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: More Widgets (morewidgets.html) | Contents (index.html) | Single Page (onepage.html) | Next: Windows and Dialogs (windows.html)

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
```

```
Tk.windowingsystem;  # will return x11, win32 or aqua
```

```
Tkx::tk_windowingsystem();   # will return x11, win32 or aqua
```

```
root.tk.call('tk', 'windowingsystem')     # will return x11, win32 or aqua
```

> *To the best of my knowledge, Tkinter does not provide a direct equivalent to this call. However, as you can see from the example, it is possible to execute a Tcl-based Tk command directly, using the ".tk.call()" function available on any Tkinter widget.*

> *This is probably more useful than examining global variables like `tcl_platform` or `sys.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.

> *You'll notice on some recent Linux distributions that many applications show their menus at the top of the screen when active, rather than in the window itself. Tk does not yet support this style of menus.*

Menu Widgets and Hierarchy

- Widget Roundup (../widgets/menu.html)
- Reference Manual (http://www.tcl.tk/man/tcl8.6/TkCmd/menu.htm)

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
```

```
TkOption.add '*tearOff', 0
```

```
Tkx::option_add("*tearOff", 0);
```

```
root.option_add('*tearOff', FALSE)
```

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 macOS 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 macOS 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.

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

```
win = TkToplevel.new(root)
menubar = TkMenu.new(win)
win['menu'] = menubar
```

```
$w = $mw->new_toplevel;
$m = $w->new_menu;
$w->configure(-menu => $m);
```

```
win = Toplevel( root )
menubar = Menu(win)
win['menu'] = menubar
```

Note that you can use the same menubar for more than one window (i.e. use one menubar as the value of the "menu" configuration option for different toplevel windows). This is particularly useful on Windows and X11, where you may want a window to include a menu, but don't necessarily need to juggle different menus in your application. But remember, if the contents or state of the menubar depend on what's going on in the active window, you'll have to deal with that yourself.

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
```

```
file = TkMenu.new(menubar)
edit = TkMenu.new(menubar)
menubar.add :cascade, :menu => file, :label => 'File'
menubar.add :cascade, :menu => edit, :label => 'Edit'
```

```
$m = $w->new_menu;
$file = $m->new_menu;
$edit = $m->new_menu;
$m->add_cascade(-menu => $file, -label => "File");
$m->add_cascade(-menu => $edit, -label => "Edit");
```

```
menubar = Menu( parent )
menu_file = Menu(menubar)
menu_edit = Menu(menubar)
menubar.add_cascade(menu=menu_file, label='File')
menubar.add_cascade(menu=menu_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"
```

```
file.add :command, :label => 'New', :command => proc{newFile}
file.add :command, :label => 'Open...', :command => proc{openFile}
file.add :command, :label => 'Close', :command => proc{closeFile}
```

```
$file->add_command(-label => "New", -command => sub {newFile()});
$file->add_command(-label => "Open...", -command => sub {openFile()});
$file->add_command(-label => "Close", -command => sub {closeFile()});
```

```
menu_file.add_command(label='New', command=newFile)
menu_file.add_command(label='Open...', command=openFile)
menu_file.add_command(label='Close', command=closeFile)
```

On macOS, 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
```

```
file.add :separator
```

```
$file->add_separator
```

```
menu_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
```

```
check = TkVariable.new
file.add :checkbutton, :label => 'Check', :variable => check, :onvalue => 1, :offvalue => 0
radio = TkVariable.new
file.add :radiobutton, :label => 'One', :variable => radio, :value => 1
file.add :radiobutton, :label => 'Two', :variable => radio, :value => 2
```

```
$file->add_checkbutton(-label => "Check", -variable => \$check, -onvalue => 1, -offvalue => 0);
$file->add_radiobutton(-label => "One", -variable => \$radio, -value => 1);
$file->add_radiobutton(-label => "Two", -variable => \$radio, -value => 2);
```

```
check = StringVar()
menu_file.add_checkbutton(label='Check', variable=check, onvalue=1, offvalue=0)
radio = StringVar()
menu_file.add_radiobutton(label='One', variable=radio, value=1)
menu_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 macOS 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 macOS 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 macOS, 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
```

```
puts( file.entrycget 0, :state ); # get label of top entry in menu
file.entryconfigure 'Close', :state => 'disabled'; # change an entry
puts( file.entryconfiginfo 0 ); # print info on all optinos for an item
```

```
print $file->entrycget(0, -label); # get label of top entry in menu
$file->entryconfigure("Close", -state => "disabled"); # change an entry
print $file->entryconfigure(0); # print info on all options for an item
```

```
print( menu_file.entrycget(0, 'label') )
menu_file.entryconfigure('Close', state=DISABLED)
print( menu_file.entryconfigure(0) )
```

Platform Menus

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

macOS

You've probably noticed if you've been playing around with the previous examples on macOS, that Tk supplies its own default menubar, including a menu named after the program being run (in this case, your programming language's shell, e.g. 'Wish', 'Python', etc.), a File menu, and standard Edit, Windows, and Help menu, all stocked with various menu items.

You can override this menubar in your own program, but to get the results you want, you'll need to follow some particular steps (in some cases, in a particular order).

> *Starting at Tk 8.5.13, the handling of special menus on the Mac changed, a result of the underlying Tk code switching from the obsolete Carbon API to Cocoa. If you're seeing duplicate menu names, missing items, things you didn't put there, etc. review this section carefully.*

The first thing to know is that if you don't specify a menubar for a window (or its parent window, e.g. the root window) you'll end up with the default menubar Tk supplies, which unless you're just mucking around on your own, is almost certainly not what you want.

The Application Menu

If you do supply a menubar, *at the time the menubar is attached to the window*, if there is *not* a specially named ".apple" menu (see below), Tk will provide a standard application menu, *named after the binary being run*. It will contain an "About Tcl & Tk" item, followed by the standard menu items: preferences, the services submenu, hide/show items, and quit. Again, you don't want this.

If you supply your own ".apple" menu, when the menubar is attached to the window, it will add the standard items (preferences and onward) onto the end of any items you have added. Perfect! (Items you add *after* the menubar is attached to the window will appear after the quit item, which, again, you don't want.)

> *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.*

So in other words, in your program, make sure you:

1. Create a menubar for each window, or the root window. *Do not attach the menubar to the window yet!*
2. Add a menu to the menubar named ".apple" which will be used as the application menu.
3. The title of the menu will automatically be named the same as the application binary; if you want to change this, rename (or make a copy of) the binary used to run your script.
4. Add the items you want to appear at the top of the application menu, i.e. an "About yourapp" item, followed by a separator.
5. *After* you have done all this, you can *then* attach the menubar to your window, via the window's `"menu"` configuration option.

```
toplevel .win
menu .win.menubar
.win.menubar add cascade -menu [menu .win.menubar.apple]
.win.menubar.apple add command -label "About My Application"
.win.menubar.apple add separator
.win configure -menu .win.menubar
```

The pathname of the application menu must be `".apple"`.

```
win = TkToplevel.new(root)
menubar = TkMenu.new(win)
appmenu = TkSysMenu_Apple.new(menubar)
menubar.add :cascade, :menu => appmenu
appmenu.add :command, :label => 'About My Application'
appmenu.add :separator
win['menu'] = menubar
```

The TkSysMenu_Apple call ensures the menu is named '.apple' internally.

```
$w = $mw->new_toplevel;
$m = $w->new_menu;
$appmenu = Tkx::widget->new(Tkx::menu($m->_mpath . ".apple"));
$m->add_cascade(-menu => $appmenu);
$appmenu->add_command(-label => "About My Application")
$appmenu->add_separator;
$w->configure(-menu => $m);
```

While normally Tkx chooses a widget path name for us, here we've had to explicitly provide one ('.apple') using the '_mpath' option when creating the application menu.

```
win = Toplevel(root)
menubar = Menu(win)
appmenu = Menu(menubar, name='apple')
menubar.add_cascade(menu=appmenu)
appmenu.add_command(label='About My Application')
appmenu.add_separator()
win['menu'] = menubar
```

While normally Tkinter chooses a widget path name for us, here we've had to explicitly provide one ('apple') using the 'name' option when creating the application menu.

Handling the Preferences Menu Item

As you've noticed, the application menu always includes a "Preferences..." menu item; this is automatically included. If your application has a preferences dialog, selecting this menu item should open it. If your application has no preferences dialog, this menu item should be disabled, which it is by default.

To hook up your preferences dialog, 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.

```
proc tk::mac::ShowPreferences {} {showMyPreferencesDialog}
```

To hook up your preferences dialog, 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.

```
Tk.ip_eval("proc ::tk::mac::ShowPreferences {} {#{Tk.install_cmd(proc{showMyPreferencesDialog})}}")
```

To hook up your preferences dialog, 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.

```
Tkx::proc('::tk::mac::ShowPreferences', '{args}', sub {showMyPreferencesDialog()});
```

To hook up your preferences dialog, 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.

```
def showMyPreferencesDialog():
    ....

root.createcommand('tk::mac::ShowPreferences', showMyPreferencesDialog)
```

Providing a Help Menu

Like the application menu, any help menu you add to your own menubar is treated specially on macOS. As with the application menu that needed a special name ('.apple'), the help menu must be given the name '.help'. Also like the application menu, the help menu should also be added *before the menubar is attached to the window*.

The help menu will include the standard OS X search box to search help, as well as an item named '*yourapp* Help'. As with the name of the application menu, the name of this item comes from the name of the binary running your program and cannot be changed. Similar to how preferences dialogs are handled, to respond to this help item, you need to define a Tcl procedure named "`::tk::mac::ShowHelp`". Unlike with preferences, not defining this procedure will generate an error, not disable the menu item.

> *If you don't want to include help, just don't add a help menu to the menubar, and none will be shown.*

> *Unlike on X11 and earlier versions of Tk on macOS, the Help menu will not automatically be put at the end of the menubar, so ensure it is the last menu added.*

You can also add other items to the help menu, which will appear after the application help item.

```
.win.menubar add cascade -menu [menu .win.menubar.help] -label Help
::tk::mac::ShowHelp {} {...}
```

```
helpmenu = TkSysMenu_Help.new(menubar)
menubar.add :cascade, :menu => helpmenu, :label => 'Help'
Tk.ip_eval("proc ::tk::mac::ShowHelp {} {#{Tk.install_cmd(proc{...})}}")
```

```
$helpmenu = Tkx::widget->new(Tkx::menu($m->_mpath . ".help"));
$m->add_cascade(-menu => $helpmenu);
Tkx::proc('::tk::mac::ShowHelp', '{args}', sub {...});
```

```
helpmenu = Menu(menubar, name='help')
menubar.add_cascade(menu=helpmenu, label='Help')
root.createcommand('tk::mac::ShowHelp', ...)
```

Providing a Window Menu

On macOS, a 'Window' menu is used to contain items like minimize, zoom, bring all to front, etc. It also contains a list of currently open windows. Before that list, other application-specific items are sometimes provided.

By providing a menu named ".window", this standard window menu will be added, and Tk will automatically keep it in sync with all your toplevel windows, without any extra code on your part. You can also add any application-specific commands to this menu, which will appear before the list of your windows.

```
.win.menubar add cascade -menu [menu .win.menubar.window] -label Window
```

```
class Tk::TkSysMenu_Window<Tk::Menu
  include Tk::SystemMenu
  SYSMENU_NAME = 'window'
end
windowmenu = Tk::TkSysMenu_Window.new(menubar)
menubar.add :cascade, :menu => windowmenu, :label => 'Window'
```

> *As of this writing, RubyTk hadn't yet added the TkSysMenu_Window helper.*

```
$windowmenu = Tkx::widget->new(Tkx::menu($m->_mpath . ".window"));
$m->add_cascade(-menu => $windowmenu);
```

```
windowmenu = Menu(menubar, name='window')
menubar.add_cascade(menu=windowmenu, label='Window')
```

Other Menu Handlers

You saw previously how handling certain menu items required you to define Tcl callback procedures, in particular for displaying the preferences dialog ('tk::mac::ShowPreferences') and displaying help ('tk::mac::ShowHelp').

There are several other callbacks that you can define, for example, to intercept the Quit menu item to prompt to save changes or to be informed when the application is hidden or shown. Here is the complete list:

`tk::mac::ShowPreferences`	Called when the "Preferences..." menu item is selected.
`tk::mac::ShowHelp`	Called to display main online help for the application.
`tk::mac::Quit`	Called when the Quit menu item is selected, when the user is trying to shut down the system etc.
`tk::mac::OnHide`	Called when your application has been hidden.
`tk::mac::OnShow`	Called when your application is shown after being hidden.
`tk::mac::OpenApplication`	Called when your application is first opened.
`tk::mac::ReopenApplication`	Called when the user "reopens" your already-running application (e.g. clicks on it in the Dock)
`tk::mac::OpenDocument`	Called when the Finder wants the application to open one or more documents (e.g. that were dropped on it). The procedure is passed a list of pathnames of files to be opened.
`tk::mac::PrintDocument`	As with OpenDocument, but the documents should be printed rather than opened.

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"`.*

```
sysmenu = TkSysMenu_System.new(menubar)
menubar.add :cascade, :menu => sysmenu
```

```
$system = Tkx::widget->new(Tkx::menu($m->_mpath . ".system"));
$m->add_cascade(-menu => $system);
```

> *The pathname of the menu must be explicitly provided, in this case with the name **".system"**.*

```
sysmenu = Menu(menubar, name='system')
menubar.add_cascade(menu=sysmenu)
```

> *While normally Tkinter will choose a widget path name for us, here we've had to explicitly provide one with the name 'system'; this is the cue that Tk needs to recognize it as the system menu.*

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"`.*

```
help = TkSysMenu_Help.new(menubar)
menubar.add :cascade, :menu => help, :label => 'Help'
```

```
$help = Tkx::widget->new(Tkx::menu($m->_mpath . ".help"));
$m->add_cascade(-menu => $help);
```

> *The pathname of the menu must be explicitly provided, in this case with the name **".help"**.*

```
menu_help = Menu(menubar, name='help')
menubar.add_cascade(menu=menu_help, label='Help')
```

> *The widget pathname of the menu must be explicitly provided, in this case with the name **"help."** This can be specified for any Tkinter widget using the 'name' option when creating the widget.*

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 macOS, 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, macOS 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 pop up 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"
}
```

```
require 'tk'
root = TkRoot.new
menu = TkMenu.new(root)
%w(One Two Three).each {|i| menu.add :command, :label => i}
if Tk.windowingsystem == 'aqua'
    root.bind '2', proc{|x,y| menu.popup(x,y)}, "%X %Y"
    root.bind 'Control-1', proc{|x,y| menu.popup(x,y)}, "%X %Y"
else
    root.bind '3', proc{|x,y| menu.popup(x,y)}, "%X %Y"
end
Tk.mainloop
```

```
use Tkx;
my $mw = Tkx::widget->new(".");
my $menu = $mw->new_menu();
foreach ("One", "Two", "Three") {$menu->add_command(-label => $_);}
if (Tkx::tk_windowingsystem() eq "aqua") {
    $mw->g_bind("<2>", [sub {my($x,$y) = @_; $menu->g_tk___popup($x,$y)}, Tkx::Ev("%X", "%Y")] );
    $mw->g_bind("<Control-1>", [sub {my($x,$y) = @_; $menu->g_tk___popup($x,$y)}, Tkx::Ev("%X", "%Y")]);
} else {
    $mw->g_bind("<3>", [sub {my($x,$y) = @_; $menu->g_tk___popup($x,$y)}, Tkx::Ev("%X", "%Y")]);
}
Tkx::MainLoop();
```

```
from tkinter import *
root = Tk()
menu = Menu(root)
for i in ('One', 'Two', 'Three'):
    menu.add_command(label=i)
if (root.tk.call('tk', 'windowingsystem')=='aqua'):
    root.bind('<2>', lambda e: menu.post(e.x_root, e.y_root))
    root.bind('<Control-1>', lambda e: menu.post(e.x_root, e.y_root))
else:
    root.bind('<3>', lambda e: menu.post(e.x_root, e.y_root))
root.mainloop()
```

Previous: More Widgets (morewidgets.html) | Contents (index.html) | Single Page (onepage.html) | Next: Windows and Dialogs (windows.html)