Essentials

- Tk Backgrounder (../resources/backgrounder.html)
- Installing Tk (../tutorial/install.html)
- Tutorial (../tutorial/index.html)
- Widget Roundup (../widgets/index.html)
- Languages Using Tk (../resources/languages.html)
- Official Tk Command Reference
  (Tcl-oriented; at www.tcl.tk) (http://www.tcl.tk/man/tcl8.6/TkCmd/contents.htm)

Tutorial

Show: All Languages ▾

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.

Previous: Menus (menus.html) | Contents (index.html) | Single Page (onepage.html) | Next: Organizing Complex Interfaces (complex.html)

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
```

Toplevel windows are created using the **`TkToplevel`** class:

```
t = TkToplevel.new( parent )
```

Toplevel windows are created using the **`new_toplevel`** method, a.k.a. **`Tkx::toplevel`**:

```
my $win = $parent->new_toplevel;
```

Toplevel windows are created using the **`Toplevel`** function:

```
t = Toplevel(parent)
```

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.*

To destroy a window, you can use the **`destroy`** method on a widget:

```
window.destroy
```

To destroy a window, you can use the **`g_destroy`** method on a widget:

```
$window->g_destroy
```

To destroy a window, you can use the **`destroy`** method on a widget:

```
window.destroy()
```

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 macOS and Windows.*

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

```
oldtitle = window['title']
window['title'] = "New title"
```

```
my $oldtitle = $window->g_wm_title;
$window->g_wm_title("New title");
```

```
oldtitle = window.title()
window.title('New title')
```

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 is the actual coordinates on the screen, and doesn't make allowances for systems like macOS 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
```

```
window['geometry'] = '300x200-5+40'
```

```
$window->g_wm_geometry("300x200-5+40");
```

```
window.geometry('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 highest, via:

```
wm stackorder .window
```

```
window.stackorder
```

```
$window->wm_stackorder
```

```
root.tk.eval('wm stackorder '+str(window))
```

> *This method doesn't appear to be exposed cleanly through Tkinter's API.*

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]} {...}
```

```
if (window.stackorder_isabove otherwin) ...
if (window.stackorder_isbelow otherwin) ...
```

```
if ($window->g_wm_stackorder("isabove", $otherwin)) ...
if ($window->g_wm_stackorder("isbelow", $otherwin)) ...
```

```
if (root.tk.eval('wm stackorder '+str(window)+' isabove '+str(otherwindow))=='1') ...
if (root.tk.eval('wm stackorder '+str(window)+' isbelow '+str(otherwindow))=='1') ...
```

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
```

```
window.raise
window.raise_window otherwin
window.lower
window.lower_window otherwin
```

```
$window->g_raise;
$window->g_raise($otherwin);
$window->g_lower;
$window->g_lower($otherwin);
```

```
window.lift()
window.lift(otherwin)
window.lower()
window.lower(otherwin)
```

> *Tkinter uses the name 'lift', since 'raise' is a reserved keyword.*

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
```

```
require 'tk'
require 'tkextlib/tile'
root = TkRoot.new
little = Tk::Tile::Label.new(root) {text 'Little'}.grid( :column => 0, :row => 0)
bigger = Tk::Tile::Label.new(root) {text 'Much Bigger Label'}.grid( :column => 0, :row => 0)
Tk.after 2000, proc{little.raise}
Tk.mainloop
```

```
use Tkx;
$mw = Tkx::widget->new(".");
$little = $mw->new_ttk__label(-text => "Little");
$little->g_grid(-column => 0, -row => 0);
$bigger = $mw->new_ttk__label(-text => "Much Bigger Label");
$bigger->g_grid(-column => 0, -row => 0);
Tkx::after(2000, sub {$little->g_raise();});
Tkx::MainLoop();
```

```
from tkinter import *
from tkinter import ttk
root = Tk()
little = ttk.Label(root, text="Little")
bigger = ttk.Label(root, text='Much bigger label')
little.grid(column=0,row=0)
bigger.grid(column=0,row=0)
root.after(2000, lambda: little.lift())
root.mainloop()
```

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
```

```
window['resizable'] = false, false
```

```
$window->g_wm_resizable(0,0);
```

```
window.resizable(FALSE,FALSE)
```

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
```

```
window['minsize'] = 200, 100
window['maxsize'] = 500, 500
```

```
$window->g_wm_minsize(200,100);
$window->g_wm_maxsize(500,500);
```

```
window.minsize(200,100)
window.maxsize(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
```

```
thestate = window['state']
window['state'] = 'normal'
window.iconify
window.deiconify
```

```
my $thestate = $window->g_wm_state;
$window->g_wm_state("normal");
$window->g_wm_iconify;
$window->g_wm_deiconify;
```

```
thestate = window.state()
window.state('normal')
window.iconify()
window.deiconify()
```

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).

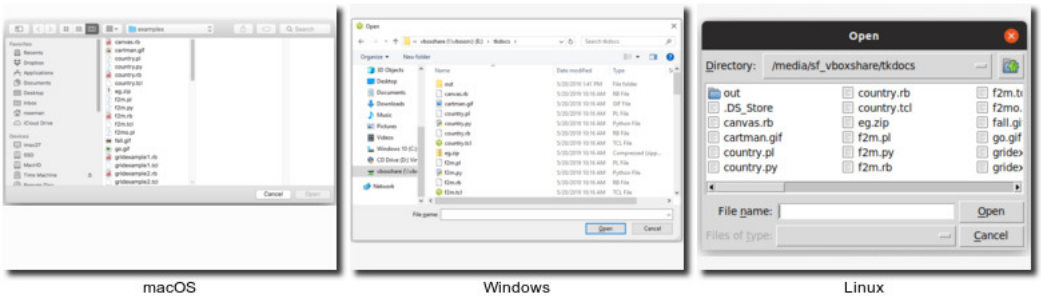
```
set filename [tk_getOpenFile]
set filename [tk_getSaveFile]
set dirname [tk_chooseDirectory]
```

```
filename = Tk::getOpenFile
filename = Tk::getSaveFile
dirname = Tk::chooseDirectory
```

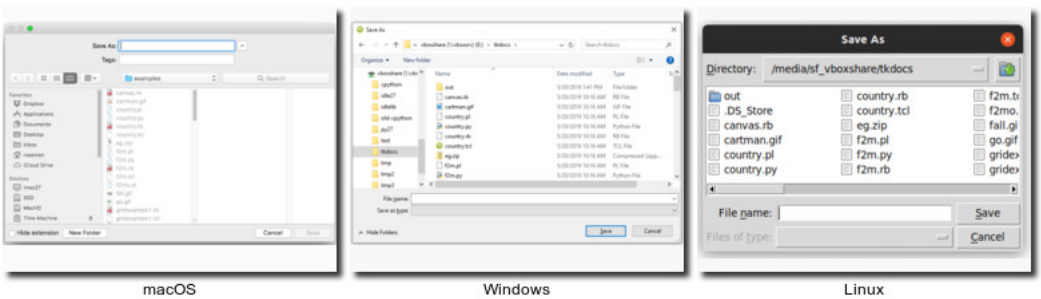
```
$filename = Tkx::tk___getOpenFile();
$filename = Tkx::tk___getSaveFile();
$dirname = Tkx::tk___chooseDirectory();
```

```
from tkinter import filedialog
filename = filedialog.askopenfilename()
filename = filedialog.asksaveasfilename()
dirname = filedialog.askdirectory()
```

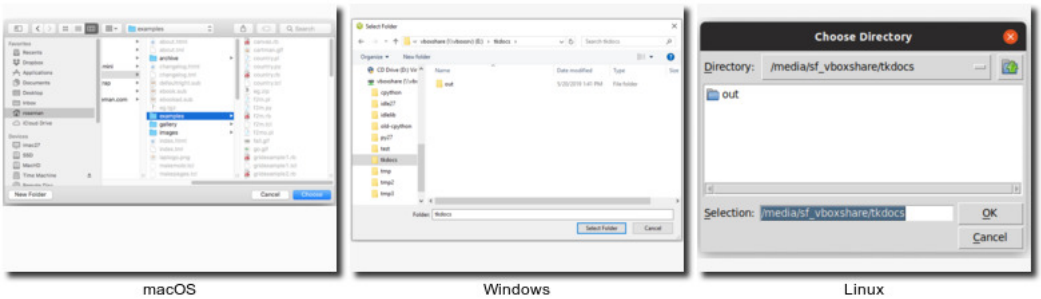
All of these commands produce *modal* dialogs, which means that the commands (and hence your program) will not continue running until the user submits the dialog. The commands return the full pathname of the file or directory the user has chosen, or return an empty string if the user cancels out of the 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 (http://www.tcl.tk/man/tcl8.6/TkCmd/getOpenFile.htm) and chooseDirectory (http://www.tcl.tk/man/tcl8.6/TkCmd/chooseDirectory.htm) reference manual pages.

Selecting Colors

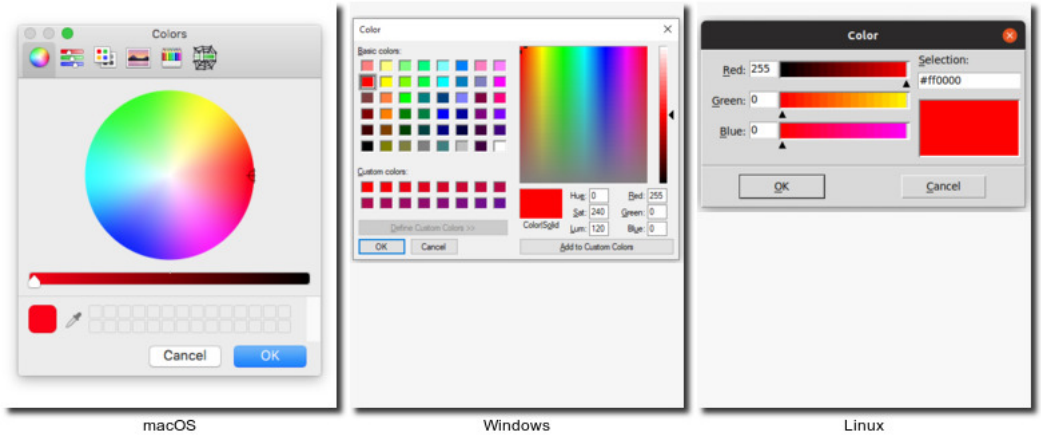
There is also a modal dialog to let the user select a color. It will return a color value, e.g. "#ff62b8". The dialog takes an optional `"initialcolor"` option to specify an existing color that the user is presumably replacing.

```
tk_chooseColor -initialcolor #ff0000
```

```
Tk::chooseColor :initialcolor => '#ff0000'
```

```
Tkx::tk___chooseColor(-initialcolor => "#ff0000");
```

```
from tkinter import colorchooser
colorchooser.askcolor(initialcolor='#ff0000')
```



Choose Color Dialogs

> *One thing missing in Tk 8.5 is a font chooser dialog. That has actually been added in Tk 8.6. You can find details in the reference manual (http://www.tcl.tk/man/tcl8.6/TkCmd/fontchooser.htm).*

Alert and Confirmation Dialogs

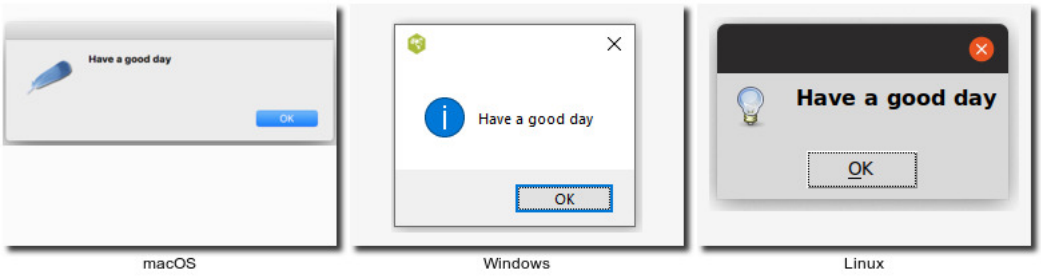
Many applications use various simple modal alerts or dialogs to notify the user of an event, ask them to confirm an action, or make another similar choice which is done by clicking on a button. Tk provides a versatile "message box" that encapsulates all these different types of dialogs.

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

```
Tk::messageBox :message => 'Have a good day'
```

```
Tkx::tk___messageBox(-message => "Have a good day");
```

```
from tkinter import messagebox
messagebox.showinfo(message='Have a good day')
```



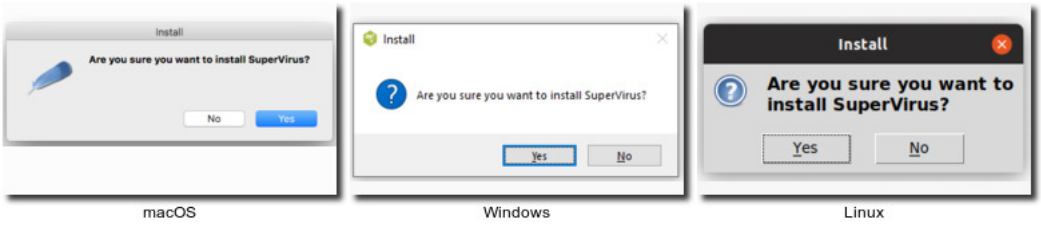
Simple Message Boxes

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

```
Tk::messageBox :type => 'yesno',
            :message => 'Are you sure you want to install SuperVirus?',
            :icon => 'question', :title => 'Install'
```

```
Tkx::tk___messageBox(-type => "yesno",
            -message => "Are you sure you want to install SuperVirus?",
            -icon => "question", -title => "Install");
```

```
messagebox.askyesno(
           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"`

> Rather than using a 'type' option, Tkinter uses a different method name for each type of dialog. The available methods are: askokcancel, askquestion, askretrycancel, askyesno, askyesnocancel, showerror, showinfo, showwarning.

The full list of possible options is shown here:

`type`	As described above.
`message`	The main message displayed inside the alert.
`detail`	A secondary message (if needed).
`title`	Title for the dialog window. Not used on macOS.
`icon`	Icon, one of `"info"` (default), `"error"`, `"question"` or `"warning"`.
`default`	Default button, e.g. `"ok"` or `"cancel"` for a `"okcancel"` dialog.
`parent`	Window of your application this dialog is being posted for.

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

Previous: Menus (menus.html) | Contents (index.html) | Single Page (onepage.html) | Next: Organizing Complex Interfaces (complex.html)