

ptk: Parenthesized tk

Pascal Raymond

2020-04-01

Contents

1	ptk overview	2
1.1	Containers	2
1.2	Leaves	2
1.3	Hello world	2
1.4	Customization and components	3
2	Basic items	3
2.1	Basic leaves	3
2.2	Packers	3
2.2.1	Line/column layout	3
2.2.2	Example:	4
2.2.3	Packers summary:	4
2.2.4	Packing customization	4
2.2.5	Conditional display	5
3	Meta-containers	5
3.1	Switch and toggle	5
3.2	Popup	5
4	Meta-leaves	6
4.1	Show var	6
4.2	Combobox	6
5	Store and reuse	6
5.1	Principle	6
5.2	Reopen container	6
5.3	Reconfigure	6
6	Appendix: Combobox (Bryan Oakley)	6
6.1	STANDARD OPTIONS	6
6.2	WIDGET-SPECIFIC OPTIONS	7

1 ptk overview

Ptk (parenthesized tk) is a library to ease the programming of tcltk-based gui's. The main goal is to provide a programming style that reflects the *logical* layout of the the gui, and hides/abstracts the technical details. - avoid the manipulation widget paths; - write things once; in particular creation and packing are not two different features. Concretely, a gui consists in a recursive structure of horizontal and vertical frames containing basic widgets (leaves).

1.1 Containers

Each container has a *kind* that determines how nested items will be displayed and managed. Containers are created and finalized using `begin` and `end`. Every ptk items appearing within `begin` and `end` belongs to the container.

```
ptk::begin kind ?params? ?opts?
```

```
    ....
```

```
ptk::end ?kind?
```

- `kind` is the type of container; basic kinds are `main` (top-level), `line`, `col` etc.
- `params` is the list mandatory parameters (passed by position); their number and types depend on `kind`
- `opts` is a list of classical tk options `-key value`; valid options depends on `kind`.
- `kind` is optional for command `end`, but recommended to tract nesting errors.

1.2 Leaves

Each leaf has a *kind* that determines its layout and behavior. Ptk *inherits* classical tk widgets (`label`, `button`, `checkbutton` etc.), and provides some other meta-widgets (`combobox`, `select_file`, etc.). Leaves are created by the `add` command, whose arguments are similar to those of `begin`:

```
ptk::add kind ?params? ?opts?
```

1.3 Hello world

```
package require Ptk
namespace import ptk::*

begin main "My program"
    begin line
        button -text "push me" -command {puts "Hello world!"}
    end line
end main
```

1.4 Customization and components

Ptk items are made of tk-widget that can be customized using `-key values` options. Each item has a *main* component, and possibly some others. When creating an item (with `add` or `begin`) options are directly passed to this main component. It is also possible to pass options *just after* the creation using the `configure` command. Note that this command is necessary to customize other component than the main one.

```
ptk::configure ?component? ?opts?
```

- `configure` applies implicitly to the last created item, and then must follow a `add` or `begin` command.
- `component` identifies the widget to customize *inside the current item*; valid components depend on item kind; if omitted, `configure` applies to the main component.

Note: it is possible to customize *distant* items (not the last created one) using the advanced store and reuse feature.

2 Basic items

2.1 Basic leaves

Basic leaves are just *wrappers* around basic tk widgets, and inherit their options:

```
ptk::label ?label-opts?  
ptk::button ?button-opts?  
ptk::checkboxbutton ?checkboxbutton-opts?  
ptk::radiobutton ?radiobutton-opts?  
ptk::scale ?scale-opts?  
ptk::entry ?entry-opts?
```

Customization can be performed after `add`; valid options are the same than for `add`.

```
ptk::configure ?opts?
```

2.2 Packers

2.2.1 Line/column layout

Basic containers are *packers*. They allow to organize the layout as a structure of nested horizontal and vertical frames.

- horizontal packers: sons are inserted from left to right; basic ones are `line` and `hbar`.
- vertical packers: sons are inserted from top to bottom; basic ones `col` and `vbar`.

Each container has a predefined packing policy that defines how the container behaves within its own container, and how it handles its content:

- `line` and `col` expand in both direction within their container, and expand their content if both direction too.
- `hbar` and `vbar` expand in one direction only (resp. `x` and `y`), and does not expand their content but *stack* them side by side (resp. left to right, top to bottom).

2.2.2 Example:

Here is a typical layout with a top hbar (e.g., for filling with menu entries), a left vbar (e.g. tool bar), and a central expandable zone to put the rest.

```
begin main "My program"
  begin hbar -text "Menu bar"
    .....
  end hbar
  begin line
    begin vbar -text "Tool bar"
      .....
    end vbar
    begin col -relief sunken
      .....
    end col
  end line
end main
```

2.2.3 Packers summary:

```
ptk::begin col|line|vbar|hbar ?opts?
...
ptk::end ?col|line|vbar|hbar?
```

- packers have no mandatory parameters
- valid options are any valid tk *frame* or *labelframe* options:
 - if `-text <string>` is specified, the packer is interpreted as a *labelframe*
 - otherwise as a simple *frame*

Customization can be performed after `begin`, valid options are *tk-frame* options.

```
ptk::configure ?opts?
```

2.2.4 Packing customization

Do not mix up with *widget* customization. The `packreq` command passes specific packing request from a item to its contained.

```
ptk::packreq ?pack-opts?
```

- `pack-opts` are packing option (cf. tk command `pack configure`)
- useful in particular to bypass the default packing policy, e.g. `-expand 1` inside a `vbar/hbar` or `-expand 0` inside a `line/col`
- useful options are:
 - `-anchor anchor`
 - `-expand boolean`
 - `-fill style`
 - `-ipadx amount`
 - `-ipady amount`
 - `-padx amount`

- `-pady amount`
- **NEVER USE OTHER OPTIONS** since they interfere with ptk algorithm and will lead to undefined behaviors.

2.2.5 Conditional display

TODO showif

3 Meta-containers

3.1 Switch and toggle

`switch` displays exactly one from its n sons one according to the value of an enumerate variable.

```
ptk::begin switch var values ?opts?
    <son 1>
    ...
    <son n>
ptk::end ?switch?
```

- `values` is a list of n string values ($v1 \dots vn$)
- `var` is a variable taking its value in `values`
- must contain exactly n sons, the i th son is displayed iff `var` holds value vi
- valid `opts` are *tk-frame* options

`toggle` is basically a switch commanded by a Boolean variable.

```
ptk::add toggle var ?opts?
    <son 1>
    <son 0>
ptk::add ?toggle?
```

- `var` is a variable containing Boolean values
- must contain exactly 2 sons, first son is displayed if `var` is true, second son if `var` is false.
- valid `opts` are *tk-frame* options

Customization: can be performed after begin:

```
ptk::configure ?opts?
```

3.2 Popup

`popup` displays a button which, when pressed opens a toplevel (independent) window displaying its content plus a predefined *dismiss* button.

```
ptk::begin popup title ?opts?
ptk::end ?popup?
```

- `title` is the name of the opened window; and the default text of the popup button.

- *opts* are *tk-button* options, in particular one can use *-text* to change button label.

Customization: *popup* has two configurable tk components, *button* (main one) and *frame* (main frame of the popup window):

```
ptk::configure ?button-opts?  
ptk::configure button ?button-opts?  
ptk::configure frame ?frame-opts?
```

4 Meta-leaves

4.1 Show var

{kind outer params components} {showvar packer {var} {labelframe label}}

4.2 Combobox

wrapper for Bryan Oakley's combobox *opts* -> passed to combobox

{kind outer params components} {combobox packer {title var values} {combobox}}

5 Store and reuse

5.1 Principle

store

5.2 Reopen container

reopen reclose

5.3 Reconfigure

reconfigure

6 Appendix: Combobox (Bryan Oakley)

This section is extracted from Bryan Oakley's Combobox manual.

6.1 STANDARD OPTIONS

**-background -borderwidth -cursor -font -foreground -height -highlightthickness
-highlightbackground -maxheight -relief -selectbackground -selectborderwidth -
selectforeground -state -textvariable**

See the *options* manual entry for detailed descriptions of the above options.

6.2 WIDGET-SPECIFIC OPTIONS

-command

Defines a command to be run whenever the value of the combobox changes. The command will have two values appended to it: the name of the window and the new value of the combobox.

-commandstate

One of “normal” or “disabled”. If set to “disabled”, the value of the **-command** option will not be evaluated when the value of the combobox changes.

-editable

A boolean value which specifies whether the entry widget of the combobox can be typed in. If false, values may only be set by selecting them from the dropdown list.

-height

Specifies the height of the dropdown listbox, in number of lines. A value of zero will make the dropdown list just tall enough to hold all of the elements in the list. The height defaults to 10.

-image

Defines an image to use on the button used to drop down the list. If it is not specified it defaults to a small black triangle.

-maxheight

Sets the maximum height of the dropdown listbox, in the event **-height** is set to zero and there are a large number of items in the list. If this value is set to 0 (zero) the listbox will be as large as the total number of items in the listbox. The maxheight defaults to 10.

-value

Specifies the value for the combobox.

-width

Specifies an integer value indicating the desired width of the combobox entry widget, in average size characters of the widget's font. If the value is less than or equal to zero, the widget picks a size just large enough to hold its current text.