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: Windows and Dialogs | Contents | Single Page | Next: Fonts, Colors, Images

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

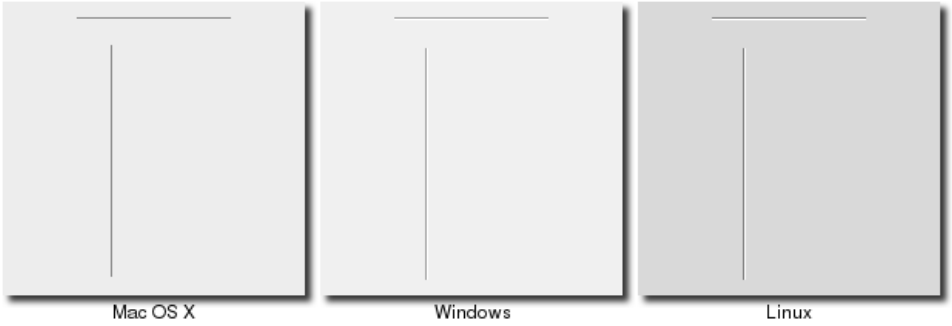
If you do need to display a large number of widgets onscreen at the same time, you have to think about how to organize them visually. You've seen how `grid` can help by making it easy to align widgets with each other. White space is another useful aid. Placing related widgets quite close to each other (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.

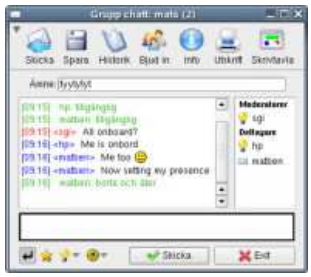
> *The amount of white space around different widgets, between groups of widgets, around borders and so on is highly platform specific. While you can do an adequate job without worrying about exact pixel numbers, if you want a highly polished user interface, you'll need to tune this for each platform.*

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
```

Separators are created using the **`Tk::Tile::Separator`** class:

```
s = Tk::Tile::Separator.new(parent) { orient 'horizontal' }
```

Separators are created using the **`new_ttk__separator`** method, a.k.a. **`Tkx::ttk__separator`**:

```
$sep = $parent->new_ttk__separator(-orient => 'horizontal');
```

Separators are created using the **`ttk.Separator`** function:

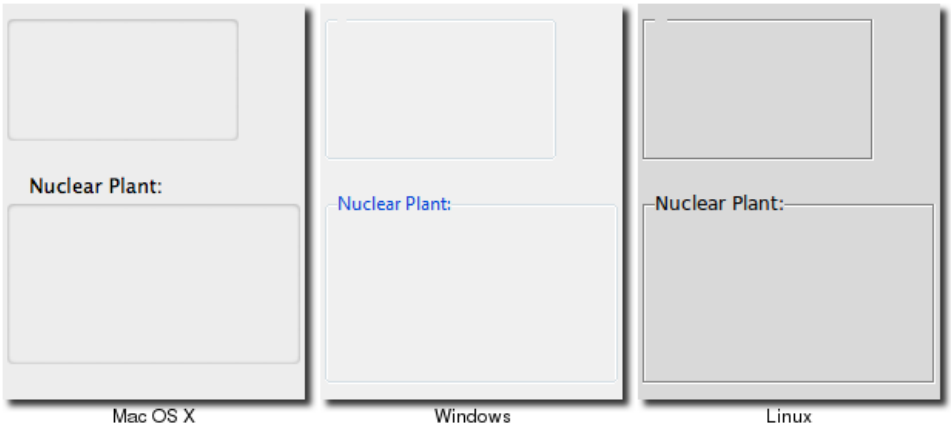
```
s = ttk.Separator(parent, orient=HORIZONTAL)
```

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

Label Frames

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.

- Widget Roundup
- Reference Manual



Labelframe Widgets

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

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

Labelframes are created using the **`Tk::Tile::Labelframe`** class:

```
lf = Tk::Tile::Labelframe.new(parent) { text 'Label' }
```

Labelframes are created using the **`new_ttk__labelframe`** method, a.k.a. **`Tkx::ttk__labelframe`**:

```
$lf = $parent->new_ttk__labelframe(-text => "Label");
```

Labelframes are created using the **`ttk.Labelframe`** function:

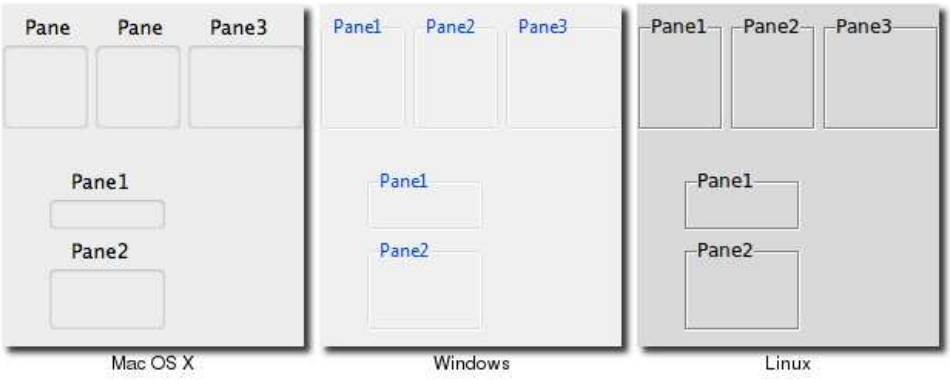
```
lf = ttk.Labelframe(parent, text='Label')
```

Paned Windows

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

- Widget Roundup
- Reference Manual

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.



Panedwindow Widgets (Shown 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
```

Panedwindows are created using the **`Tk::Tile::Paned`** class:

```
p = Tk::Tile::Paned.new(parent) { orient 'vertical' }
# first pane, which would get widgets gridded into it:
f1 = Tk::Tile::Labelframe.new(p) {text 'Pane1'; width 100; height 100;}
f2 = Tk::Tile::Labelframe.new(p) {text 'Pane2'; width 100; height 100;}; # second pane
p.add f1, nil
p.add f2, nil
```

> *The extra "nil" parameter to add can be replaced with a hash of pane-specific options, which usually aren't needed.*

Panedwindows are created using the **`new_ttk__panedwindow`** method, a.k.a. **`Tkx::ttk__panedwindow`**:

```
$p = $mw->new_ttk__panedwindow(-orient => 'vertical');
# first pane, which would get widgets gridded into it:
$f1 = $p->new_ttk__labelframe(-text => "Panel", -width => 100, -height => 100);
$f2 = $p->new_ttk__labelframe(-text => "Pane2", -width => 100, -height => 100); # second pane
$p->add($f1);
$p->add($f2);
```

Panedwindows are created using the **`ttk.Panedwindow`** function:

```
p = ttk.Panedwindow(parent, orient=VERTICAL)
# first pane, which would get widgets gridded into it:
f1 = ttk.Labelframe(p, text='Pane1', width=100, height=100)
f2 = ttk.Labelframe(p, text='Pane2', width=100, height=100)    # second pane
p.add(f1)
p.add(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.

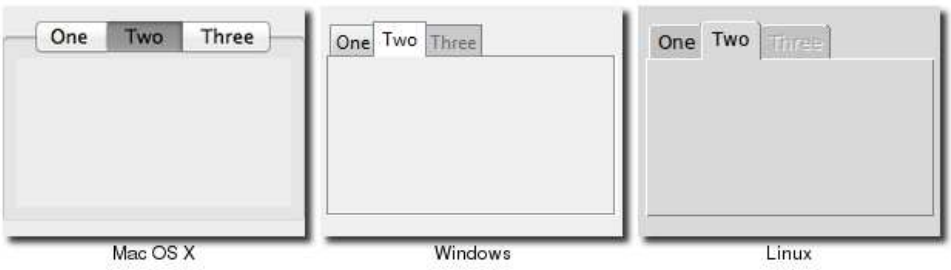
Calling the `"add"` method will add a new pane at the end of the list of panes. The "`insert` *`position subwindow`*" method allows you to place the pane at the given position in the list of panes (0..n-1); if the pane is already managed by the panedwindow, it will be moved to the new position. You can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

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.



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"
```

Notebooks are created using the **`Tk::Tile::Notebook`** class:

```
n = Tk::Tile::Notebook.new(parent)
f1 = Tk::Tile::Frame.new(n); # first page, which would get widgets gridded into it
f2 = Tk::Tile::Frame.new(n); # second page
n.add f1, :text => 'One'
n.add f2, :text => 'Two'
```

Notebooks are created using the **`new_ttk__notebook`** method, a.k.a. **`Tkx::ttk__notebook`**:

```
$n = $mw->new_ttk__notebook;
$f1 = $n->new_ttk__frame; # first page, which would get widgets gridded into it
$f2 = $n->new_ttk__frame; # second page
$n->add($f1, -text => "One");
$n->add($f2, -text => "Two");
```

Notebooks are created using the **`ttk.Notebook`** function:

```
n = ttk.Notebook(parent)
f1 = ttk.Frame(n)   # first page, which would get widgets gridded into it
f2 = ttk.Frame(n)   # second page
n.add(f1, text='One')
n.add(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.

To retrieve the subwindow that is currently selected, call the "`selected`" method, and change the selected tab by calling the "`select`" method, 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 "`itemconfigure` *`tabid, :option => value`*" method (where "`tabid`" is again the tab's position or subwindow); use the "`itemcget` *`tabid, :option`*" to return the current value of the option.

> *The `itemconfigure` and `itemcget` methods exist alongside the `tabconfigure` and `tabcget` methods, which are more true to the API; however the `tabcget` method currently has a bug in it.*

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.

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.

Previous: Windows and Dialogs | Contents | Single Page | Next: Fonts, Colors, Images