currently selected, use the `"curselection"` method, which will return the list of indices of all items currently selected; this may be an empty list, and for lists with a `selectmode` of `"browse"`, will never be longer than one item. You can also use the `"selection includes` *`index`*`"` method to check if the item with the given index is currently selected.

To programmatically change the selection, you can use the `"selection clear` *`first ?last?`*`"` method to deselect either a single item, or any within the range of indices specified. To select an item, or all items in a range, use the `"selection set` *`first ?last?`*`"` method. Both of these will not touch the selection of any items outside the range specified.

If you do change the selection, you should also make sure that the newly selected item is visible to the user (i.e. it is not scrolled out of view). To do this, use the "`see` *`index`*" method.

When the selection is changed by the user, a "`<ListboxSelect>`" virtual event is generated. You can bind to this to take any action you need. Depending on your application, you may also want to bind to a double-click "`Double-1`" event, and use it to invoke an action with the currently selected item.

Stylizing the List

Like most of the "classic" Tk widgets, you have immense flexibility in modifying the appearance of a listbox. As described in the reference manual , you can modify the font the listbox items are displayed in, the foreground (text) and background colors for items in their normal state, when selected, when the widget is disabled, and so on. There is also an "`itemconfigure`" method which allows you to change the foreground and background colors of individual items.

As is often the case, restraint is useful. Generally, the default values will be entirely suitable, and a good match for platform conventions. In the example we'll get to momentarily, we'll show how restrained use of these options can be put to good effect, in this case displaying alternate lines of the listbox in slightly different colors.

Keeping Extra Item Data

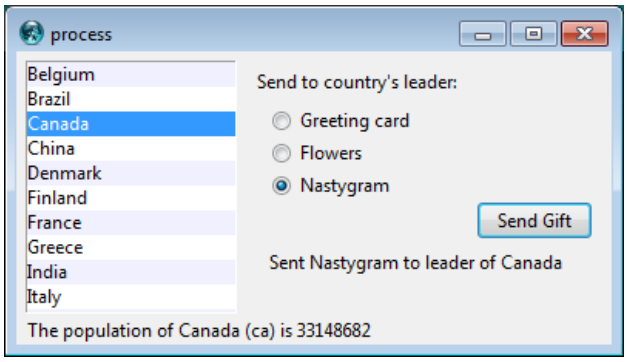
The "`listvariable`" (or the internal list, if you're managing things the old way) contains the strings that will be shown in the listbox. It's very often the case though that each string in your program is associated with some other data item, and what you're really interested in is not so much the string displayed in the listbox, but the associated data item. For example, a listbox containing names may be what is presented to the user, but your program is really interested in the user object (or id number) that is selected, not the particular name.

How can we associate this value with the displayed name? Unfortunately, the listbox widget itself doesn't offer any facilities, so it's something we'll have to manage separately. There are a couple of obvious approaches. First, if the displayed strings are guaranteed unique, you could use a hash table to map the names to the associated object. So, given the name you can easily get the associated object. This probably wouldn't work well for doing names, but could work for choosing countries for example.

A second approach is to keep a second list, parallel to the list of strings displayed in the listbox, which will hold the associated objects. So the first item in the strings list corresponds to the first item in the objects list, the second to the second, and so on. Any changes you make in one list (insert, delete, reorder) you need to make in the other. You can then easily map from the list item to the underlying object, based on their position in the list.

Example

Here is a silly example showing these various listbox techniques. We'll have a list of countries displayed. Behind the scenes, we have a database (a simple hash table) which contains the population of each country, indexed by the two letter country code. We'll be able to select only a single country at a time, and as we do so, a status bar will display the population of the country. Double-clicking on the list, or hitting the Return key, will send one of several gifts to the selected country's head of state (well, not really, but use your imagination).



Country Selector Listbox Example

```
# Initialize our country "databases":
#  - the list of country codes (a subset anyway)
#  - a parallel list of country names, in the same order as the country codes
#  - a hash table mapping country code to population<
set countrycodes [list ar au be br ca cn dk fi fr gr in it jp mx nl no es se ch]
set countrynames [list Argentina Australia Belgium Brazil Canada China Denmark \
        Finland France Greece India Italy Japan Mexico Netherlands Norway Spain \
        Sweden Switzerland]
array set populations [list ar 41000000 au 21179211 be 10584534 br 185971537 \
        ca 33148682 cn 1323128240 dk 5457415 fi 5302000 fr 64102140 gr 11147000 \
        in 1131043000 it 59206382 jp 127718000 mx 106535000 nl 16402414 \
        no 4738085 es 45116894 se 9174082 ch 7508700]

# Names of the gifts we can send
array set gifts [list card "Greeting card" flowers "Flowers" nastygram "Nastygram"]
```

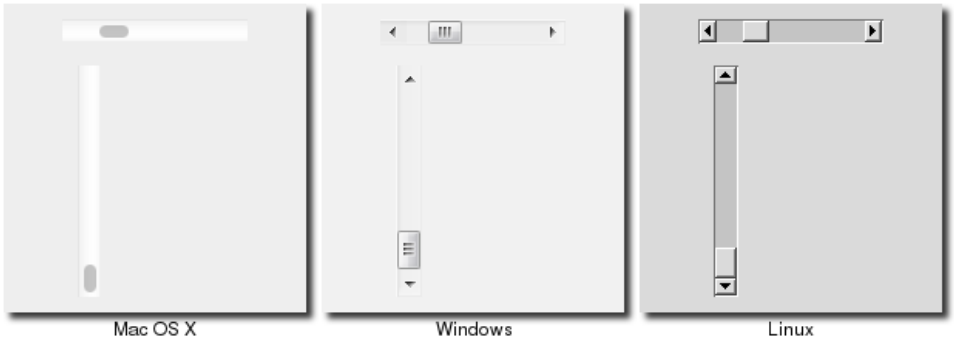
```
# Create and grid the outer content frame
grid [ttk::frame .c -padding "5 5 12 0"] -column 0 -row 0 -sticky nwes
grid columnconfigure . 0 -weight 1; grid rowconfigure . 0 -weight 1

# Create the different widgets; note the variables that many
# of them are bound to, as well as the button callback.
# The listbox is the only widget we'll need to refer to directly
# later in our program, so for convenience we'll assign it to a variable.
set ::lbox [tk::listbox .c.countries -listvariable countrynames -height 5]
ttk::label .c.lbl -text "Send to country's leader:"
ttk::radiobutton .c.g1 -text $gifts(card) -variable gift -value card
ttk::radiobutton .c.g2 -text $gifts(flowers) -variable gift -value flowers
ttk::radiobutton .c.g3 -text $gifts(nastygram) -variable gift -value nastygram
ttk::button .c.send -text "Send Gift" -command {sendGift} -default active
ttk::label .c.sentlbl -textvariable sentmsg -anchor center
ttk::label .c.status -textvariable statusmsg -anchor w

# Grid all the widgets
grid .c.countries -column 0 -row 0 -rowspan 6 -sticky nsew
grid .c.lbl       -column 1 -row 0 -padx 10 -pady 5
grid .c.g1        -column 1 -row 1 -sticky w -padx 20
grid .c.g2        -column 1 -row 2 -sticky w -padx 20
grid .c.g3        -column 1 -row 3 -sticky w -padx 20
grid .c.send      -column 2 -row 4 -sticky e
grid .c.sentlbl   -column 1 -row 5 -columnspan 2 -sticky n -pady 5 -padx 5
grid .c.status    -column 0 -row 6 -columnspan 2 -sticky we
grid columnconfigure .c  0 -weight 1; grid rowconfigure .c 5 -weight 1

# Set event bindings for when the selection in the listbox changes,
# when the user double clicks the list, and when they hit the Return key
bind $::lbox <<ListboxSelect>> "showPopulation"
bind $::lbox <Double-1> "sendGift"
bind . <Return> "sendGift"

# Called when the selection in the listbox changes; figure out
# which country is currently selected, and then lookup its country
# code, and from that, its population.  Update the status message
# with the new population.  As well, clear the message about the
# gift being sent, so it doesn't stick around after we start doing
# other things.
proc showPopulation {} {
    set idx [$::lbox curselection]
    if {[llength $idx]==1} {
        set code [lindex $::countrycodes $idx]
        set name [lindex $::countrynames $idx]
        set popn $::populations($code)
        set ::statusmsg "The population of $name ($code) is $popn"
    }
    set ::sentmsg ""
}

# Called when the user double clicks an item in the listbox, presses
# the "Send Gift" button, or presses the Return key.  In case the selected
# item is scrolled out of view, make sure it is visible.
#
# Figure out which country is selected, which gift is selected with the
# radiobuttons, "send the gift", and provide feedback that it was sent.
proc sendGift {} {
    set idx [$::lbox curselection]
    if {[llength $idx]==1} {
        $::lbox see $idx
        set name [lindex $::countrynames $idx]
        # Gift sending left as an exercise to the reader
        set ::sentmsg "Sent $::gifts($::gift) to leader of $name"
    }
}

# Colorize alternating lines of the listbox
for {set i 0} {$i<[llength $countrynames]} {incr i 2} {
    $::lbox itemconfigure $i -background #f0f0ff
}

# Set the starting state of the interface, including selecting the
# default gift to send, and clearing the messages.  Select the first
# country in the list; because the <<ListboxSelect>> event is only
# generated when the user makes a change, we explicitly call showPopulation.
set gift card
set sentmsg ""
set statusmsg ""
$::lbox selection set 0
showPopulation
```

One obvious thing missing from this example was that while the list of countries could be quite long, only part of it fit on the screen at once. To show countries further down in the list you had to either drag with your mouse or use the down arrow key. A scrollbar would have been nice. Let's fix that.

Scrollbar

A **scrollbar** helps the user to see all parts of another widget, whose content is typically much larger than what can be shown in the available screen space.

- Widget Roundup
- Reference Manual



Scrollbar Widgets

Scrollbars are created using the **`ttk::scrollbar`** command:

```
ttk::scrollbar .s -orient vertical -command ".l yview"
.l configure -yscrollcommand ".s set"
```

Unlike in some toolkits, scrollbars are not a part of another widget (e.g. a listbox), but are a separate widget altogether. Instead, scrollbars communicate with the scrolled widget by calling methods on the scrolled widget; as it turns out, the scrolled widget also needs to call methods on the scrollbar.

The `"orient"` configuration option of scrollbars determines whether it will be used to scroll in the `"horizontal"` or `"vertical"`. You then need to set up the `"command"` configuration option to communicate with the scrolled widget. This needs to be the method to call on the scrolled widget.

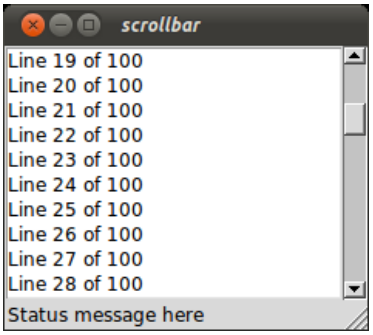
Every widget that can be scrolled vertically includes a method named `"yview"` (those that can be scrolled horizontally have a method named "`xview`"). As long as this method is present, the scrollbar doesn't need to know anything else about the scrolled widget. When the scrollbar is manipulated, it will tack on some number of parameters to the method call, indicating how it was scrolled, to what position, etc.

The scrolled widget also needs to communicate back to the scrollbar, telling it what percentage of the entire content area is now visible. Besides the `yview` and/or `xview` methods, every scrollable widget also has a `"yscrollcommand"` and/or `"xscrollcommand"` configuration option. This is used to specify a method call, which must be the scrollbar's `"set"` method. Again, additional parameters will be automatically tacked onto the method call.

> *If for some reason you want to move the scrollbar to a particular position from within your program, you can call the `"set first last"` method yourself. Pass it two values between 0 and 1 indicating the start and end percentage of the content area that is visible.*

Example

Listboxes are one of several types of widgets that are scrollable. Here we'll build a very simple user interface, consisting just of a vertically scrollable listbox that takes up the entire window, with just a status line at the bottom.



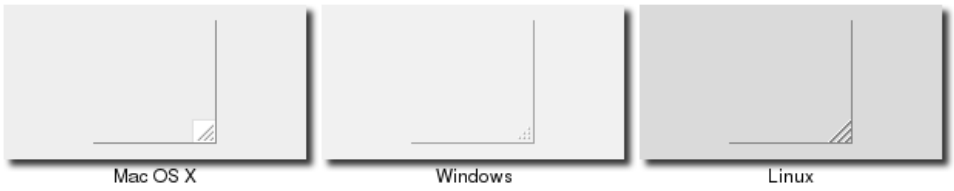
Scrolling a Listbox

```
grid [tk::listbox .l -yscrollcommand ".s set" -height 5] -column 0 -row 0 -sticky nwes
grid [ttk::scrollbar .s -command ".l yview" -orient vertical] -column 1 -row 0 -sticky ns
grid [ttk::label .stat -text "Status message here" -anchor w] -column 0 -row 1 -sticky we
grid [ttk::sizegrip .sz] -column 1 -row 1 -sticky se
grid columnconfigure . 0 -weight 1; grid rowconfigure . 0 -weight 1
for {set i 0} {$i<100} {incr i} {
   .l insert end "Line $i of 100"
}
```

SizeGrip

We actually snuck in one new widget in that last example, the **sizegrip**. This is the little box at the bottom right corner of the window that allows you to resize it.

- Widget Roundup
- Reference Manual



SizeGrip Widgets

SizeGrips are created using the **ttk::sizegrip** command:

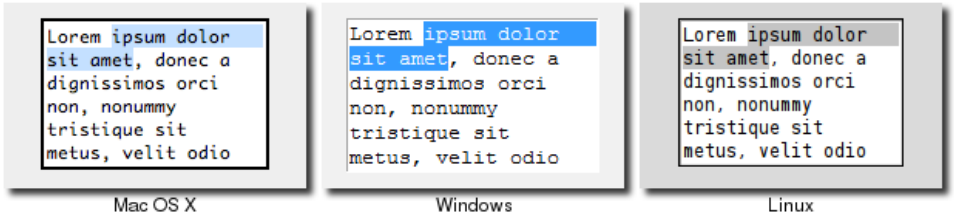
```
grid [ttk::sizegrip .sz] -column 999 -row 999 -sticky se
```

While you'll notice that on some platforms (e.g. Mac OS X), Tk will automatically put the size grip there for you, it doesn't hurt to explicitly add it yourself. We'll discuss how to change the window size, determine if it's resizable, etc. in a later chapter.

Text

A **text** widget provides users with an area so that they can enter multiple lines of text. Text widgets are part of the classic Tk widgets, not the themed Tk widgets.

- Widget Roundup
- Reference Manual



Text Widgets

> *Tk's text widget is, along with the canvas widget, one of two uber-powerful widgets that provide amazingly deep but easily programmed features. Text widgets have formed the basis for full word processors, outliners, web browsers and more. We'll get into some of the advanced stuff in a later chapter, but here we'll just show you what you need to use the text widget to capture fairly simple, multi-line text input.*

Text widgets are created using the **tk::text** command:

```
tk::text .t -width 40 -height 10
```

The `"width"` and `"height"` options specify the requested screen size of the text widget, in characters and rows respectively. The contents of the text can be arbitrarily large. You can use the `"wrap"` configuration option to control how line wrapping is handled: values are `"none"` (no wrapping, text may horizontally scroll), "char" (wrap at any character), and `"word"` (wrapping will only occur at word boundaries).

A text widget can be disabled so that no editing can occur; because text is not a themed widget, the usual `"state"` and `"instate"` methods are not available. Instead, use the configuration option `"state"`, setting it to either `"disabled"` or `"normal"`.

Scrolling works the same way as in listboxes. The `"xscrollcommand"` and `"yscrollcommand"` configuration options

are available to attach the text widget to horizontal and/or vertical scrollbars, and the `"xview"` and `"yview"` methods are available to be called from scrollbars. To ensure that a given line is visible (i.e. not scrolled out of view), you can use the `"see index"` method, where *index* is in the form *"line number.character number"*, e.g. `"5.0"` for the first (0-based) character of line 5 (1-based).

Contents

Text widgets do not have a linked variable associated with them, like for example entry widgets do. To retrieve the text content for the entire widget, call the method `"get 1.0 end"`; the `"1.0"` is an index into the text, and means the first character of the first line, and `"end"` is a shortcut for the index of the last character, last line. Other indices could be passed to retrieve smaller ranges of text if needed.

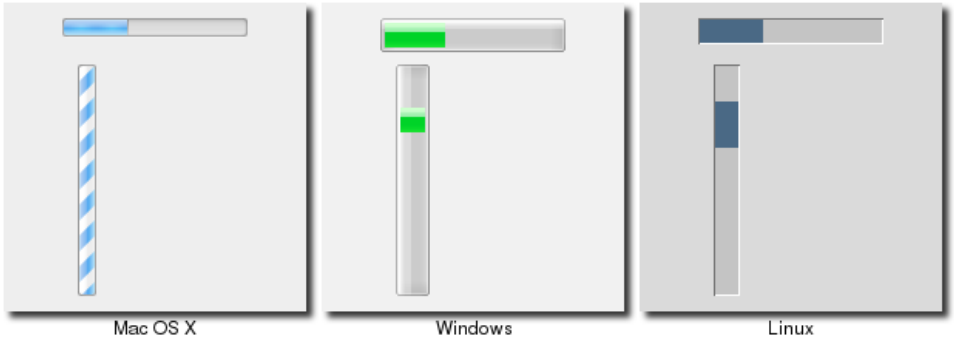
Text can be added to the widget using the `"insert index string"` method; again *index* is in the form `"line.char"` and marks the character before which text is inserted; use `"end"` to add text to the end of the widget. You can delete a range of text using the `"delete start end"` method, where both *start* and *end* are text indices as already described.

We'll get into the text widget's many additional advanced features in a later chapter.

Progressbar

- Widget Roundup
- Reference Manual

A **progressbar** widget provides a way to give feedback to the user about the progress of a lengthy operation. This may be done either as a percentage complete display for operations where this can be estimated, or a display that changes to indicate the operation is continuing, but without an estimate of completion.



Progressbar Widgets

Progessbar widgets are created using the **`ttk::progressbar`** command:

```
ttk::progressbar .p -orient horizontal -length 200 -mode determinate
```

The `"orient"` option may be either `"horizontal"` or `"vertical"`. The `"length"` option, which represents the longer axis of either horizontal or vertical progressbars, is specified in screen units (e.g. pixels). The `"mode"` configuration option can be set to either `"determinate"`, where the progressbar will indicate relative progress towards completion, or to `"indeterminate"`, where its not possible to know how far along in the task the program is, but we still want to provide feedback that things are still running.

Determinate Progess

In determinate mode, you're able to provide more-or-less accurate feedback to the user about how far an operation has progressed. To do this, you need to first of all tell the progressbar how many "steps" the operation will take, and then as you go along, tell the progressbar how far along the operation is.

You can provide the total number of steps to the progressbar using the `"maximum"` configuration option; this is a floating point number that defaults to 100 (i.e. each step is 1%). To tell the progressbar how far along you are in the operation, you will repeatedly change the `"value"` configuration option. So this would start at 0, and then count upwards to the maximum value you have set. There are two slight variations on this. First, you can just store the current value for the progressbar in a variable linked to it by the progressbar's `"variable"` configuration option; that way, when you change the variable, the progressbar will update. The other alternative is to call the progressbar's `"step ?amount?"` method to increment the value by the given *"amount"* (defaults to 1.0).

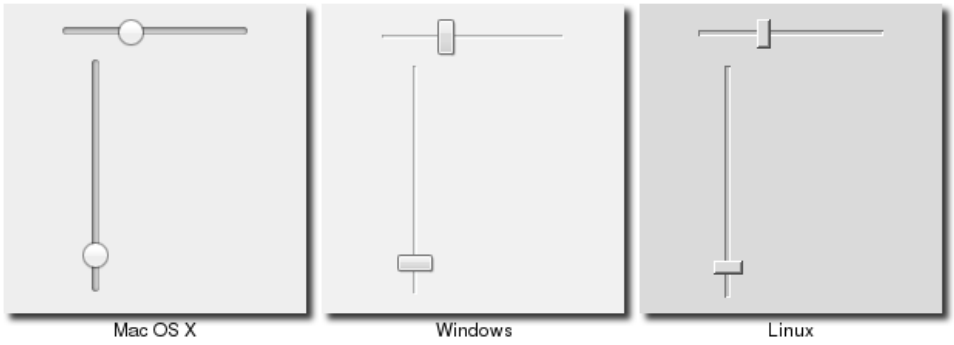
Indeterminate Progress

Indeterminate mode is for when you're not able to easily know (or estimate) how far along in a long running task you actually are, but still want to provide feedback to the user that the operation is still running (or that your program hasn't crashed). Rather than providing specific values to indicate progress along the way, at the start of the operation you'll just call the progressbar's `"start"` method, and at the end of the operation, you'll call its `"stop"` method. The progressbar will take care of the rest.

Scale

A **scale** widget provides a way for users to choose a numeric value through direct manipulation.

- Widget Roundup
- Reference Manual



Scale Widgets

Scale widgets are created using the **`ttk::scale`** command:

```
ttk::scale .s -orient horizontal -length 200 -from 1.0 -to 100.0
```

In some ways, scale widgets are like progressbars, except they are designed for the user to manipulate them. As with progressbars, they should be given an orientation (horizontal or vertical) with the `"orient"` configuration option, and an optional `"length"`. You should also define the range of the number that the scale allows users to choose; to do this, set a floating point number for each of the `"from"` and `"to"` configuration options.

There are several different ways you can set the current value of the scale (which must be a floating point value between the `"from"` and `"to"` values). You can set (or read, to get the current value) the scale's `"value"` configuration option. You can link the scale to a variable using the `"variable"` option. Or, you can call the scale's `"set` *`value`*`"` method to change the value, or the `"get"` method to read the current value.

There is a `"command"` configuration option, which lets you specify a script to call whenever the scale is changed. Tk will automatically append the current value of the scale as a parameter each time it invokes this script (we saw a similar thing with extra parameters being added to scrollbar callbacks and those on the widgets they scroll).

As with other themed widgets, you can use the `"state disabled"`, `"state !disabled"` and `"instate disabled"` methods if you wish to prevent the user from modifying the scale.

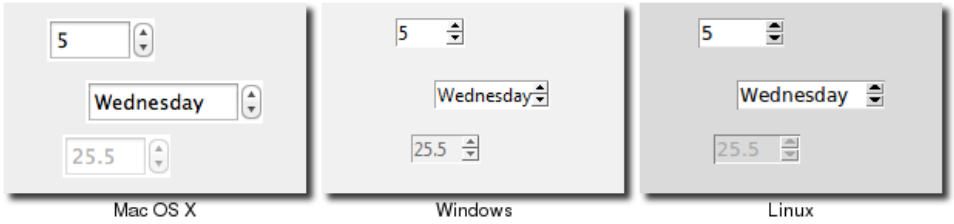
> *As the scale widget does not display the actual values, you may want to add those as labels.*

Spinbox

A **spinbox** widget allows users to choose numbers (or in fact, items from an arbitrary list). It does this by combining an entry-like widget showing the current value with a pair of small up/down arrows which can be used to step through the range of possible choices.

- Widget Roundup
- Reference Manual

Spinboxes are part of the classic Tk widgets; there is not presently a spinbox in the themed Tk widget set.



Spinbox Widgets

Spinbox widgets are created using the **`tk::spinbox`** command:

```
tk::spinbox .s -from 1.0 -to 100.0 -textvariable spinval
```

Like scale widgets, spinboxes are normally used to let the user choose a number between a certain range (specified using the `"from"` and `"to"` configuration options), though through a very different user interface. You can also specify a `"increment"`, which controls how much the value changes everytime you click the up or down button.

Like a listbox or combobox, spinboxes can also be used to let the user choose an item from an arbitrary list of strings; these can be specified using the `"values"` configuration option, which works in the same was as in comboboxes. Specifying a list of values will override to `"from"` and `"to"` settings.

> *You might be puzzled about when to choose a scale, listbox, combobox, entry or a spinbox, since usually several of these can be used for the same types of data. The answer really depends on the type of data you want the user to select, platform user interface conventions, and the role the value plays in your user interface.*
>
> *For example, both a combobox and a spinbox have the benefit of taking up fairly small amounts of space, which might make sense for a more peripheral setting, where the primary choice in a user interface may warrant the extra space a listbox occupies. Spinboxes don't make much sense when items don't have a natural and obvious ordering to them. You should be careful about both comboboxes and spinboxes that have too many items in them, which can make it more time consuming to select an item.*

Both both a numeric and arbitrary range, there is a `"wrap"` option which accepts a boolean, and determines whether the value should wrap around when it goes beyond the starting or ending values. You can also specify a `"width"` for the entry holding the current value of the spinbox.

Again there are choices as to how to set or get the current value in the spinbox. Normally, you would specify a linked variable with the `"textvariable"` configuration option; as usual, any changes to the variable are reflected in the spinbox, while any changes in the spinbox are reflected in the linked variable. As well, the `"set value"` and `"get"` methods allow you to set or get the value directly.

You can arranged to be called whenever the spinbox changes using the `"command"` configuration option.

> *The command has a couple of percent substitutions, %s = current value, and %d = up or down. Need to figure out the right way to specify this in Ruby. Also need to add stuff about validation.*

Because spinbox is not a themed widget, the `"state"` and `"instate"` methods are not available. Instead, you can change its state using the `"state"` configuration option. This may be `"normal"`, `"disabled"` to prevent any changes.

Menus

This chapter describes how to handle menubars and popup menus in Tk. For a polished application, these are areas you particularly want to pay attention to. Menus need special care if you want your application to fit in with other applications on your users' platform.

Speaking of which, the recommended way to figure out which platform you're running on is:

```
tk windowingsystem;  # will return x11, win32 or aqua
```

> *This is probably more useful than examining global variables like `tcl_platform` or `RUBY_PLATFORM`, and older checks that used these methods should be examined. While in the olden days there was a pretty good correlation between platform and windowing system, it's less true today. For example, if your platform is identified on Unix, that might mean Linux under X11, Mac OS X under Aqua, or even Mac OS X under X11.*

Menubars

In this section we'll look at menubars: how to create them, what goes in them, how they're used, and so on.

Properly designing a menubar and its set of menus is beyond the scope of this tutorial, but a few pieces of advice. First, if you find yourself with a large number of menus, very long menus, or deeply nested menus, you may need to rethink how your user interface is organized. Second, many people use the menus to explore what the program can do, particularly when they're first learning it, so try to ensure major features are accessible by the menus. Finally, for each platform you're targeting, become familiar with how applications use menus, and consult the platform's human interface guidelines for full details about design, terminology, shortcuts, and much more. This is an area you will likely have to customize for each platform.

Menu Widgets and Hierarchy

- **Widget Roundup**
- **Reference Manual**

Menus are implemented as widgets in Tk, just like buttons and entries. Each menu widget consists of a number of different *items* in the menu. Items are things like the "Open..." command in a File menu, but also separators between other items, and items which open up their own submenu (so-called *cascading* menus). Each of these menu items also has attributes, such as the text to display for the item, a keyboard accelerator, and a command to invoke.

Menus are arranged in a hierarchy. The menubar is itself a menu widget. It has several children (submenus) consisting of items like "File", "Edit" and so on. Each of those in turn is a menu containing different items, some of which might themselves contain submenus. As you'd expect from other things you've seen already in Tk, anytime you have a submenu, it must be created as a child of its parent menu.

Before you Start

It's important to put the following line in your application somewhere before you start creating menus.

```
option add *tearOff 0
```

Without it, each of your menus (on Windows and X11) will start with what looks like a dashed line, and allows you to "tear off" the menu so it appears in its own window. You really don't want that there.

> *This is a throw-back to the Motif-style X11 that Tk's original look and feel were based on. Unless your application is designed to run on that old box collecting dust in the basement, you really don't want to see this, as its not a part of any modern user interface style.*

> *And we'll all be looking forward to a version of Tk where this backwards compatibility is not preserved, and the default is not to have these tear-off menus.*

Creating a Menubar

In Tk, menubars are associated with individual windows; each toplevel window can have at most one menubar. On Windows and X11, this is visually obvious, as the menus are part of each window, sitting just below the title bar at the top.

On Mac OS X though, there is a single menubar along the top of the screen, shared by each window. As far as your Tk program is concerned, each window still does have its own menubar; as you switch between windows, Tk will automatically take care of making sure that the correct menubar is displayed at the top of the screen. If you don't specify a menubar for a particular window, Tk will use the menubar associated with the root window; you'll have noticed by now that this is automatically created for you when your Tk application starts.

> *Because on Mac OS X all windows have a menubar, it's important to make sure you do define one, either for each window or a fallback menubar for the root window. Otherwise, you'll end up with the "built-in" menubar, which contains menus that are only intended for use when typing commands directly into the interpreter.*

To actually create a menubar for a window, we first create a menu widget, and then use the window's `"menu"` configuration option to attach the menu widget to the window. As noted, the menu widget must be a child of the toplevel window.

```
toplevel .win
menu .win.menubar
.win configure -menu .win.menubar
```

> *This is truly ancient history, but menubars used to be done by creating a frame widget containing the menu items, and packing it into the top of the window like you would any other widget. Hopefully you don't have any code or documentation that still does this.*

Adding Menus

We now have a menubar, but that's pretty useless without some menus to go in it. So again, we'll want to create a menu widget for each menu that will go in the menubar (each one a child of the menubar), and then add them all to the menubar.

```
set m .win.menubar
menu $m.file
menu $m.edit
$m add cascade -menu $m.file -label File
$m add cascade -menu $m.edit -label Edit
```

Adding Menu Items

Now that we have a couple of menus in our menubar, it's probably a good time to add a few items to each menu. Remember that menu items are part of the menu itself, so we thankfully don't have to go and create another menu widget for each one.

```
$m.file add command -label "New" -command "newFile"
$m.file add command -label "Open..." -command "openFile"
$m.file add command -label "Close" -command "closeFile"
```

> *On Mac OS X, the ellipsis ("...") is actually a special character, which is more tightly spaced than three periods in a row. Tk takes care of substituting this character for you automatically.*

So adding menu items to a menu is essentially the same as adding a submenu, but rather than adding a menu item of type "cascade", we're adding one of type "command".

Each menu item has associated with it a number of configuration options, in the same way widgets do, though each menu item type has a different set of relevant options. Cascade menu items have a `"menu"` option used to specify the submenu, command menu items have a `"command"` option used to specify the command to invoke when the item is selected, and both have a `"label"` option to specify the text to display for the item.

As well as adding items to the end of menus, you can also insert them in the middle of menus via the "`insert` *`index type ?option value...?`*" method; here "*`index`*" is the position (0..n-1) of the item you want to insert before. You can also delete a menu using the "`delete` *`index`*" method.

Types of Menu Items

We've already seen `"command"` menu items, which are the common menu items that when selected will invoke a command.

We've also seen the use of `"cascade"` menu items, used to add a menu to a menubar. Not surprisingly, if you want to add a submenu to an existing menu, you also use a `"cascade"` menu item, in exactly the same way.

A third type of menu item is the `"separator"`, which produces the dividing line you often see between different sets of menu items.

```
$m.file add separator
```

Finally, there are also `"checkbutton"` and `"radiobutton"` menu items, which behave analogously to checkbutton and radiobutton widgets. These menu items have a variable associated with them, and depending on the value of that variable, will display an indicator (i.e. a checkmark or a selected radiobutton) next to the item's label.

```
$m.file add checkbutton -label Check -variable check -onvalue 1 -offvalue 0
$m.file add radiobutton -label One -variable radio -value 1
$m.file add radiobutton -label Two -variable radio -value 2
```

When the user selects a checkbutton item that is not already checked, it will set the associated variable to the value in `"onvalue"`, while selecting a item that is already checked sets it to the value in `"offvalue"`. Selecting a radiobutton item sets the associated variable to the value in `"value"`. Both types of items also react to changes in the associated variable from within other parts of your program.

As with command items, checkbutton and radiobutton menu items do accept a `"command"` configuration option, that will be invoked when the menu item is selected; the associated variable, and hence the menu item's state, is updated before the callback is invoked.

> *Radiobutton menu items are not part of the Windows or Mac OS X human interface guidelines, so on those platforms the indicator next to the item's label is a checkmark, as it would be for a checkbutton item. The semantics still work though; it's a good way to select between multiple items, since the display will show one of the items selected (checked).*

Accelerator Keys

The `"accelerator"` option is used to indicate the menu accelerator that should be associated with this menu. This does not actually *create* the accelerator, but only displays what it is next to the menu item. You still need to create a binding for the accelerator yourself.

> *Remember that event bindings can be set on individual widgets, all widgets of a certain type, the toplevel window containing the widget you're interested in, or the application as a whole. As menu bars are associated with individual windows, normally the event bindings you create will be on the toplevel window the menu is associated with.*

Accelerators are very platform specific, not only in terms of which keys are used for what operation, but what modifier keys are used for menu accelerators (e.g. on Mac OS X it is the "Command" key, on Windows and X11 it is usually the "Control" key). Example of valid accelerator options are `"Command-N"`, `"Shift+Ctrl+X"`, and `"Command-Option-B"`. Commonly used modifiers include "Control", "Ctrl", "Option", "Opt", "Alt", "Shift", "Command", "Cmd" and "Meta").

> *On Mac OS X, those modifiers will be automatically mapped to the different modifier icons that appear in menus.*

More on Item Options

There are a few more common options for menu items.

Underline

While all platforms support keyboard traversal of the menubar via the arrow keys, on Windows and X11, you can also use other keys to jump to particular menus or menu items. The keys that trigger these jumps are indicated by an

underlined letter in the menu item's label. If you want to add one of these to a menu item, you can use the `"underline"` configuration option for the item. The value of this option should be the index of the character you'd like underlined (from 0 to the length of the string - 1).

Images

It is also possible to use images in menu items, either beside the menu item's label, or replacing it altogether. To do this, you can use the `"image"` and `"compound"` options, which work just like in label widgets. The value for `"image"` must be a Tk image object, while `"compound"` can have the values `"bottom"`, `"center"`, `"left"`, `"right"`, `"top"` or `"none"`.

State

It is also possible to disable a menu, so that the user cannot select it. This can be done via the `"state"` option, setting it to a value of `"disabled"`, or a value of `"normal"` to reenable the item.

Querying and Changing Item Options

Like most everything in Tk, you can look at or change the value of an item's options at any time. Items are referred to via an *index*. Normally, this is a number (0..n-1) indicating the item's position in the menu, but you can also specify the label of the menu item (or in fact, a "glob-style" pattern to match against the item's label).

```
puts [$m.file entrycget 0 -label]; # get label of top entry in menu
$m.file entryconfigure Close -state disabled; # change an entry
puts [$m.file entryconfigure 0]; # print info on all options for an item
```

Platform Menus

Each platform has a few menus that are handled specially by Tk.

Mac OS X

On Mac OS X, there are two menus that are treated specially. The application menu is the second menu in the menubar, titled with your application name. It will have any menu items you add to it, then following that, the standard application menu items (preferences, services, hide/show, and quit) are automatically added by Tk. The second special menu is the help menu, which is always placed as the last menu on the menubar. Again, along with the items you add, Tk will add others expected by the system (nothing on Tiger, a search box on Leopard).

```
$m add cascade -menu [menu $m.apple]
$m add cascade -menu [menu $m.help]
```

> *The pathnames of these menu widgets must be `".apple"` and `".help"`, as this is what causes the menus to be treated specially by Tk.*

> *The application menu, which is the one we're dealing with here, is distinct from the apple menu (the one with the apple icon, just to the left of the application menu). Despite that we do really mean the application menu, in Tk it is still referred to as the "apple" menu. This is a holdover from pre-OS X days, when these sorts of items did go in the actual apple menu, and there was no separate application menu.*

It's normal practice to add an "About *appname*" command item to the Application menu, followed by a separator item. When invoked this should display your application's about box.

Each application should respond to the "Preferences" item in the Application menu. To do so, you'll need to define a Tcl procedure named `"::tk::mac::ShowPreferences"`. This will be called when the Preferences menu item is chosen; if the procedure is not defined, the menu item will be disabled.

Windows

On Windows, each window has a "System" menu at the top left of the window frame, with a small icon for your application. It contains items like "Close", "Minimize", etc. In Tk, if you create a system menu, you can add new items that will appear below the standard items.

```
$m add cascade -menu [menu $m.system]
```

> *The pathname of the menu widget must be `".system"`.*

X11

On X11, if you create a help menu, Tk will ensure that it is always the last menu in the menubar.

```
$m add cascade -label Help -menu [menu $m.help]
```

> *The pathname of the menu widget must be "`.help`".*

Contextual Menus

Contextual menus ("popup" menus) are typically invoked by a right mouse button click on an object in the application. A menu pops up at the location of the mouse cursor, and the user can select from one of the items in the menu (or click outside the menu to dismiss it without choosing any item).

To create a contextual menu, you'll use exactly the same commands you did to create menus in the menubar. Typically, you'll create one menu with several command items in it, and potentially some cascade menu items and their associated menus.

To activate the menu, the user will use a contextual menu click, which you will have to bind to. That however, can mean different things on different platforms. On Windows and X11, this is the right mouse button being clicked (the third mouse button). On Mac OS X, this is either a click of the left (or only) button with the control key held down, or a right click on a multi-button mouse. Unlike Windows and X11, Mac OS X refers to this as the second mouse button, not the third, so that's the event you'll see in your program.

> *Most earlier programs that have used popup menus assumed it was only "button 3" they needed to worry about.*

Besides capturing the correct contextual menu event, you'll also need to capture the location the mouse was clicked. It turns out you need to do this relative to the entire screen (global coordinates) and not local to the window or widget you clicked on (local coordinates). The "%X" and "%Y" substitutions in Tk's event binding system will capture those for you.

The last step is simply to tell the menu to popup at the particular location. Here's an example of the whole process, using a popup menu on the application's main window.

```
menu .menu
foreach i [list One Two Three] {.menu add command -label $i}
if {[tk windowingsystem]=="aqua"} {
        bind . <2> "tk_popup .menu %X %Y"
        bind . <Control-1> "tk_popup .menu %X %Y"
} else {
        bind . <3> "tk_popup .menu %X %Y"
}
```

Windows and Dialogs

Everything we've done up until now has been in a single window. In this chapter, we'll cover how to use multiple windows, changing various attributes of windows, and use some of the standard dialog box windows that are available in Tk.

Creating and Destroying Windows

You've already seen that all Tk programs start out with a root toplevel window, and then widgets are created as children of that root window. Creating new toplevel windows works almost exactly the same as creating new widgets.

Toplevel windows are created using the **`tk::toplevel`** command:

```
tk::toplevel .t
```

Unlike regular widgets, you don't have to "grid" a toplevel for it to appear onscreen. Once you've created a new toplevel, you can then create other widgets which are children of that toplevel, and grid them inside the toplevel. In other words, the new toplevel behaves exactly like the automatically created root window.

To destroy a window, you can use the **`destroy`** command:

```
destroy .win1 ?.win2 ...?
```

> *Note that the `destroy` command lives in the global namespace; there is not a `tk::destroy` command.*

Note that you can use `destroy` on any widget, not just a toplevel window. Also, when you destroy a window, all windows (widgets) that are children of that window are also destroyed. So if you happen to destroy the root window (that all other widgets are descended from), that will normally end your application.

Changing Window Behavior and Styles

There are lots of things about how windows behave and how they look that can be changed.

Window Title

To examine or change the title of the window:

```
set oldtitle [wm title .window]
wm title .window "New title"
```

> *The "wm" here stands for "window manager" which is an X11 term used for a program that manages all the windows onscreen, including their title bars, frames, and so on. What we're effectively doing is asking the window manager to change the title of this particular window for us. The same terminology has been carried over to Tk running on Mac OS X and Windows.*

> *Note that the `wm` command lives in the global namespace; there is not a `tk::wm` command.*

Size and Location

In Tk, a window's position and size on the screen is known as its *geometry*. A full geometry specification looks like this:

```
width x height ± x ± y
```

Width and height (normally in pixels) are pretty self-explanatory. The "x" (horizontal position) is specified with a leading plus or minus, so `"+25"` means the left edge of the window should be 25 pixels from the left edge of the screen, while `"-50"` means the right edge of the window should be 50 pixels from the right edge of the screen. Similarly, a "y" (vertical) position of "+10" means the top edge of the window should be ten pixels below the top of the screen, while "-100" means the bottom edge of the window should be 100 pixels above the bottom of the screen.

> *Remember that the geometry's position are the actual coordinates on the screen, and don't make allowances for systems like Mac OS X which have a menubar along the top, or a dock area along the bottom. So specifying a position of "+0+0" would actually place the top part of the window under the system menu bar. It's a good idea to leave a healthy margin (at least 30 pixels) from an edge of the screen.*

Here is an example of changing the size and position, placing the window towards the top righthand corner of the screen:

```
wm geometry .window 300x200-5+40
```

Stacking Order

Stacking order refers to the order that windows are "placed" on the screen, from bottom to top. When the positions of two windows overlap each other, the one closer to the top of the stacking order will obscure or overlap the one lower in the stacking order.

You can obtain the current stacking order, a list from lowest to heighest, via:

```
wm stackorder .window
```

You can also just check if one window is above or below another:

```
if {[wm stackorder .window isabove .other]} {...}
if {[wm stackorder .window isbelow .other]} {...}
```

You can also raise or lower windows, either to the very top (bottom) of the stacking order, or just above (below) a designated window:

```
raise .window
raise .window .other
lower .window
lower .window .other
```

Wondering why you needed to pass a window to get the current stacking order? It turns out that stacking order, raise and lower, etc. work not only for toplevel windows, but with any "sibling" widgets (those having the same parent). So if you have several widgets gridded together but overlapping, you can raise and lower them relative to each other. For example:

```
grid [ttk::label .little -text "Little"] -column 0 -row 0
grid [ttk::label .bigger -text "Much Bigger Label"] -column 0 -row 0
after 2000 raise .little
```

> *The `"after"` command schedules a script to be executed at a certain number of milliseconds in the future, but allows normal processing of the event loop to continue in the meantime.*

Resizing Behavior

Normally, toplevel windows, both the root window and others that you create, can be resized by the user. However, sometimes in your user interface, you may want to prevent the user from resizing the window. You can prevent it from being resized, in fact independently specifying whether the window's width (first parameter) can be changed, as well as its height (second parameter). So to disable all resizing:

```
wm resizable .window 0 0
```

Remember that if you've added a `ttk::sizegrip` widget to the window, that you should remove it if you're making the window non-resizable.

If resizing is enabled, you can specify a minimum and/or maximum size that you'd like the window's size to be constrained to (again, parameters are width and height):

```
wm minsize .window 200 100
wm maxsize .window 500 500
```

Iconifying and Withdrawing

On most systems, you can temporarily remove the window from the screen by iconifying it. In Tk, whether or not a window is iconified is referred to as the window's *state*. The possible states for a window include `"normal"` and `"iconic"` (for an iconified window), as well as several others: `"withdrawn"`, `"icon"` or `"zoomed"`.

You can query or set the current window state, and there are also the methods `"iconify"` and `"deiconify"` which are shortcuts for setting the `"iconic"` or `"normal"` states respectively.

```
set thestate [wm state .window]
wm state .window normal
wm iconify .window
wm deiconify .window
```

Standard Dialogs

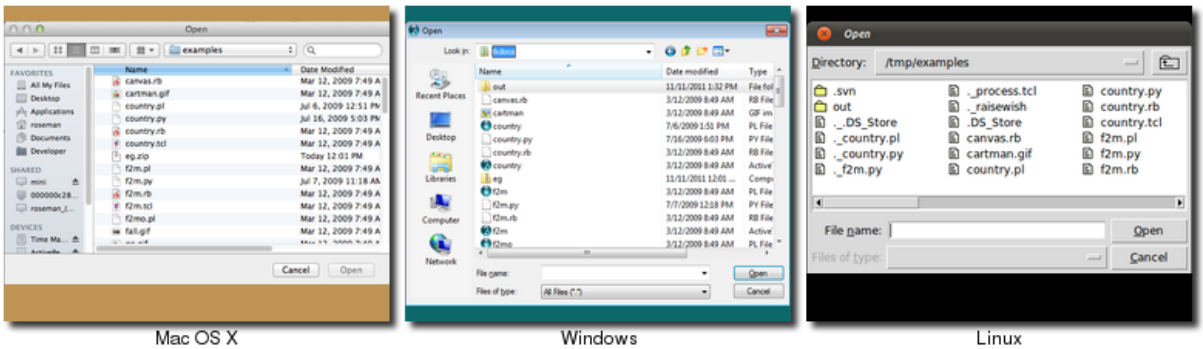
Dialog boxes are a type of window used in applications to get some information from the user, inform them that some event has occurred, confirm an action and more. The appearance and usage of dialog boxes is usually quite specifically detailed in a platform's style guide. Tk comes with a number of dialog boxes built-in for common tasks, and which help you conform to platform specific style guidelines.

Selecting Files and Directories

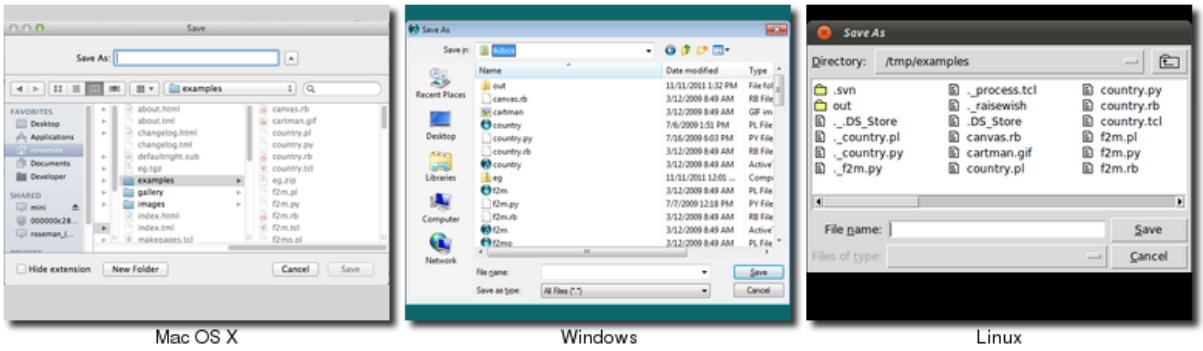
Tk provides several dialogs to let the user select files or directories. On Windows and Mac, these invoke the underlying operating system dialogs directly. The "open" variant on the dialog is used when you want the user to select an existing file (like in a "File | Open..." menu command), while the "save" variant is used to choose a file to save into (normally used by the "File | Save As..." menu command).

```
set filename [tk_getOpenFile]
set filename [tk_getSaveFile]
set dirname [tk_chooseDirectory]
```

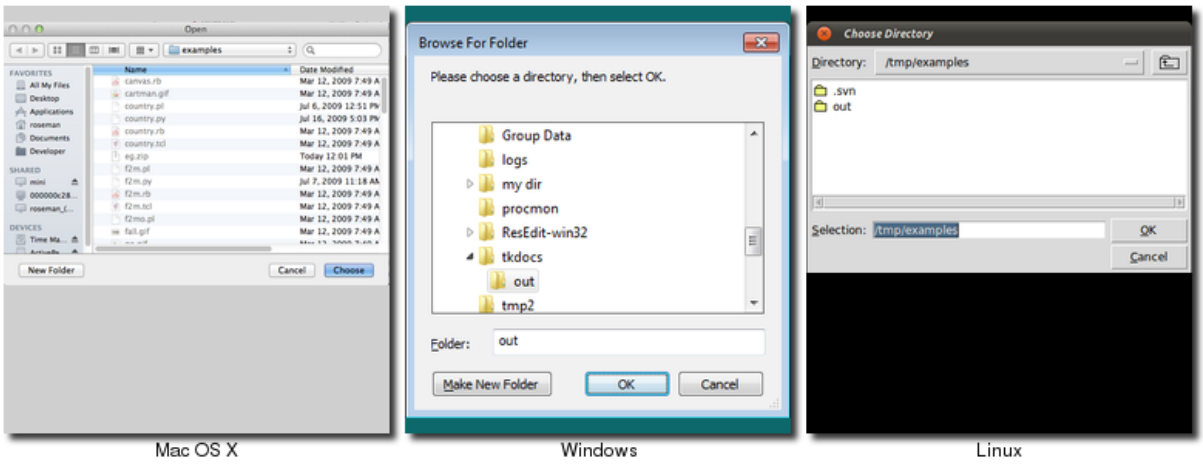
All of these commands produce *modal* dialogs, which means that the commands (and hence your program) will not continue running until the user submits the dialog. The commands return the full pathname of the file or directory the user has chosen, or return an empty string if the user cancels out of the dialog.



Open File Dialogs



Save File Dialogs



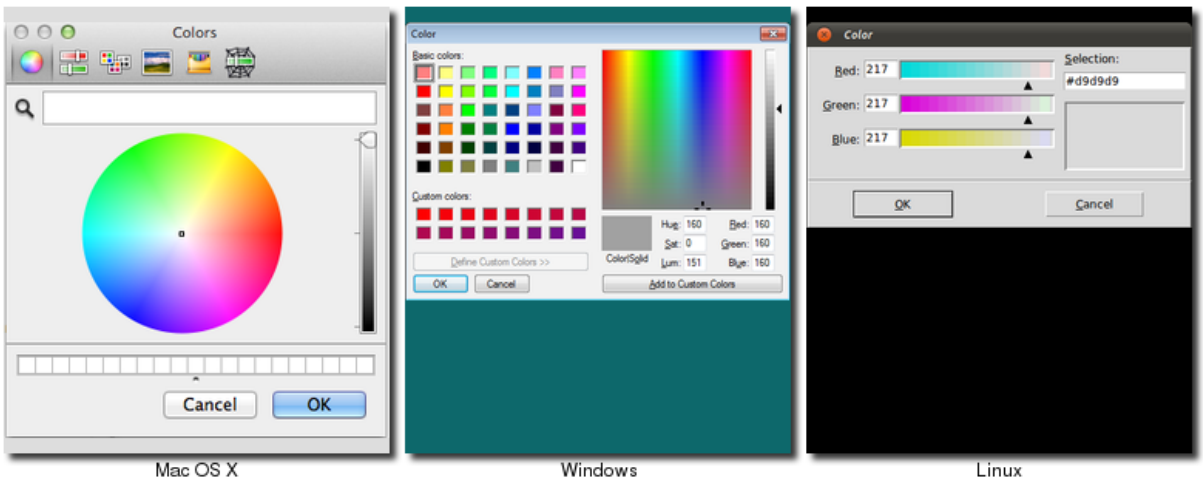
Choose Directory Dialogs

There are a variety of different options that can be passed to these dialogs, allowing you to set the allowable file types, default filename, and more. These are detailed in the getOpenFile/getSaveFile and chooseDirectory reference manual pages.

Selecting Colors

There is also a modal dialog to let the user select a color. It will return a color value, e.g. "#ff62b8". The dialog takes an optional `"initialcolor"` option to specify an existing color that the user is presumably replacing.

```
tk_chooseColor -initialcolor #ff0000
```



Choose Color Dialogs

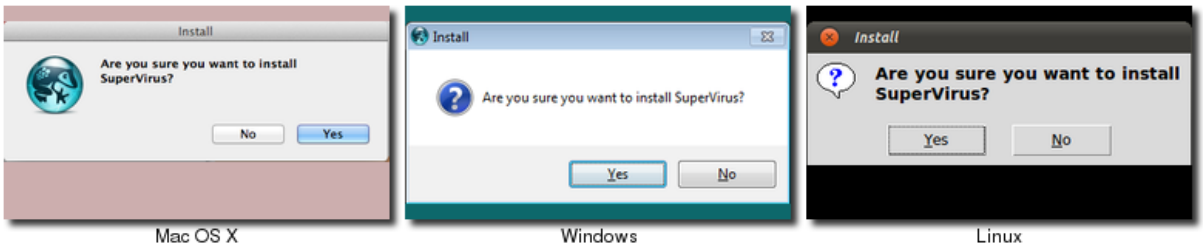
Alert and Confirmation Dialogs

Many applications use various simple modal alerts or dialogs to notify the user of an event, ask them to confirm an action, or make another similar choice which is done by clicking on a button. Tk provides a versatile "message box" that encapsulates all these different types of dialogs.

```
tk_messageBox -message "Have a good day"
```

Simple Message Boxes

```
tk_messageBox -type "yesno"
            -message "Are you sure you want to install SuperVirus?"
            -icon question -title "Install"
```



Example Message Boxes

Like the previous dialogs that we've seen, these are modal, and will return the result of the user's action to the caller. The exact return value will depend on the `"type"` option passed to the command, as shown here:

Type option	Possible return values
`ok` *(default)*	`"ok"`
`okcancel`	`"ok"` or `"cancel"`
`yesno`	`"yes"` or `"no"`
`yesnocancel`	`"yes"`, `"no"` or `"cancel"`
`retrycancel`	`"retry"` or `"cancel"`
`abortretryignore`	`"abort"`, `"retry"` or `"ignore"`

The full list of possible options is shown here:

`type`	As described above.
`message`	The main message displayed inside the alert.
`detail`	If needed, a secondary message, often displayed in a smaller font under the main message.
`title`	Title for the dialog window. Not used on Mac OS X.
`icon`	Icon to show: one of `"info"` (default), `"error"`, `"question"` or `"warning"`.
`default`	Specify which button (e.g. `"ok"` or `"cancel"` for a `"okcancel"` type dialog) should be the default.
`parent`	Specify a window of your application this dialog is being posted for; this may cause the dialog to appear on top, or on Mac OS X appear as a sheet for the window.

> *These new messagebox dialogs are a replacement from the older `"tk_dialog"` command, which does not comply with current platform user interface conventions.*

Organizing Complex Interfaces

If you have a large user interface, you'll need to find ways to organize it in ways that don't overwhelm your users with the complexity. There are a number of different approaches to doing this; again, both general and platform specific human interface guidelines are a good resource when deciding.

Note that when we're talking about complexity in this chapter, it's not the underlying technical complexity of how the

program is put together, but how it's presented to the user. A user interface can be pulled together from many different modules, be built up from multiple canvas widgets and deeply nested frames, but that doesn't necessarily mean the user perceives it to be complex.

Multiple windows

One of the benefits of using multiple windows in an application can be to simplify the user interface, by requiring the user to focus only on the contents of one window at a time (requiring them to focus on or switch between several windows can also have the opposite effect). Similarly, showing only the widgets that are relevant for the current task (via `grid`) can help simplify the user interface.

White space

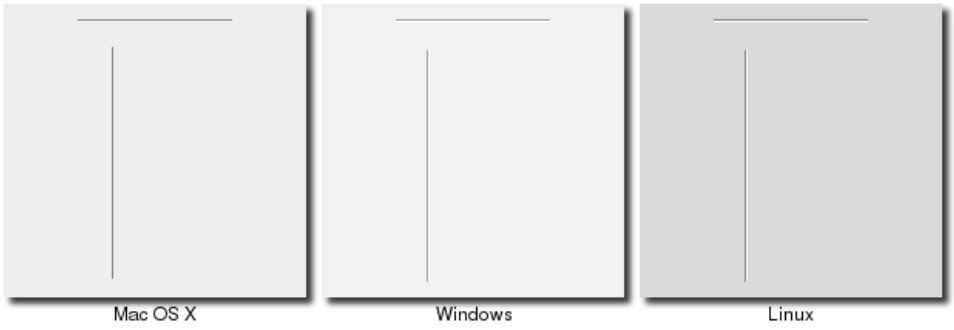
If you do need to display a large number of widgets onscreen at the same time, you have to think about how to organize them visually. You've seen how `grid` can help by making it easy to align widgets with each other. White space is another useful aid. Placing related widgets quite close to each other (possible with an explanatory label immediately above) and separated from other less-related widgets by white space helps the user organize the user interface in their own mind.

> *The amount of white space around different widgets, between groups of widgets, around borders and so on is highly platform specific. While you can do an adequate job without worrying about exact pixel numbers, if you want a highly polished user interface, you'll need to tune this for each platform.*

Separator

- **Widget Roundup**
- **Reference Manual**

A second approach to grouping widgets in one display is to place a thin horizontal or vertical rule between groups of widgets; often this can be more space efficient than using white space, which may be relevant for a tight display. Tk provides a very simple **separator** widget for this purpose.



Separator Widgets

Separators are created using the **`ttk::separator`** command:

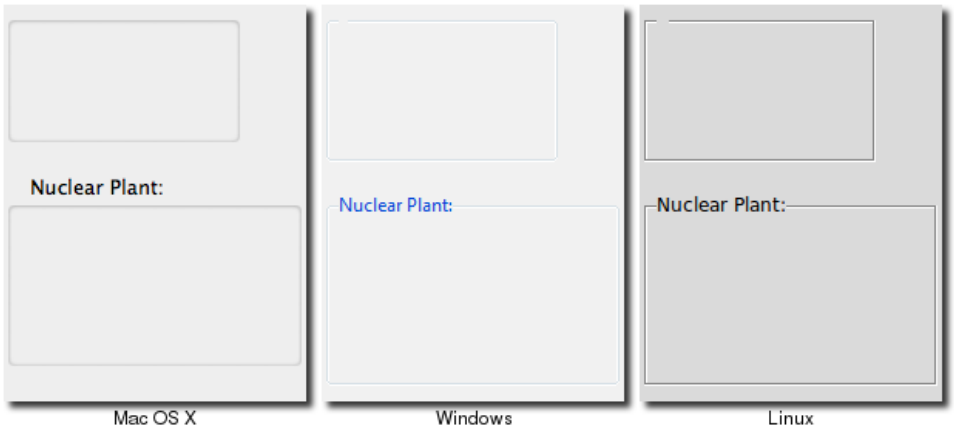
```
ttk::separator .s -orient horizontal
```

The `"orient"` option may be specified as either `"horizontal"` or `"vertical"`.

Label Frames

- **Widget Roundup**
- **Reference Manual**

A **labelframe** widget, also commonly known as a *group box*, provides another way to group related components. It acts like a normal `ttk::frame`, in that you will normally use it as a container for other widgets you `grid` inside it. However, it displays in a way that visually sets it off from the rest of the user interface. You can optionally provide a text label to be displayed outside the labelframe.



Labelframe Widgets

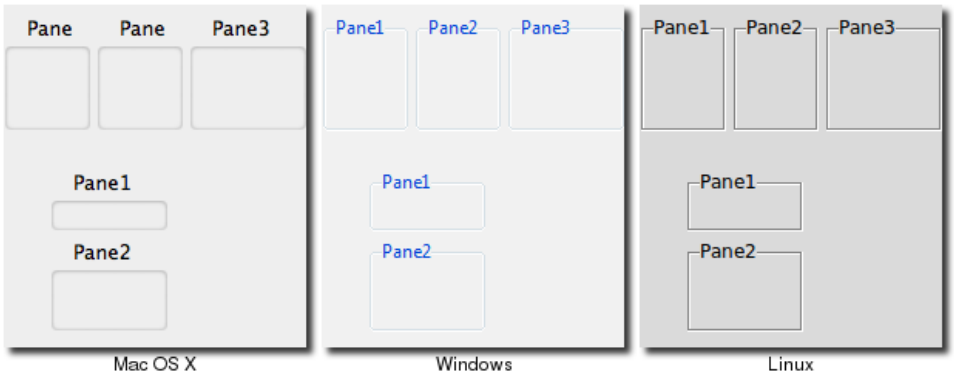
Labelframes are created using the **ttk::labelframe** command:

```
ttk::labelframe .lf -text "Label"
```

Paned Windows

- Widget Roundup
- Reference Manual

A **panedwindow** widget lets you stack two or more resizable widgets above and below each other (or to the left and right). The user can adjust the relative heights (or widths) of each pane by dragging a *sash* located between them. Typically the widgets you're adding to a panedwindow will be frames containing many other widgets.



Panedw indow Widgets (Show n here managing several labelframes)

Panedwindows are created using the **ttk::panedwindow** command:

```
ttk::panedwindow .p -orient vertical
# first pane, which would get widgets gridded into it:
ttk::labelframe .p.f1 -text Pane1 -width 100 -height 100
ttk::labelframe .p.f2 -text Pane2 -width 100 -height 100; # second pane
.p add .p.f1
.p add .p.f2
```

A panedwindow is either `"vertical"` (it's panes are stacked vertically on top of each other), or `"horizontal"`. Importantly, all of the panes that you add to the panedwindow must be a *direct child* of the panedwindow itself.

Calling the `"add"` method will add a new pane at the end of the list of panes. The `"insert` *`position subwindow`*`"` method allows you to place the pane at the given position in the list of panes (0..n-1); if the pane is already managed by the panedwindow, it will be moved to the new position. You can use the `"forget` *`subwindow`*`"` to remove a pane from the panedwindow; you can also pass a position instead of a subwindow.

Other options let you sign relative weights to each pane so that if the overall panedwindow resizes, certain panes will get more space than others. As well, you can adjust the position of each sash between items in the panedwindow. See the command reference for details.

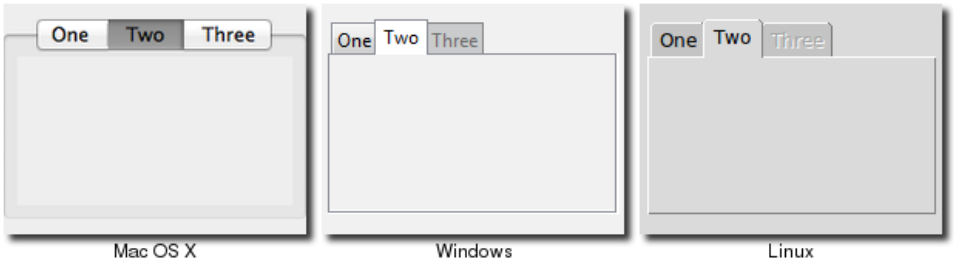
Notebook

- Widget Roundup

A **notebook** widget uses the metaphor of a tabbed notebook to let the user switch

between one of several *pages*, using an index *tab*. In this case, unlike with paned windows, we're only allowing the user to look at a single page (akin to a pane) at a time.

- Reference Manual



Notebook Widgets

Notebooks are created using the **ttk::notebook** command:

```
ttk::notebook .n
ttk::frame .n.f1; # first page, which would get widgets gridded into it
ttk::frame .n.f2; # second page
.n add .n.f1 -text "One"
.n add .n.f2 -text "Two"
```

The operations on tabbed notebooks are similar to those on panedwindows. Each page is typically a frame, again a direct child (subwindow) of the notebook itself. A new page and its associated tab are added to the end of the list of tabs with the "`add` *`subwindow ?option value...?`*" method. The "`text`" tab option is used to set the label on the tab; also useful is the "`state`" tab option, which can have the value "`normal`", "`disabled`" (not selectable), or "`hidden`".

To insert a tab at somewhere other than the end of the list, you can use the "`insert` *`position subwindow ?option value...?`*", and to remove a given tab, use the "`forget`" method, passing it either the position (0..n-1) or the tab's subwindow. You can retrieve the list of all subwindows contained in the notebook via the "`tabs`" method.

To retrieve the subwindow that is currently selected, call the "`select`" method, and change the selected tab by passing it either the tab's position or the subwindow itself as a parameter.

To change a tab option (like the text label on the tab or its state), you can use the "`tab` *`tabid option value`*" method (where "`tabid`" is again the tab's position or subwindow); omit the "`value`" to return the current value of the option.

Again, there are a variety of less frequently used options and commands detailed in the command reference .

Fonts, Colors, Images

This chapter describes how Tk handles fonts, colors and images. We've touched on all of these before, but here we'll provide a more in depth treatment.

Fonts

Several Tk widgets, such as the label, text, and canvas, allow you to specify the fonts used to display text, typically via a "`font`" configuration option. As with many things in Tk, the default fonts are usually a good choice, but if you do want to make changes, this section will describe several ways to do so. Fonts are one of several areas that are highly platform specific, so how you specify them is important.

The font command reference provides full details on specifying fonts, as well as other font-related operations.

Most of the themed widgets that display text don't have a "`font`" configuration option, unlike the classic Tk widgets. Rather than modifying individual widgets, the correct approach in the themed widgets is to specify the fonts used in a style, and then use that style for the individual widget. This is akin to the difference between hardcoding display-oriented markup like font tags inside HTML pages, vs. using CSS stylesheets that keep all the display specific information in one place.

> *Many older Tk programs hardcoded a lot of fonts, using either the "family size style" format we'll see below, X11 font names, or the older and more arcane X11 font specification string. In many cases, this left these applications with a dated look as platforms evolved. Further, many programs specified fonts on a per-widget basis, leaving the font decisions spread out through the program. Named fonts, and use of the standard fonts that Tk provides, are a far better solution. Reviewing and updating the usage of fonts is an easy and important change to make in any existing applications.*

Standard Fonts

Particularly for more-or-less standard user interface elements, each platform defines specific fonts that should be

used. Tk encapsulates many of these into a standard set of fonts that are always available, and of course the standard widgets use these fonts. This helps abstract away platform differences. The predefined fonts are:

`TkDefaultFont`	The default for all GUI items not otherwise specified.
`TkTextFont`	Used for entry widgets, listboxes, etc.
`TkFixedFont`	A standard fixed-width font.
`TkMenuFont`	The font used for menu items.
`TkHeadingFont`	The font typically used for column headings in lists and tables.
`TkCaptionFont`	A font for window and dialog caption bars.
`TkSmallCaptionFont`	A smaller caption font for subwindows or tool dialogs
`TkIconFont`	A font for icon captions.
`TkTooltipFont`	A font for tooltips.

Platform-Specific Fonts

A number of additional predefined fonts are available, but the precise set depends on the platform. Obviously, if you're using these, and your application is portable across different platforms, you'll need to ensure that proper fonts are defined individually for each platform.

On X11, any valid X11 font name (see e.g. the "xlsfonts" command) may be used. Remember though that there is no guarantee that a particular font has been installed on a particular machine.

On Windows, the following font names, which map to the fonts that can be set in the "Display" Control Panel, are available: `system`, `ansi`, `device`, `systemfixed`, `ansifixed`, `oemfixed`.

On Mac OS X, the following fonts are available (see the Apple HIG for details): `systemSystemFont`, `systemSmallSystemFont`, `systemApplicationFont`, `systemViewsFont`, `systemMenuItemFont`, `systemMenuItemCmdKeyFont`, `systemPushButtonFont`, `systemAlertHeaderFont`, `systemMiniSystemFont`, `systemDetailEmphasizedSystemFont`, `systemEmphasizedSystemFont`, `systemSmallEmphasizedSystemFont`, `systemLabelFont`, `systemMenuTitleFont`, `systemMenuItemMarkFont`, `systemWindowTitleFont`, `systemUtilityWindowTitleFont`, `systemToolbarFont`, `systemDetailSystemFont`.

Named Fonts

You can also create your own fonts, which can be used exactly like the predefined ones. To do so, you'll need to pick a name to refer to the font, and then specify various font attributes that define how the font should look. Typically, you'd use different font attributes on different platforms; that way, you can use the font in your program without worrying about the details except in the one place the font is actually defined.

Here's an example:

```
font create AppHighlightFont -family Helvetica -size 12 -weight bold
grid [ttk::label .l -text "Attention!" -font AppHighlightFont]
```

The `"family"` specifies the font name; the names `Courier`, `Times`, and `Helvetica` are guaranteed to be supported (and mapped to an appropriate monospaced, serif, or sans-serif font), but other fonts installed on the system can be used (again, be careful to ensure the font exists, or the system will supply a different font, which may not necessarily be a good match). You can get the names of all available fonts with:

```
font families
```

The `"size"` option specifies the size of the font, in points. The `"weight"` option can be either `bold` or `normal`. You can specify a `"slant"` of `roman` (normal) or `italic`. Finally, the boolean options `"underline"` and `"overstrike"` are available.

The current settings of these options can be examined or changed using the same mechanisms that you'd use for changing the configuration options of a widget (e.g. `configure`).

Font Descriptions

Another way to specify fonts is via a list of attributes, starting with the name of the font, and then optionally including a size, and optionally one or more style options. Some examples of this are `"Helvetica"`, "Helvetica 12", `"Helvetica 12 bold"`, and `"Helvetica 12 bold italic"`. These font descriptions are then used as the value of the `"font"` configuration option, rather than a predefined or named font.

> *In general, switching from font descriptions to named fonts is advisable, again to isolate font differences in one location in the program.*

Colors

As with fonts, there are various ways to specify colors. Full details can be found in the colors command reference .

In general, the system will provide the right colors for most things. Like with fonts, both Mac and Windows specify a

large number of system-specific color names (see the reference), whose actual color may depend upon system settings (e.g. text highlight colors, default backgrounds).

You can also specify colors via RGB, like in HTML, e.g. `"#3FF"` or `"#FF016A"`. Finally, Tk recognizes the set of color names defined by X11; normally these are not used, except for very common ones such as `"red"`, `"black"`, etc.

For themed Tk widgets, colors are often used in defining styles that are applied to widgets, rather than applying the color to a widget directly.

It probably goes without saying that restraint in the use of colors is normally warranted.

Images

We've seen the basics of how to use images already, displaying them in labels or buttons for example. We create an image object, usually from a file on disk.

```
image create photo imgobj -file "myimage.gif"
.label configure -image imgobj
```

Out of the box, Tk includes support for GIF and PPM/PNM images. However, there is a Tk extension library called "Img" which adds support for many others: BMP, XBM, XPM, PNG, JPEG, TIFF, etc. Though not included directly in the Tk core, Img is usually included with other packaged distributions (e.g. ActiveTcl).

```
package require Img
image create photo myimg -file "myimage.png"
```

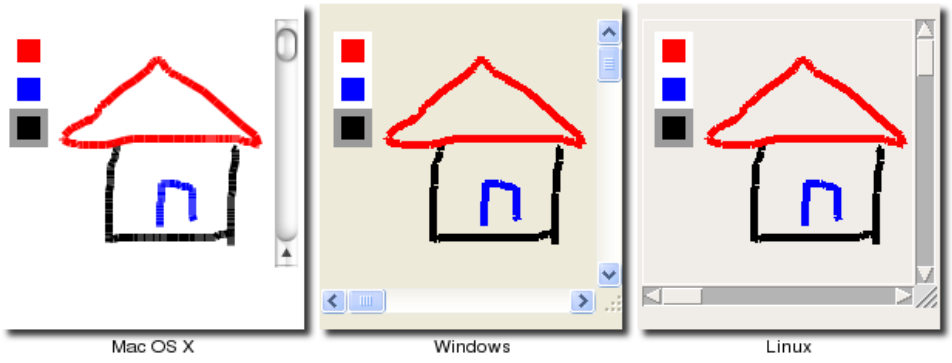
Tk's images are actually quite powerful and sophisticated, and provide a wide variety of ways to inspect and modify images. You can find out more from the image command reference and the photo command reference .

> *The types of multi-color images we've seen here are referred to in Tk as photo images. Tk also provides a second type of images, two-bit bitmap images, which were widely used in the 90's when most Unix workstations used quite large (compared with PCs) monitors, but they were only black and white. Needless to say, color is de rigueur these days, so updating to full color images for icons and so on is highly advisable.*

Canvas

- **Widget Roundup**
- **Reference Manual**

A **canvas** widget manages a 2D collection of graphical objects — lines, circles, images, other widgets and more. Tk's canvas is an incredibly powerful and flexible widget, and truly one of Tk's highlights. It is suitable for a wide range of uses, including drawing or diagramming, CAD tools, displaying or monitoring simulations or actual equipment, and for building more complex widgets out of simpler ones. Canvas widgets are part of the classic Tk widgets, not the themed Tk widgets.



Canvas Widgets

Canvas widgets are created using the **`tk::canvas`** command:

```
tk::canvas .canvas
```

Because canvas widgets have a huge amount of features, we won't be able to cover everything here. What we will do is take a fairly simple example (a freehand sketching tool) and incrementally add new pieces to it, each showing another new feature of canvas widgets. Towards the end of the chapter, we'll then cover some of the other major features not illustrated in the example.

Creating Items

When you create a new canvas widget, it will essentially be a large rectangle with nothing on it; truly a blank canvas in other words. To do anything useful with it, you'll need to add *items* to it. As mentioned, there are a wide variety of different types of items you can add. Here, we'll look at adding a simple line item to the canvas.

To create a line, the one piece of information you'll need to specify is where the line should be. This is done by using the coordinates of the starting and ending point, expressed as a list of the form *x0 y0 x1 y1*. The *origin* (0,0) is at the top left corner of the canvas, with the *x* value increasing as you move to the right, and the *y* value increasing as you move down. So to create a line from (10,10) to (200,50), we'd use this code:

```
.canvas create line 10 10 200 50
```

The "create line" command will return an item id (an integer) that can be used to uniquely refer to this item; every item created will get its own id. Though often we don't need to refer to the item later and will therefore ignore the returned id, we'll see how it can be used shortly.

Let's start our simple sketchpad example. For now, we'll want to be able to draw freehand on the canvas by dragging the mouse on it. We'll create a canvas widget, and then attach event bindings to it to capture mouse clicks and drags. When we first click the mouse, we'll remember that location as our "start" position. Every time the mouse is moved with the mouse button still held down, we'll create a line item going from this "start" position to the current mouse position. The current position will then be the "start" position for the next line segment.

```
package require Tk

grid [tk::canvas .canvas] -sticky nwes -column 0 -row 0
grid columnconfigure . 0 -weight 1
grid rowconfigure . 0 -weight 1

bind .canvas <1> "set lastx %x; set lasty %y"
bind .canvas <B1-Motion> "addLine %x %y"

proc addLine {x y} {
    .canvas create line $::lastx $::lasty $x $y
    set ::lastx $x; set ::lasty $y
}
```

Try it out - drag the mouse around the canvas to create your masterpiece.

Item Attributes

When creating items, you can also specify one or more *attributes* for the item, that will affect how it is displayed. For example, here we'll specify that the line should be red, and three pixels wide.

```
.canvas create line 10 10 200 50 -fill red -width 3
```

The exact set of attributes will vary according to the type of item.

Like with Tk widgets, changing attributes for canvas items after you've already created them can also be done.

```
set id [.canvas create line 0 0 10 10 -fill red]
...
.canvas itemconfigure $id -fill blue -width 2
```

Bindings

We've already seen that the canvas widget as a whole, like any other Tk widget, can capture events using the "bind" command.

You can also attach bindings to individual items in the canvas (or groups of them, as we'll see in the next section using tags). So if you want to know whether or not a particular item has been clicked on, you don't need to watch for mouse click events for the canvas as a whole, and then figure out if that click happened on your item. Tk will take care of all this for you.

To capture these events, you use a bind command built into the canvas. It works exactly like the regular bind command, taking an event pattern and a callback. The only difference is you specify the canvas item this binding applies to.

```
.canvas bind $id <1> {...}
```

Let's add some code to our sketchpad example to allow changing the drawing color. We'll first create a few different rectangle items, each filled with a different color. Creating rectangle items is just like creating line items, where you'll specify the coordinates of two diagonally opposite corners. We'll then attach a binding to each of these so that when they're clicked on, they'll set a global variable to the color to use. Our mouse motion binding will look at that variable when creating the line segments.

```
set id [.canvas create rectangle 10 10 30 30 -fill red]
.canvas bind $id <1> "setColor red"
```

```
set id [.canvas create rectangle 10 35 30 55 -fill blue]
.canvas bind $id <1> "setColor blue"

set id [.canvas create rectangle 10 60 30 80 -fill black]
.canvas bind $id <1> "setColor black"

set ::color black

proc setColor {color} {
    set ::color $color
}
proc addLine {x y} {
    .canvas create line $::lastx $::lasty $x $y -fill $::color
    set ::lastx $x; set ::lasty $y
}
```

Tags

We've seen that every canvas item has a unique id number, but there is another very useful and powerful way to refer to items on a canvas, and that is using *tags*.

A tag is just an identifier of your creation, something meaningful to your program. You can attach tags to canvas items; each item can have any number of tags. Unlike item id numbers, which are unique for each item, many items can have the same tag.

What can you do with tags? We saw that you can use the item id to modify a canvas item (and we'll see soon there are other things you can do to items, like move them around, delete them, etc.). Any time you can use an item id, you can use a tag. So for example, you can change the color of all items having a specific tag.

Tags are a good way to identify certain types of items in your canvas (items that are part of a drawn line, items that are part of the pallette, etc.). You can use tags to correlate canvas items to particular objects in your application (so for example, tag all canvas items that are part of the robot with id #37 with the tag "robot37"). With tags, you don't have to keep track of the ids of canvas items to refer to groups of items later; tags let Tk do that for you.

You can assign tags when creating an item using the `"tags"` item configuration option. You can add tags later with the `"addtag"` method, or remove them with the `"dtags"` method. You can get the list of tags for an item with the `"gettags"` method, or return a list of item id numbers having the given tag with the `"find"` command.

For example:

```
% canvas .c
.c
% .c create line 10 10 20 20 -tags "firstline drawing"
1
% .c create rectangle 30 30 40 40 -tags "drawing"
2
% .c addtag rectangle withtag 2
% .c addtag polygon withtag rectangle
% .c gettags 2
drawing rectangle polygon
% .c dtag 2 polygon
% .c gettags 2
drawing rectangle
% .c find withtag drawing
1 2
```

As you can see, things like `"withtag"` will take either an individual item or a tag; in the latter case, they will apply to all items having that tag (which could be none). The `"addtag"` and `"find"` have many other options, allowing you to specify items near a point, overlapping a particular area, and more.

Let's use tags first to put a border around whichever item in our color palette is currently selected.

```
set id [.canvas create rectangle 10 10 30 30 -fill red -tags "palette palettered"]
set id [.canvas create rectangle 10 35 30 55 -fill blue -tags "palette paletteblue"]
set id [.canvas create rectangle 10 60 30 80 -fill black -tags "palette paletteblack paletteSelected"]

proc setColor {color} {
    set ::color $color
    .canvas dtag all paletteSelected
    .canvas itemconfigure palette -outline white
    .canvas addtag paletteSelected withtag palette$color
    .canvas itemconfigure paletteSelected -outline #999999
}

setColor black
.canvas itemconfigure palette -width 5
```

Let's also use tags to make the current stroke we're drawing appear more visible; when we release the mouse we'll put it back to normal.

```
bind .canvas <B1-ButtonRelease> "doneStroke"

proc addLine {x y} {
    .canvas create line $::lastx $::lasty $x $y -fill $::color -width 5 -tags currentline
    set ::lastx $x; set ::lasty $y
}
proc doneStroke {} {
    .canvas itemconfigure currentline -width 1
}
```

Modifying Items

You've seen how you can modify the configuration options on an item — its color, width and so on. There are a number of other things you can do items.

To delete items, use the `"delete"` method. To change an item's size and position, you can use the `"coords"` method; this allows you to provide new coordinates for the item, specified the same way as when you first created the item. Calling this method without a new set of coordinates will return the current coordinates of the item. To move one or more items by a particular horizontal or vertical amount from their current location, you can use the `"move"` method.

All items are ordered from top to bottom in what's called the stacking order. If an item later in the stacking order overlaps the coordinates of an item below it, the item on top will be drawn on top of the lower item. The `"raise"` and `"lower"` methods allow you to adjust an item's position in the stacking order.

There are several more operations described in the reference manual page, both to modify items and to retrieve additional information about them.

Scrolling

In many applications, you'll want the canvas to be larger than what appears on the screen. You can attach horizontal and vertical scrollbars to the canvas in the usual way, via the `"xview"` and `"yview"` methods.

As far as the size of the canvas, you can specify both how large you'd like it to be on screen, as well as what the full size of the canvas is, which would require scrolling to see. The `"width"` and `"height"` configuration options for the canvas widget will request the given amount of space from the geometry manager. The `"scrollregion"` configuration option (e.g. `"0 0 1000 1000"`) tells Tk how large the canvas surface is.

You should be able to modify the sketchpad program to add scrolling, given what you already know. Give it a try.

Once you've done that, scroll the canvas down just a little bit, and then try drawing. You'll see that the line you're drawing appears *above* where the mouse is pointing! Surprised?

What's going on is that the global `"bind"` command doesn't know that the canvas is scrolled (it doesn't know the details of any particular widget). So if you've scrolled the canvas down by 50 pixels, and you click on the top left corner, bind will report that you've clicked at (0,0). But we know that because of the scrolling, that position should really be (0,50).

The `"canvasx"` and `"canvasy"` methods will translate the position onscreen (which bind is reporting) into the actual point on the canvas, taking into account scrolling. If you're adding these directly to the event bindings (as opposed to procedures called from the event bindings), be careful about quoting and substitutions, to make sure that the conversions are done when the event fires.

Here then is our complete example. We probably don't want the palette to be scrolled away when the canvas is scrolled, but we'll leave that for another day.

```
package require Tk

grid [tk::canvas .canvas -scrollregion "0 0 1000 1000" -yscrollcommand ".v set" -xscrollcommand ".h set"] -sticky nwes -column 0 -r
grid columnconfigure . 0 -weight 1
grid rowconfigure . 0 -weight 1

grid [tk::scrollbar .h -orient horizontal -command ".canvas xview"] -column 0 -row 1 -sticky we
grid [tk::scrollbar .v -orient vertical -command ".canvas yview"] -column 1 -row 0 -sticky ns
grid [ttk::sizegrip .sz] -column 1 -row 1 -sticky se

bind .canvas <1> {set lastx [.canvas canvasx %x]; set lasty [.canvas canvasy %y]}
bind .canvas <B1-Motion> {addLine [.canvas canvasx %x] [.canvas canvasy %y]}
bind .canvas <B1-ButtonRelease> "doneStroke"


set id [.canvas create rectangle 10 10 30 30 -fill red -tags "palette palettered"]
.canvas bind $id <1> "setColor red"

set id [.canvas create rectangle 10 35 30 55 -fill blue -tags "palette paletteblue"]
.canvas bind $id <1> "setColor blue"
```

```
set id [.canvas create rectangle 10 60 30 80 -fill black -tags "palette paletteblack paletteSelected"]
.canvas bind $id <1> "setColor black"

proc setColor {color} {
    set ::color $color
    .canvas dtag all paletteSelected
    .canvas itemconfigure palette -outline white
    .canvas addtag paletteSelected withtag palette$color
    .canvas itemconfigure paletteSelected -outline #999999
}
proc addLine {x y} {
    .canvas create line $::lastx $::lasty $x $y -fill $::color -width 5 -tags currentline
    set ::lastx $x; set ::lasty $y
}
proc doneStroke {} {
    .canvas itemconfigure currentline -width 1
}

setColor black
.canvas itemconfigure palette -width 5
```

Other Item Types

Besides lines and rectangles, there are a number of different types of items that canvas widgets support. Remember that each one has its own set of item configuration options, detailed in the reference manual.

Items of type `"line"` can actually be a bit fancier than what we've seen. A line item can actually be a series of line segments, not just one; in our example, we could have chosen to use a single line item for each complete stroke. The line can also be drawn directly point-to-point, or smoothed out into a curved line.

Items of type `"rectangle"` we've seen. Items of type `"oval"` work the same but draw as an oval. Items of type `"arc"` allow you to draw just a piece of an oval. Items of type `"polygon"` allow you to draw a closed polygon with any number of sides.

Pictures can be added to canvas widgets, using items of type `"bitmap"` (for black and white), or type `"image"` (for full color).

You can add text to a canvas using items of type `"text"`. You have complete control of the font, size, color and more, as well as the actual text that is displayed.

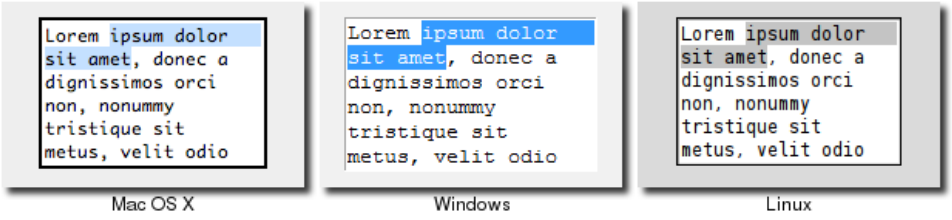
Perhaps most interestingly, you can embed other widgets (which would include a frame which itself contains other widgets) into a canvas using an item of type `"window"`. When we do this, the canvas in effect acts as a geometry manager for those other widgets. This capability raises all kinds of possibilities for your application.

There is a lot more to canvas widgets than we've described here; be sure to consult the reference manual, as well as the wide range of example programs included with the Tk distribution.

Text

- Widget Roundup
- Reference Manual

A **text** widget manages a multi-line text area. Like the canvas widget, Tk's text widget is an immensely flexible and powerful tool which can be used for a wide variety of tasks. Some example uses of text widgets have included anything from providing a simple multi-line text area as part of a form, to a stylized code editor, to an outliner, to a web browser. Text widgets are part of the classic Tk widgets, not the themed Tk widgets.



Text Widgets

Text widgets are created using the **`tk::text`** command:

```
tk::text .text -width 40 -height 10
```

While we briefly introduced text widgets in an earlier chapter, here we'll go into much more detail, to give you a sense of the level of sophistication it allows. Still, for any significant work with the text widget, the reference manual is very well organized and useful.

The Basics

If you just want to use the text widget to get a simple multi-line text from the user as part of a form, there's only a few things you'll need to worry about: creating and sizing the widget (check), providing an initial value for the text in the form, and retrieving the text in the widget after the user has submitted the form.

Providing Initial Content

When you first create it, the text widget has nothing in it, so if you want to provide an initial piece of text, you're going to have add it yourself. Unlike for example the entry widget, text widgets don't support a `"textvariable"` configuration option; as we'll soon see, text widgets can contain a lot more than just plain text, so a simple variable isn't sufficient to hold it all.

Instead, to set the initial text for the widget, you'll use the widget's `"insert"` method:

```
.text insert 1.0 "here is my text to insert"
```

The "1.0" here represents where to insert the text, and can be read as "line 1, character 0". This refers to the first character of the first line; for historical conventions related to how programmers normally refer to lines and characters, line numbers are 1-based, and character numbers are 0-based.

The text to insert is just a string. Because the widget can hold multi-line text, the string you supply can be multi-line as well. To do, simply embed `"\n"` (newline) characters in your string at the appropriate locations.

Scrolling

Scrollbars, both horizontal and vertical, can be attached to the text widget. This works exactly the same as using scrollbars in any other widget, such as a listbox or canvas.

You can also ask the widget to ensure that a certain part of the text is visible. For example, if you've added more text to the widget than will fit onscreen (so it will scroll) but want to ensure that the top of the text rather than the bottom is visible, call the `"see"` method, passing it the position of the text (e.g. "1.0").

Controlling Wrapping

What if some lines of text in the widget are very long, longer than the width of the widget? By default, text will just wrap around to the next line, but if and how it does this can be controlled by the `"wrap"` configuration option. The default value is `"char"`, meaning wrap around right at the character at the end of the line; other options are `"word"` to cause wrapping, but only at word breaks (e.g. spaces), and `"none"` meaning don't wrap around at all. In the latter case, some of the text won't be visible unless you attach a horizontal scrollbar to the widget.

Disabling the Widget

Some forms will temporarily disable editing in particular widgets unless certain conditions are met (e.g. some other options are set to a certain value). To prevent the user from making any changes to a text widget, set the `"state"` configuration option to `"disabled"`; re-enable editing by setting this option back to `"normal"`.

Retrieving the Text

Finally, after the user has made any changes and submitted the form, your program will want to retrieve the contents of the widget, which is done with the `"get"` method:

```
set thetext [.text get 1.0 end]
```

Modifying the Text in Code

While the user can modify the text in the text widget interactively, your program can also make changes. Adding text is done with the `"insert"` method, which we used above to provide an initial value for the text widget.

Text Positions and Indices

When we specified a position of "1.0" (first line, first character), this was an example of an *index*. It tells the insert method where to insert the new text (just before the first line, first character, i.e. at the very start of the widget). There are a variety of ways to specify these indices. You've also seen another one: the `"end"` (from the "get" example) means just past the end of the text ("just past" because text insertions go right before the given index, so inserting at "end" will add text to the end of the widget). Note that Tk will *always* add a newline at the very end of the text widget.

Here are a few additional examples of indices, and what they mean:

`3.end`	The newline at the end of line 3.
`1.0 + 3 chars`	Three characters past the start of line 1.
`2.end -1 chars`	The last character before the new line in line 2.
`end -1 chars`	The newline that Tk always adds at the end of the text.
`end -2 chars`	The actual last character of the text.

`end -1 lines`	The start of the last actual line of text.
`2.2 + 2 lines`	The third character (index 2) of the fourth line of text.
`2.5 linestart`	The first character of line 2.
`2.5 lineend`	The position of the newline at the end of line 2.
`2.5 wordstart`	The first character of the word containing the character at index 2.5.
`2.5 wordend`	The first character just past the last character of the word containing index 2.5.

Some additional things to keep in mind:

- The term "chars" can be abbreviated as "c", and "lines" as "l".
- You can omit the spaces between the terms, e.g. "1.0+3c".
- If you specify an index past the end of the widget (e.g. "end + 100c") it will be interpreted as the end.
- Adding characters will wrap to the next lines as needed; e.g. "1.0 + 10 chars" on a line with only five characters will end up being on the second line.
- When using indices containing multiple words, make sure to quote them appropriately so that Tk sees the entire index as a single argument.
- When moving up or down a certain number of lines, this is interpreted as *logical* lines, where each line is terminated only by the "\n". With long lines and wrapping enabled, this may be represent multiple lines on the display. If you'd like to move up or down a single line on the display, you can specify this as e.g. "1.0 + 2 display lines".
- To determine the actual canonical position of an index, use the `"index"` method, passing it the index expression, and it will return the corresponding index in the form "*`line.char`*".
- You can compare two indices using the `"compare"` method, which lets you check for equality, whether one index is later in the text than the other, etc.

Deleting Text

While the `"insert"` method adds new text anywhere in the widget, the `"delete"` method removes it. You can specify either a single character to be deleted (by index), or a range of characters specified by the start and end index. In the latter case, characters from (and including) the start index through to *just before* the end index will be deleted (so the character at the end index is not deleted). So this would remove a single line of text (including its trailing newline) from the start of the text:

```
.text delete 1.0 2.0
```

There is also a `"replace"` method, taking a starting index, and ending index and a string as parameters. It does the same as a delete, followed by an insert at the same location.

Example: Logging Window

Here's a short example illustrating how to use a text widget as a 80x24 logging window for your application. The user doesn't edit the text widget at all; instead, your program will write log messages to it. You'd like to keep the content to no more than 24 lines (so no scrolling), so as you add new messages at the end, you'll have to remove old ones from the top if there are already 24 lines.

```
package require Tk
grid [text .log -state disabled -width 80 -height 24 -wrap none]

proc writeToLog {msg} {
    set numlines [lindex [split [.log index "end - 1 line"] "."] 0]
    .log configure -state normal
    if {$numlines==24} {.log delete 1.0 2.0}
    if {[.log index "end-1c"]!="1.0"} {.log insert end "\n"}
    .log insert end "$msg"
    .log configure -state disabled
}
```

> *Note that because the widget was disabled, we had to reenable it to make any changes, even from our program.*

Formatting with Tags

Up until now, we've just dealt with plain text. Now it's time to look at how to add special formatting, such as bold, italic, strikethrough, background colors, font sizes, and much more. Tk's text widget implements these using a feature called *tags*.

Tags are objects associated with the text widget. Each tag is referred to via a name chosen by the programmer. Each tag can have a number of different configuration options; these are things like fonts, colors, etc. that will be used to format text. Though tags are objects having state, they don't need to be explicitly created; they'll be automatically created the first time the tag name is used.

Adding Tags to Text

Tags can be associated with one or more ranges of text in the widget. As before, these are specified via indices; a single index to represent a single character, and a start and end index to represent the range from the start character to just before the end character. Tags can be added to ranges of text using the `"tag add"` method, e.g.

```
.text tag add highlightline 5.0 6.0
```

Tags can also be provided when inserting text by adding an optional parameter to the `"insert"` method, which holds a list of one or more tags to add to the text you're inserting, e.g.

```
.text insert end "new material to insert" "highlightline recent warning"
```

As the text in the widget is modified, whether by the user or your program, the tags will adjust automatically. So for example if you had tagged the text "the quick brown fox" with the tag "nounphrase", and then replaced the word "quick" with "speedy", the tag would still apply to the entire phrase.

Applying Formatting to Tags

Formatting is applied to tags via configuration options; these work similarly to configuration options for the entire widget. As an example:

```
.text tag configure highlightline -background yellow -font "helvetica 14 bold" -relief raised
```

The currently available configuration options for tags are: `"background"`, `"bgstipple"`, `"borderwidth"`, `"elide"`, `"fgstipple"`, `"font"`, `"foreground"`, `"justify"`, `"lmargin1"`, `"lmargin2"`, `"offset"`, `"overstrike"`, `"relief"`, `"rmargin"`, `"spacing1"`, `"spacing2"`, `"spacing3"`, `"tabs"`, `"tabstyle"`, `"underline"`, and `"wrap"`. Check the reference manual for detailed descriptions of these. The `"tag cget"` method allows you to query the configuration options of a tag.

Because multiple tags can apply to the same range of text, there is the possibility for conflict (e.g. two tags specifying different fonts). A priority order is used to resolve these; the most recently created tags have the highest priority, but priorities can be rearranged using the `"tag raise"` and `"tag lower"` methods.

More Tag Manipulations

To delete a tag altogether, you can use the `"tag delete"` method. This also of course removes any references to the tag in the text. You can also remove a tag from a range of text using the `"tag remove"` method; even if that leaves no ranges of text with that tag, the tag object itself still exists.

The `"tag ranges"` method will return a list of ranges in the text that the tag has been applied to. There are also `"tag nextrange"` and `"tag prevrange"` methods to search forward or backward for the first such range from a given position.

The `"tag names"` method, called with no additional parameters, will return a list of all tags currently defined in the text widget (including those that may not be presently used). If you pass the method an index, it will return the list of tags applied to just the character at the index.

Finally, you can use the first and last characters in the text having a given tag as indices, the same way you can use "end" or "2.5". To do so, just specify `"`*`tagname`*`.first"` or `"`*`tagname`*`.last"`.

Differences between Tags in Canvas and Text Widgets

While both canvas and text widgets support "tags" which can be used to apply to several objects, style them, and so on, these tags are not the same thing. There are important differences to take note of.

In canvas widgets, individual canvas items have configuration options that control their appearance. When you refer to a tag in a canvas, the meaning of that is identical to "all canvas items presently having that tag". The tag itself doesn't exist as a separate object. So in the following snippet, the last rectangle added will *not* be colored red.

```
.canvas itemconfigure important -fill red
.canvas create rectangle 10 10 40 40 -tags important
```

In text widgets by contrast, it's not the individual characters that retain the state information about appearance, but tags, which are objects in their own right. So in this snippet, the newly added text *will* be colored red.

```
.text insert end "first text" important
.text tag configure important -foreground red
.text insert end "second text" important
```

Events and Bindings

One quite cool thing is that you can define event bindings on tags. That allows you to do things like easily recognize mouse clicks just on particular ranges of text, and popup up a menu or dialog in response. Different tags can have different bindings, so it saves you the hassle of sorting out questions like "what does a click at this location mean?". Bindings on tags are implemented using the `"tag bind"` method:

```
.text tag bind important <1> "popupImportantMenu"
```

Widget-wide binding to events works as it does for every other widget. Besides the normal low-level events, there are

also two virtual events that will be generated: <Modified> whenever a change is made to the content of the widget, and <Selection> whenever there is a change made to which text is selected.

Selecting Text

Your program may want to know if a range of text has been selected by the user, and if so, what that range is. For example, you may have a toolbar button to bold the selected text in an editor. While you can tell when the selection has changed (e.g. to update whether or not the bold button is active) via the <Selection> virtual event, that doesn't tell you what has been selected.

The text widget automatically maintains a tag named "sel", which refers to the selected text. Whenever the selection changes, the "sel" tag will be updated. So you can find the range of text selected using the `"tag ranges"` method, passing it "sel" as the tag to report on.

Similarly, you can change the selection by using `"tag add"` to set a new selection, or `"tag remove"` to remove the selection. You can't actually delete the "sel" tag of course.

> *Though the default widget bindings prevent this from happening, "sel" is like any other tag in that it can support multiple ranges, i.e. disjoint selections. To prevent this from happening when changing the selection from your code, make sure you remove any old selection before adding a new one.*

The text widget manages the concept of the insertion cursor (where newly typed text will appear) separate from the selection. It does so using a new concept called a *mark*.

Marks

Marks are used to indicate a particular place in the text. In that respect they are like indices, except that as the text is modified, the mark will adjust to be in the same relative location. In that way they are resemble tags, but refer to a single position rather than a range of text. Marks actually don't refer to a position occupied by a character in the text, but specify a position *between* two characters.

Tk automatically maintains two different marks. The first is named "insert", and is the present location of the insertion cursor. As the cursor is moved (via mouse or keyboard), the mark moves with it. The second mark is named "current", and reflects the position of the character underneath the current mouse position.

To create your own marks, use the widget's `"mark set"` method, passing it the name of the mark, and an index (the mark is positioned just before the character at the given index). This is also used to move an existing mark to a different position. Marks can be removed using the `"mark unset"` method, passing it the name of the mark. If you delete a range of text containing a mark, that also removes the mark.

The name of a mark can also be used as an index (in the same way "1.0" or "end-1c" are indices). You can find the next mark (or previous one) from a given index in the text using the `"mark next"` or `"mark previous"` methods. The `"mark names"` method will return a list of the names of all marks.

Marks also have a *gravity*, which can be modified with the `"mark gravity"` method, which affects what happens when text is inserted at the mark. Suppose we have the text "ac", with a mark in between that we'll symbolize with a pipe, i.e. "a|c". If the gravity of that mark is "right" (the default) the mark will attach itself to the "c". If the new text "b" is inserted at the mark, the mark will remain stuck to the "c", and so the new text will be inserted before the mark, i.e. "ab|c". If the gravity is instead "left", the mark will attach itself to the "a", and so new text will be inserted after the mark, i.e. "a|bc".

Images and Widgets

Like canvas widgets, text widgets can contain not only text, but also images and any other Tk widgets (including a frame itself containing many other widgets). In some senses, this allows the text widget to work as a geometry manager in its own right. The ability to add images and widgets within the text opens up a world of possibilities for your program.

Images are added to a text widget at a particular index, with the image normally specified as an existing Tk image. There are also other options that allow you to fine-tune padding and so on.

```
image create photo flowers -file flowers.gif
.text image create sel.first -image flowers
```

Widgets are added to a text widget pretty much the same way as images. The widget you're adding should be a descendant of the text widget in the overall window hierarchy.

```
ttk::button .text.b -text "Push Me"
.text window create 1.0 -window .text.b
```

Even More

There are many more things that the text widget can do; here we'll briefly mention just a few more of them. For details on using any of these, see the reference manual.

Search

The text widget includes a powerful `"search"` method which allows you to locate a piece of text within the widget; this is useful for a "Find" dialog, as one obvious example. You can search backwards or forwards from a particular position or within a given range, specify your search using exact text, case insensitive, or using regular expressions, find one or all occurrences of your search term, and much more.

Modifications, Undo and Redo

The text widget keeps track of whether or not changes have been made to the text (useful to know whether you need to save it for example), which you can query (or change) using the `"edit modified"` method. There is also a complete multi-level undo/redo mechanism, managed automatically by the widget when you set its `"undo"` configuration option to true. Calling `"edit undo"` or `"edit redo"` then will modify the current text using information stored on the undo/redo stack.

Eliding Text

You can actually include text in the widget that is not displayed; this is known as "elided" text, and is made available using the `"elide"` configuration option for tags. You can use this to implement for example an outliner, a "folding" code editor, or even just to bury some extra meta-data intermixed with your text. When specifying positioning with elided text you have to be a bit more careful, and so commands that deal with positions have extra options to either include or ignore the elided text.

Introspection

Like most Tk widgets, the text widget goes out of its way to expose information about its internal state; we've seen most of this in terms of the `"get"` method, widget configuration options, `"names"` and `"cget"` for both tags and marks, and so on. There is even more information available that you can use for a wide variety of tasks. Check out the `"debug"`, `"dlineinfo"`, `"bbox"`, `"count"` and `"dump"` methods in the reference manual.

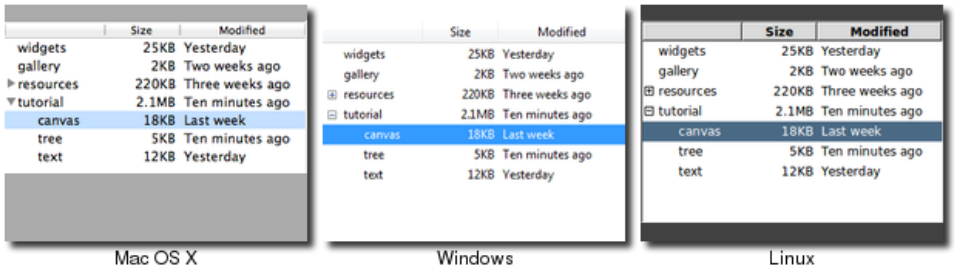
Peering

The Tk text widget allows the same underlying text data (containing all the text, marks, tags, images, and so on) to be shared between two or more different text widgets. This is known as *peering*, and is controlled via the `"peer"` method.

Tree

- Widget Roundup
- Reference Manual

A **treeview** widget can display and allow browsing through a hierarchy of items, and can show one or more attributes of each item as columns to the right of the tree. It allows you to build user interfaces similar to the tree display you'd find in file managers like the OS X Finder or Windows Explorer. As with most Tk widgets, there is a ton of flexibility to make it behave as you need for a wide range of situations.



Treeview Widgets

Treeview widgets are created using the **`ttk::treeview`** command:

```
ttk::treeview .tree
```

Horizontal and vertical scrollbars can be added in the usual manner, if desired.

Adding Items to the Tree

To do anything useful with the treeview, you'll need to add one or more *items* to it. Each item represents a single node in the tree, whether a leaf node or an internal node. Items are referred to by a unique id; this can be assigned by the

programmer when the item is first created, or the widget can automatically choose an id for the item.

Items are created by inserting them into the tree, using the treeview's '"insert" method. To insert an item, you need to know where in the tree to insert it, which means specifying the parent item as well as what position in the list of the parent's children the new item should be inserted at.

The treeview widget automatically creates a root node (which is not displayed), having the id of "{}" (i.e. the empty string), which can serve as the parent of the first level of items you add. Positions within the list of the parent's children are specified by index (0 being the first, with the special "end" index meaning inserting after all existing children).

Normally, you'll also specify the name of each item, which is displayed in the tree. There are other options to add an image beside the name, specify whether the node is open or closed, and so on.

```
# Inserted at the root, program chooses id:
.tree insert {} end -id widgets -text "Widget Tour"

# Same thing, but inserted as first child:
.tree insert {} 0 -id gallery -text "Applications"

# Treeview chooses the id:
set id [.tree insert {} end -text "Tutorial"]

# Inserted underneath an existing node:
.tree insert widgets end -text "Canvas"
.tree insert $id end -text "Tree"
```

Inserting the item returns the id of the newly created item.

Rearranging Items

A node (and its descendants, if any) can be moved to a different location in the tree; the only restriction is that a node cannot be moved underneath one of its descendants. The target location is specified via parent and index into the list of children, as was done with "insert".

```
.tree move widgets gallery end; # move widgets under gallery
```

Items can be *detached* from the tree, which removes the item and its descendants from the hierarchy, but does not destroy the items, allowing you to later reinsert them with "move".

```
.tree detach widgets
```

Items can also be *deleted*, which does completely get rid of the item and its descendants.

```
.tree delete widgets
```

If you'd like to navigate the hierarchy, there are methods that let you find the parent of an item ("parent"), find the next or previous siblings of an item ("next" and "prev"), and return the list of children of an item ("children").

You can control whether or not the item is open and shows its children by modifying the "open" item configuration option.

```
.tree item widgets -open true
set isopen [.tree item widgets -open]
```

Displaying Information for each Item

The treeview can also display one or more additional pieces of information about each item, which are shown as columns to the right of the main tree display.

Again, each column is referenced by a symbolic name that you assign. You can specify the list of columns using the "columns" configuration option of the treeview widget, either when first creating the widget, or later on.

```
ttk::treeview .tree -columns "size modified"
.tree configure -columns "size modified owner"
```

You can specify the width of the column, how the display of item information in the column is aligned, and more. You can also provide information about the heading of the column, such as the text to display, an optional image, alignment, and a script to invoke when the item is clicked (e.g. to sort the tree).

```
.tree column size -width 100 -anchor center
.tree heading size -text "Size"
```

The values to place in each column for each item can be specified either individually, or by providing a list of values for the item. In the latter case, this is done using the "values" item configuration option (and so can be used either when first inserting the item or later) which takes a list of the values; the order of the list must be the same as the order in the "columns" widget configuration option.

```
.tree set widgets size "12KB"
set size [.tree set widgets size]
.tree insert "" end -text Listbox -values [list "15KB" "Yesterday" "mark"]
```

Item Appearance and Events

Like the text and canvas widgets, the treeview widget uses *tags* to help you modify the appearance of items in the tree. You can assign a list of tags to each item using the `"tags"` item configuration option (again, when creating the item or later on).

Tag configuration options can then be specified, which will then apply to all items having that tag. Valid tag options include `"foreground"` (text color), `"background"`, `"font"`, and `"image"` (not used if the item specifies its own image).

You can also create event bindings on tags, which let you capture mouse clicks, keyboard events etc.

```
.tree insert "" end -text button -tags "ttk simple"
.tree tag configure ttk -background yellow
.tree tag bind ttk <1> "itemclicked"; # the item clicked can be found via [.tree focus]
```

The treeview will generate virtual events `"<TreeviewSelect>"`, `"<TreeviewOpen>"` and `"<TreeviewClose>"` which allow you to monitor changes to the widget made by the user. You can use the `"selection"` method to determine the current selection (the selection can also be changed from your program).

Customizing the Display

There are many aspects of how the treeview widget is displayed that you can customize. Some of them we've already seen, such as the text of items, fonts and colors, names of column headings, and more. Here are a few additional ones.

- Specify the desired number of rows to show using the `"height"` widget configuration option.
- Control the width of each column using the column's `"width"` or `"minwidth"` options. The column holding the tree can be accessed with the symbolic name `"#0"`. The overall requested width for the widget is based on the sum of the column widths.
- Choose which columns to display and the order to display them in using the `"displaycolumns"` widget configuration option.
- You can optionally hide one or both of the column headings or the tree itself (leaving just the columns) using the `"show"` widget configuration option (default is "tree headings" to show both).
- You can specify whether a single item or multiple items can be selected by the user via the `"selectmode"` widget configuration option, passing `"browse"` (single item), `"extended"` (multiple items, the default), or `"none"`.

Styles and Themes

The "themed" aspect of the new Ttk widgets is one of the most powerful and exciting aspects of the new widget set. Yet because it does things quite differently from how Tk has traditionally worked, and because in trying to be flexible it does a *lot* of things, it's certainly the most confusing for many people.

Definitions

We'll first define a few concepts and terms that Ttk themes and styles rely on.

Widget Class

A *widget class* is used by Tk to identify the type of a particular widget; essentially, whether it is a button, a label, a canvas, etc. In classic Tk, all buttons had the same class ("Button"), all labels had the same class ("Label"), etc.

You could use this widget class both for introspection, and for changing options globally via the option database. This could let you say for example that all buttons by default had a red background.

There were a *few* classic Tk widgets, including frame and toplevel widgets, which would allow you to *change* the widget class of a particular widget when the widget was first created, by passing it a `"class"` configuration option. So while normally frames would have a widget class of "Frame", you could specify that one particular frame widget had a widget class of "SpecialFrame".

Because of that, you could use the option database to define different looks for *different types* of frame widgets (not just all frame widgets, or frame widgets located at a particular place in the hierarchy).

What Ttk does is take that simple idea and give it rocket boosters.

Widget State

A *widget state* allows a single widget to have more than one apperance or behavior, depending on things like mouse position, different state options set by the application, and so on. In classic Tk, several widgets had `"state"` configuration options which allowed you to set them to "normal" or "disabled"; a "disabled" state for a button for example would draw its label greyed out. Some used an additional state, "active", again which represented a different behavior.

The widget itself, or more typically, the widget's class bindings, controlled how the appearance of the widget changed in different states, typically via consulting widget configuration options like `"foreground"`, `"activeforeground"`, and `"disabledforeground"`.

Ttk again extends and generalizes this basic idea of widget state, in two important ways. First, rather than being widget-specific, all Ttk widgets have state options, and in fact, all have exactly the same state options, accessed by the `"state"` and `"instate"` widget commands.

A Ttk widget state is actually a set of independent state flags, containing zero or more of the following flags: `"active"`, `"disabled"`, `"focus"`, `"pressed"`, `"selected"`, `"background"`, `"readonly"`, `"alternate"`, or `"invalid"` (see the widget page in the reference manual for exact meanings).

Note that while all of these state flags are available for every widget, they may not be used by each widget. For example, a label widget is likely to ignore an `"invalid"` state flag, and so no special appearance would be associated with that flag.

The second major change that Ttk makes is that it takes the decision of what to change when the state is adjusted out of the widget's control. That is, a widget author will no longer hardcode logic to the effect of "when the state is disabled, consult the disabledforeground configuration option and use that for the foreground color". With that logic hardcoded, not only did it make coding widgets longer (and more repetitive), but also restricted how a widget could be changed based on its state. That is, if the widget author hadn't coded in logic to change the font when the state changed, you as the user of the widget were out of luck.

Instead of hardcoding these decisions within each widget, Ttk moves the decisions into a separate location: styles. This means that the widget author doesn't need to provide code for every possible appearance option, which not only simplifies the widget, but paradoxically ensures that a wider range of appearances *can* be set, including those the widget author may not have anticipated.

Style

That brings us then to define a widget style. Very simply, a *style* describes the appearance (or appearances) of a Ttk widget class. All widgets created with that widget class will have the same appearance(s). While each themed widget has a default class (e.g. "TButton" for "ttk::button" widgets), you can, unlike in classic Tk, assign a different widget class for any themed widget you create. This is done using Ttk's `"style"` configuration option, which all themed widgets support.

So a style defines the normal appearance of widgets of a certain widget class, but can also define variations of that appearance that depend on the current state flag. So for example a style can specify that when the "pressed" state flag is set, the appearance should change in a particular way. Because of this, a style can describe one or more ways for the widget to appear, depending on the state.

The rest of this chapter will delve into far more detail of what a style actually is, but at least now you know the responsibility it has.

Themes

You can think of a *theme* as a collection of styles. While each style is widget-specific (one for buttons, one for entries, etc.) a theme will collect many styles together. Typically, a theme will then define one style for each type of widget, but each of those styles will be designed so that they visually "fit" together with each other — though perhaps unfortunately Ttk doesn't technically restrict bad design or judgement!

To use a particular theme for an application is really to say that you'd like to have a set of styles defined so that by default all the different type of widgets will have some common appearance, and fit in well with each other.

Using Styles and Themes

So now we know what styles and themes are supposed to do, but how exactly do we use them? To do this, we need to know how to refer to styles and themes, and how to apply them to a widget or user interface.

Style Names

Every style has a name. If you're going to modify a style, create a new one, or use a style for a widget, you need to know its name.

How do you know what the names of the styles are? If you have a particular widget, and you want to know what style it is currently using, you can first check the value of its `"style"` configuration option. If that is empty, it means the widget is using the *default* style for the widget. You can retrieve that via the widget's class. For example:

```
% ttk::button .b
.b
% .b cget -style      # empty string as a result
% winfo class .b
```

```
TButton
```

So in this case, the style that is being used is "TButton". The default styles for other themed widgets are named similarly, e.g. "TEntry", "TLabel", "TSizeGrip", etc. It's always wise to check specifics though; for example, the treeview widget's class is "Treeview", not "TTreeview".

Beyond the default styles though, styles can be named pretty much anything. You might create your own style (or use a theme that has a style) named "FunButton", "NuclearReactorButton", or even "GuessWhatIAm" (not a smart choice). More often, you'll find names like "Fun.TButton" or "NuclearReactor.TButton", which suggest variations of a base style; as you'll see, this is something Ttk supports for creating and modifying styles.

> *The ability to retrieve a list of all currently available styles is currently not supported.*

Using a Style

To use a style means to apply that style to an individual widget. If you know the name of the style you want to use, and which widget to apply it to, it's easy. Setting the style can be done at creation time:

```
ttk::button .b -text "Hello" -style "Fun.TButton"
```

As well, you can change the style of a widget at anytime after you've created it with the `"style"` configuration option:

```
.b configure -style "NuclearReactor.TButton"
```

Using Themes

While styles control the appearance of individual widgets, themes control the appearance of the entire user interface. The ability to switch between themes is one of the significant features of the themed widgets.

Like styles, themes are identified by a name. You can obtain the names of all available themes:

```
% ttk::style theme names
aqua clam alt default classic
```

Only one theme can ever be active at a time. To obtain the name of the theme currently in use, you can use the following:

> *This API, which was originally targeted for Tk 8.6, was back-ported to Tk 8.5.9. If you're using an earlier version of Tk getting this info is a bit trickier.*

```
% ttk::style theme use
aqua
```

Switching to a new theme can be done with:

```
ttk::style theme use themename
```

What does this actually do? Obviously, it sets the current theme to the indicated theme. Doing this therefore replaces all the currently available styles with the set of styles defined by the theme. Finally, it refreshes all widgets, so that they take on the appearance described by the new theme.

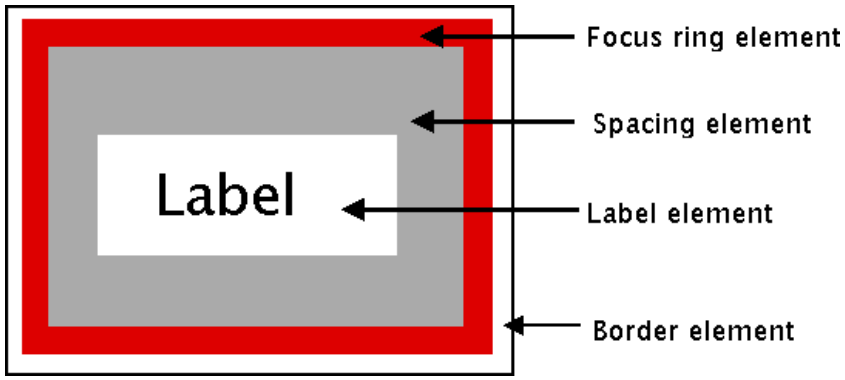
What's Inside a Style?

If all you want to do is *use* a style, you now know everything you need. If however, you want to create your own styles, or modify an existing one, now it gets "interesting".

Elements

While each style represents a single widget, each widget is normally composed of smaller pieces, called *elements*. It's the job of the style author to construct the entire widget out of these smaller elements. What these elements are depends on the widget.

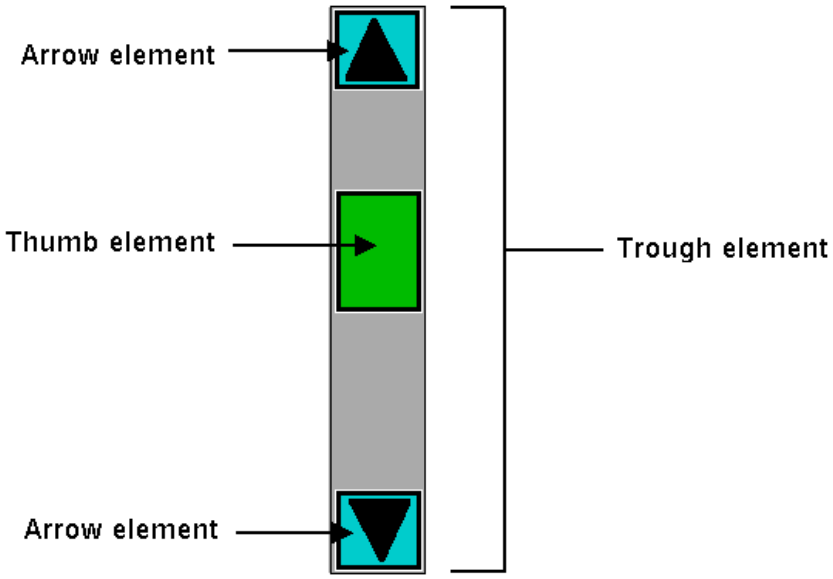
Here's an example of a button. It might have a border on the very outside, which is one element. Just inside that there may be a focus ring, which is normally just the background color but may be highlighted when the user tabs into the button. So that's a second element. Then there might be some spacing between that focus ring and the button's label. So that spacing would be a third element. Finally, there is the label of the button itself, a fourth element.



Possible Elements of a Button.

Why might the style author have divided it up that way? If you have one part of the widget that may be in a different location from another, or might be a different color than another, it may be a good candidate for an element. Note that this is just one example of how a button could be constructed from elements. Different styles and themes could (and do) accomplish this in different ways.

Here is a second example of a vertical scrollbar, containing a "trough" element containing the rest, which includes the up and down arrow elements at either end, and a "thumb" element in the middle.



Possible Elements of a Scrollbar.

Layout

Besides the choice of which elements are part of a widget, a style also defines how those elements are arranged within the widget, or in other words, their layout. In the button example, we had a label element inside a spacing element, inside a focus ring element, inside a border element. So the logical layout is like this:

```
border {
    focus {
        spacing {
            label
        }
    }
}
```

We can ask Ttk what the layout of the TButton style is like this:

```
% ttk::style layout TButton
Button.border -sticky nswe -border 1 -children {Button.focus -sticky nswe
-children {Button.spacing -sticky nswe -children {Button.label -sticky nswe}}}
```

If we clean this up and format it a bit, we get something with this structure:

```
Button.border -sticky nswe -border 1 -children {
    Button.focus -sticky nswe -children {
        Button.spacing -sticky nswe -children {
            Button.label -sticky nswe
        }
    }
```

```
}
```

This starts to make sense; we have four elements, named "Button.border", "Button.focus", "Button.spacing", and "Button.label". Each of these has different element options, such as "-children", "-sticky", and "-border" that here specify layout or sizes. Without getting into two much detail at this point, we can clearly see the nested layout, based on the "-children" and "-sticky" attributes; Ttk uses a simplified version of Tk's "pack" geometry manager to specify element layout.

Element Options

Each of these different elements may have a number of different options. For example, the thumb of the scrollbar may have an option to set its background color, or another to provide the width of a border, if any. These can be customized to adjust how the elements within the widget look.

What options are available for each element? Here's an example of checking what options are available for the label inside the button (which we know from the "layout" command is identified as "Button.label"):

```
% ttk::style element options Button.label
-compound -space -text -font -foreground -underline -width -anchor -justify
-wraplength -embossed -image -stipple -background
```

In the next sections, we'll take a look at the not-entirely-straightforward way that you can work with element options.

Changing Style Options

In this section, we'll look at how you can change the style's appearance via modifying style options. You can do this either by modifying an existing style, or more typically, by creating a new style.

Modifying a Style Option

Modifying a configuration option for an existing style is done in a similar fashion as modifying any other configuration option, by specifying the style, name of the option, and new value:

```
ttk::style configure TButton -font "helvetica 24"
```

You'll learn more about what the valid options are shortly.

If you need to retrieve the current value of an option, this can be done with the `"lookup"` method.

```
% ttk::style lookup TButton -font
helvetica 24
```

Creating a New, Derived Style

If you modify an existing style, such as "TButton", that modification will apply to all widgets using that style (so by default, all buttons). That may well be what you want to do.

More often, you're interested in creating a new style that is similar to an existing one, but varies in a certain aspect. For example, you'd like to have most of the buttons in your application keep their normal appearance, but create certain special "emergency" buttons, which will be highlighted in a different way. In this case, creating a new style (e.g. "Emergency.TButton") derived from the base style ("TButton") would be the appropriate thing to do.

By prepending another name ("Emergency") followed by a dot onto an existing style, you are implicitly creating a new style derived from the existing one. So in this example, our new style will have exactly the same options as a regular button except for the indicated differences:

```
ttk::style configure Emergency.TButton -font "helvetica 24"
-foreground red -padding 10
```

State Specific Style Options

Besides the normal configuration options for the style, the widget author may have specified different options to use when the widget is in a particular widget state. For example, when a button is disabled, you'd like to have the color of the button's label greyed out.

To do this, you can specify a "map", which allows you to specify variations for one or more of a style's configuration options. For each configuration option, you can specify a list of widget states, along with the particular value the option should be assigned when the widget is in that state.

> *Remember that the state is composed of one or more state flags (or their negation), as set by the widget's `"state"` method, or queried via the `"instate"` method.*

The following example provides for the following variations from a button's "normal" appearance:

- when the widget is in the disabled state, the background color should be set to "#d9d9d9"
- when the widget is in the active state (mouse over it), the background color should be set to "#ececec"
- when the widget is in the disabled state, the foreground color should be set to "#a3a3a3" (this is in addition to the background color change we already noted)

- when the widget is in the state where the button is pressed and the widget is not disabled, the relief should be set to "sunken"

```
ttk::style map TButton \
        -background [list disabled #d9d9d9  active #ececec] \
        -foreground [list disabled #a3a3a3] \
        -relief [list {pressed !disabled} sunken] \
        ;
```

Remember that in the past, with classic Tk widgets, exactly what changed when the widget was in each state would have been determined solely by the widget author. With themed widgets, it is the style itself that determines what changes, which could include things that the original widget author had never anticipated.

> *Because widget states can contain multiple flags, it's possible that more than one state will match for an option (e.g. "pressed" and "pressed !disabled" will both match if the widget's "pressed" state flag is set). The list of states is evaluated in the order you provide in the map command, with the first state in the list that matches being used.*

Sound Difficult to you?

So you now know that styles are made up of elements, which have a variety of options, and are composed together in a particular layout. You can change various options on styles, to make all widgets using the style appear differently. Any widgets using that style take on the appearance that the style defines. Themes collect an entire set of related styles, making it easy to change the appearance of your entire user interface.

So what makes styles and themes so difficult in practice? Three things. First:

> **You can only modify options for a style, not element options (except sometimes).**

We talked earlier about how to discover what elements were used in the style by examining the style's layout, and also how to discover what options were available for each element. But when we went to make changes to a style, we seemed to be configuring an option for the style, without specifying an individual element. What's going on?

Again, using our button example, we had an element "Button.label", which among other things had a `"font"` configuration option. What happens is that when that "Button.label" element is drawn, it looks at the `"font"` configuration option set on the *style* to determine what font to draw itself in.

> *To understand why, you need to know that when a style includes an element as a piece of it, that element does not maintain any (element-specific) storage. In particular, it does not store any configuration options itself. When it needs to retrieve options, it does so via the containing style, which is passed to the element. Individual elements therefore are "flyweight" objects in GoF pattern parlance.*

Similarly, any other elements will lookup their configuration options from options set on the style. What if two elements use the same configuration option (like a background color)? Because there is only one background configuration option, stored in the style, that means both elements will use the same background color. You can't have one element use one background color, and the other use a different background color.

> *Except when you can. There are a few nasty, widget-specific things called "sublayouts" in the current implementation which let you sometimes modify just a single element, via configuring an option like "TButton.Label" (rather than just "TButton", the name of the style). Are the cases where you can do this documented? Is there some way to introspect to determine when you can do this? No to both. This is one area of the themed widget API that I definitely expect to evolve over time.*

The second difficulty is also related to modifying style options:

> **Options that are available don't necessarily have an effect, and it's not an error to modify a bogus option.**

You'll sometimes try to change an option that is supposed to exist according to element options, but it will have no effect. As an example, you can't modify the background color of a button in the "aqua" theme used by Mac OS X. While there are valid reasons for these cases, at the moment its not easy to discover them, which can make experimenting frustrating at times.

Perhaps more frustrating when you're experimenting is that specifying an "incorrect" style name or option name does not generate an error. When doing a "configure" or "lookup" you can in fact specify any name at all for a style, and specify any name at all for an option. So if you're bored with the "background" and "font" options, feel free to configure a "dowhatimean" option. It may not do anything, but it's not an error. Again, it may make it hard to know what you should be modifying and what you shouldn't.

> *This is one of the downsides of having a very lightweight and dynamic system. You can create new styles by just providing their name when configuring style options; this means you don't need to explicitly create a style object. At the same time, this does open itself to errors. It's also not possible*

> *to find out what styles currently exist or are used. And because style options are really just a front end for element options, and the elements in a style can change at any time, it's not necessarily obvious that options should be restricted to those referred to by current elements alone, which may themselves not all be introspectable.*

Finally, here is the last thing that makes styles and themes so difficult:

> **The elements available, the names of those elements, which options are available or have an effect for each of those elements, and which are used for a particular widget can be different in every theme.**

So? Keep in mind among other things that the default theme for each platform (Windows, Mac OS X, and Linux) are different, (which is a good thing). Some implications of this:

1. If you want to define a new type of widget (or more likely a variation of an existing widget) for your application, you're going to need to do it separately and differently for each theme your application uses (so at least three for a cross-platform application).
2. The elements and options available will differ for each theme/platform, meaning you may have to come up with a quite different customization approach for each theme/platform.
3. The elements, names, and element options available with each theme are not typically documented (outside of reading the theme definition files themselves), but are generally identified via theme introspection (which we'll see soon). Because all themes aren't available on all platforms (e.g. "aqua" will only run on Mac OS X), you'll need ready access to every platform and theme you need to run on.

As an example, here is what the layout of the "TButton" style looks like on the theme used by default on three different platforms, as well as the advertised options for each element (not all of which have an effect):

Mac OS X	*Button.button -sticky nswe -children {Button.padding -sticky nswe -children {Button.label -sticky nswe}}*	
	Button.button	-
	Button.padding	padding, relief, shiftrelief
	Button.label	compound, space, text, font, foreground, underline, width, anchor, justify, wraplength, embossed, image, stipple, background
Windows	*Button.button -sticky nswe -children {Button.focus -sticky nswe -children {Button.padding -sticky nswe -children {Button.label -sticky nswe}}}*	
	Button.button	-
	Button.focus	-
	Button.padding	padding, relief, shiftrelief
	Button.label	compound, space, text, font, foreground, underline, width, anchor, justify, wraplength, embossed, image, stipple, background
Linux	*Button.border -sticky nswe -border 1 -children {Button.focus -sticky nswe -children {Button.padding -sticky nswe -children Button.label -sticky nswe}}}*	
	Button.border	background, borderwidth, relief
	Button.focus	focuscolor, focusthickness
	Button.padding	padding, relief, shiftrelief
	Button.label	compound, space, text, font, foreground, underline, width, anchor, justify, wraplength, embossed, image, stipple, background

The bottom line is that in classic Tk, where you had the ability to modify any of a large set of attributes for an individual widget, you'd be able to do something on one platform and it would sorta kinda work (but probably need tweaking) on others. In themed Tk, the easy option just isn't there, and you're pretty much forced to do it the right way if you want your application to work with multiple themes/platforms. It's more work up front.

Advanced: More on Elements

While that's about as far as we're going to go on styles and themes in this tutorial, for curious users and those who want to delve further into creating new themes, we can provide a few more interesting tidbits about elements.

Because elements are the building blocks of styles and themes, it begs the question of "where do elements come from?" Practically speaking, we can say that elements are normally created in C code, and conform to a particular API that the theming engine understands.

At the very lowest level, elements come from something called an *element factory*. At present, there is a "default" one, which most themes use, and uses Tk drawing routines to create elements. A second allows you to create elements from images, and is actually accessible at the script level using the `"ttk::style element create"` method (from Tcl). Finally, there is a third, Windows-specific engine using the underlying "Visual Styles" platform API.

If a theme uses elements created via a platform's native widgets, the calls to use those native widgets will normally appear within that theme's element specification code. Of course, themes whose elements depend on native widgets or API calls can only run on the platforms that support them.

Themes will then take a set of elements, and use those to assemble the styles that are actually used by the widgets.

And given the whole idea of themes is so that several styles can share the same appearance, it's not surprising that different styles share the same elements.

So while the "TButton" style includes a "Button.padding" element, and the "TEntry" style includes a "Entry.padding" element, underneath these padding elements are more than likely one and the same. They may appear differently, but that's because of different configuration options, which as we recall, are stored in the style that uses the element.

It's also probably not surprising to find out that a theme can provide a set of common options which are used as defaults for each style, if the style doesn't specify them otherwise. This means that if pretty much everything in an entire theme has a green background, the theme doesn't need to explicitly say this for each style. This uses a root style named "."; after all, if "Fun.TButton" can inherit from "TButton", why can't "TButton" inherit from "."?

Finally, it's worth having a look at how existing themes are defined, both at the C code level in Tk's C library, but also via the Tk scripts found in Tk's "library/ttk" directory. Search for `"Ttk_RegisterElementSpec"` in Tk's C library to see how elements are specified.

Table of Contents