Creating Graphical User Interfaces (GUIs) using the R tcltk package

Tk and its geometry managers

Adrian Waddell

University of Waterloo
Departement of Statistics and Actuarial Science

January 13, 2011

Outline

These slides were written on behalf of the Departement of Statistics and Actuarial Science at the University of Waterloo, Canada.

At the time of writing, the current software versions are

- Tcl 8.5
- Tk 8.5

You find an R source code file for these slides and more slides on our webpage.

My subjective good practice hints will be highlighted with this symbol: 

Geometry Managers

Widgets are GUI elements like buttons, scrollbars, text areas etc.

Geometry managers control the layout of your widgets on your screen based on rules and constraints.

To understand why geometry managers are so important, open your email client of choice and resize the window a few times and observe how all the graphical elements of your client change. In Thunderbird, for example, when decreasing the window width, the Subject column disappears first, then the From and Date columns and finally the inbox folder tree. You can imagine yourself many other –possible worse– ways Thunderbird could have changed its layout when decreasing the window width.

Tk offers several different geometry managers: `pack` and `grid` are the most popular ones among others like `place` and `table`. For now, we only discuss the `pack` geometry manager.

The Pack Geometry Manager

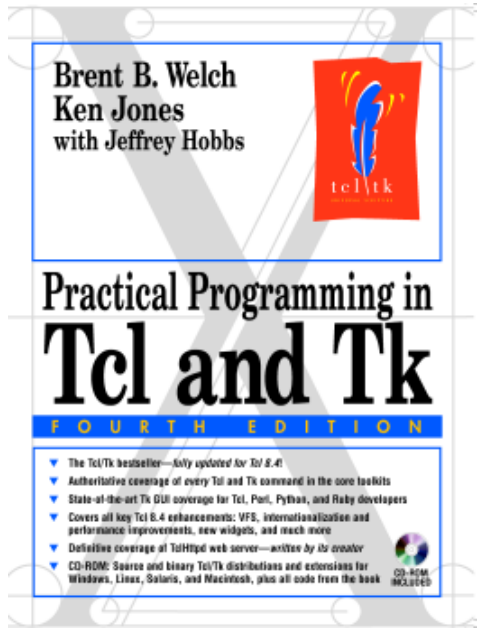
The `pack` geometry manager is constrained based and provides a very powerful way to layout your GUI.

In a nutshell, widgets have one parent and possibly several children. You define how each child is oriented relative to its parent with statements like `side='left'` and `anchor='w'` etc. A child can be a parent to new children.

We devote a whole section to this because only good understanding of how the `pack` geometry manager works allows you to build GUIs which behave as you expect.

Content from these slides

These slides cover chapter 12 (page 115) from Brent B. Welch's book Practical Programming in Tcl an Tk, **1st edition**. Most of the examples are –with permission from Brent B. Welch– almost one to one transcoded from Tcl code to R code. Often explanations are also copied one-to-one from this book.



I highly recommend buying the latest edition of this book to get a more comprehensive overview of geometry managers and the whole Tcl and Tk programming language.

Differences between Tcl and R

The child parent relationship discussed in the last slide is enforced in Tcl by syntax. Every widget is part of a *window path name* where parent and child are separated by . (dot). The toplevel window itself is called . (dot). So a widget path name might be `.a.b.c` where `c` is a child of `b` is a child of `a` is a child of . (dot), the toplevel window.

Depending on the widget type of `a`, `b` and `c`, they might have multiple children. So say, `a` and `b` are frame widgets, then possible *path names* for other widgets would be `.a.b.d`, `.a.e`, `.f`.

Multiple toplevel windows can be created with the `toplevel` Tcl command and the `tktoplevel` function in R.

R however does not employ this *path name* model so obviously. That is, if you create a new widget you always have to state its parent widget but you are not forced to state the whole path.

More on Differences between Tcl and R

A look at the *path name* model in detail: in Tcl the following code

```
package require Tk
frame .f
button .f.bok -text "OK"
button .f.bcancel -text "Cancel"
```

would create a frame within the toplevel window, and within the frame `.f` two buttons. Note these widgets wont show yet on your screen.

The equivalent R code would be

```
library(tcltk)
tt <- tktoplevel()
f <- tkframe(tt)
bok <- tkbutton(f,text="OK")
bcancel <- tkbutton(f, text="Cancel")
```

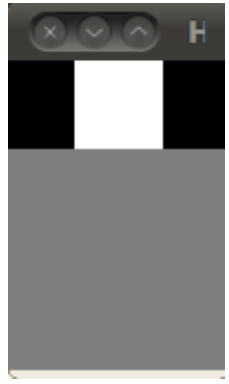
Behind the scenes however, R follows the *path name* model with numeric names. If you were to look at the return value of `bok` at your R promt, you would get

```
> bok
$ID = ".1.1"            # and some other output
```

Packing Toward a Side

The following code creates two frames and packs them toward the top of the toplevel window.

```
tt <- tktoplevel(bg = "black")
one <- tkframe(tt, width = 40, height = 40, bg = "white")
two <- tkframe(tt, width = 100, height = 100, bg = "grey50")
tkpack(one, two, side = "top")
```



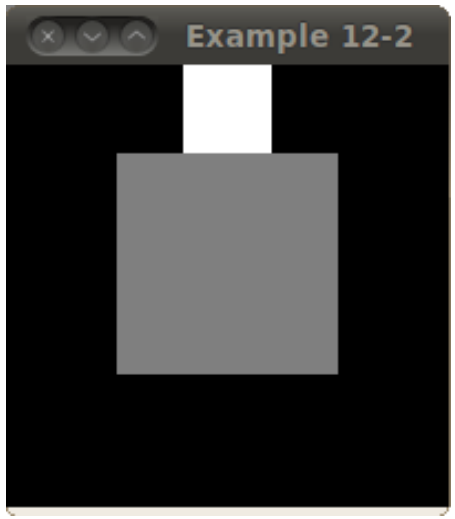
Possible sides to pack a widget to are: `top`, `right`, `bottom` and `left`.

Notice that the main window re-sizes to the minimal possible size such that all widgets (children) fit into the window.

Shrinking Frames and Pack Propagate

If you want the geometry manager to leave the main window at a certain size, rather than shrinking to a size to fit all its children, you can turn the size propagation as follows:

```
one <- tkframe(tt, width = 40, height = 40, bg = "white")
two <- tkframe(tt, width = 100, height = 100, bg = "grey50")
tkpack.propagate(tt, FALSE) ## Window wont resize
tkpack(one, two, side = "top")
```



Note that as from now on, I omit to display the initialization of the toplevel window `tt`. You can download the complete R code for all examples here.

Horizontal and Vertical Stacking

Use frames if you want stack your widgets in different orientations. Within a frame either use horizontal or vertical stacking.

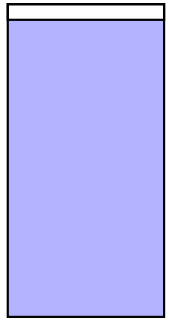
```
one <- tkframe(tt, bg = "white")
two <- tkframe(tt, width = 100, height = 50, bg = "grey50")
for(b in c("alpha", "beta", "gamma")) {
  assign(b, tkbutton(one, text = b))
  tkpack(get(b), side = "left")
}
tkpack(one, two, side = "top")
```

The Cavity Model

The packing algorithm of the pack geometry manager is based on the *cavity model*. Widgets are placed into *cavities*, and the primary rule is

> *a widget occupies one whole side of the cavity*

In the following sequence of pictures, the cavity is colored in light blue



- The initial toplevel window provides one big cavity

```
tt <- tktoplevel(bg="white", width=100, height=190)
tkpack.propagate(tt,FALSE)
```

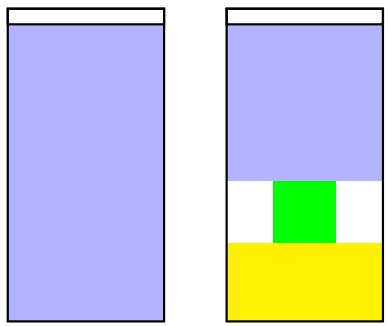
propagation is turned of for the sake visualization here.

The Cavity Model

The packing algorithm of the pack geometry manager is based on the *cavity model*. Widgets are placed into *cavities*, and the primary rule is

> *a widget occupies one whole side of the cavity*

In the following sequence of pictures, the cavity is colored in light blue



- Packing towards the bottom side of the cavity

```
one <- tkframe(tt, width = 100, height = 50, bg = "yellow")
two <- tkframe(tt, width = 40, height = 40, bg = "green")
tkpack(one, two, side = "bottom")
```

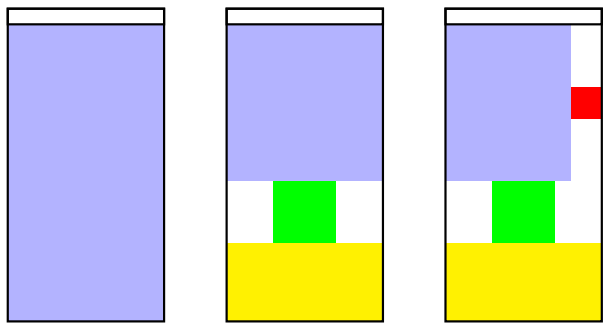
Note how the green square gets centered horizontally.

The Cavity Model

The packing algorithm of the pack geometry manager is based on the *cavity model*. Widgets are placed into *cavities*, and the primary rule is

> *a widget occupies one whole side of the cavity*

In the following sequence of pictures, the cavity is colored in light blue



- Packing towards the right side of the cavity

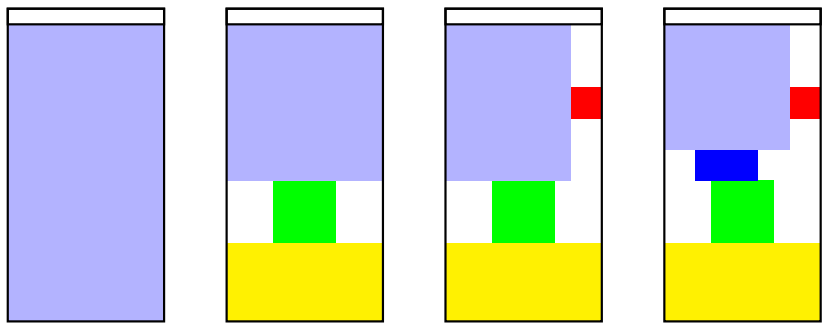
```
three <- tkframe(tt, width = 20, height = 20, bg = "red")
tkpack(three, side = "right")
```

The Cavity Model

The packing algorithm of the pack geometry manager is based on the *cavity model*. Widgets are placed into *cavities*, and the primary rule is

> *a widget occupies one whole side of the cavity*

In the following sequence of pictures, the cavity is colored in light blue



- Note how the blue rectangle is centered horizontally.

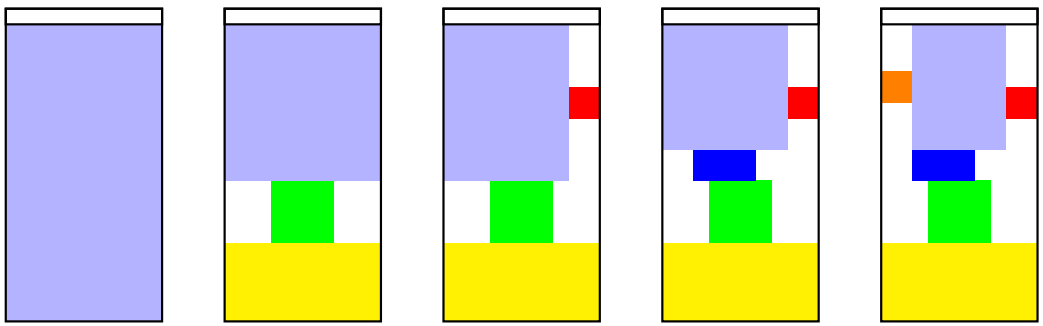
```
four <- tkframe(tt, width = 40, height = 20, bg = "blue")
tkpack(four, side = "bottom")
```

The Cavity Model

The packing algorithm of the pack geometry manager is based on the *cavity model*. Widgets are placed into *cavities*, and the primary rule is

> *a widget occupies one whole side of the cavity*

In the following sequence of pictures, the cavity is colored in light blue



- Note that each `frame` can be used as a parent for new widgets and hence provides a new cavity inside the frame itself.

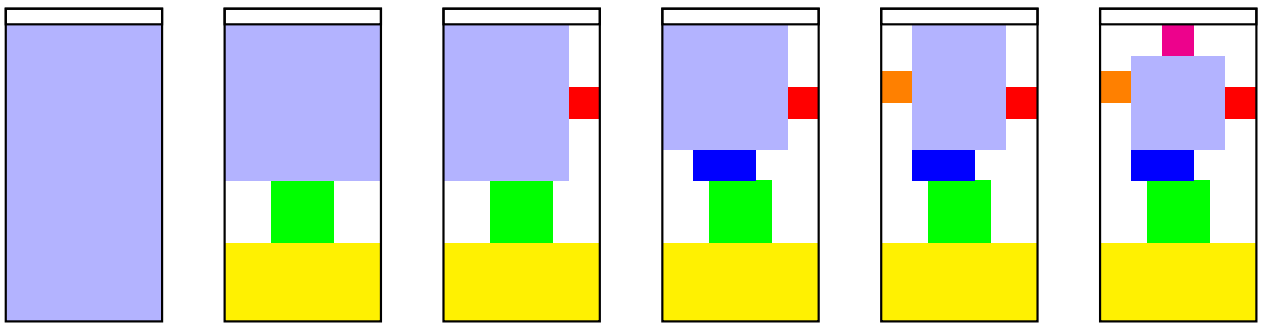
```
five <- tkframe(tt, width = 20, height = 20, bg = "orange")
tkpack(five, side = "left")
```

The Cavity Model

The packing algorithm of the pack geometry manager is based on the *cavity model*. Widgets are placed into *cavities*, and the primary rule is

> *a widget occupies one whole side of the cavity*

In the following sequence of pictures, the cavity is colored in light blue



- By now, the cavity model should be clear to you.

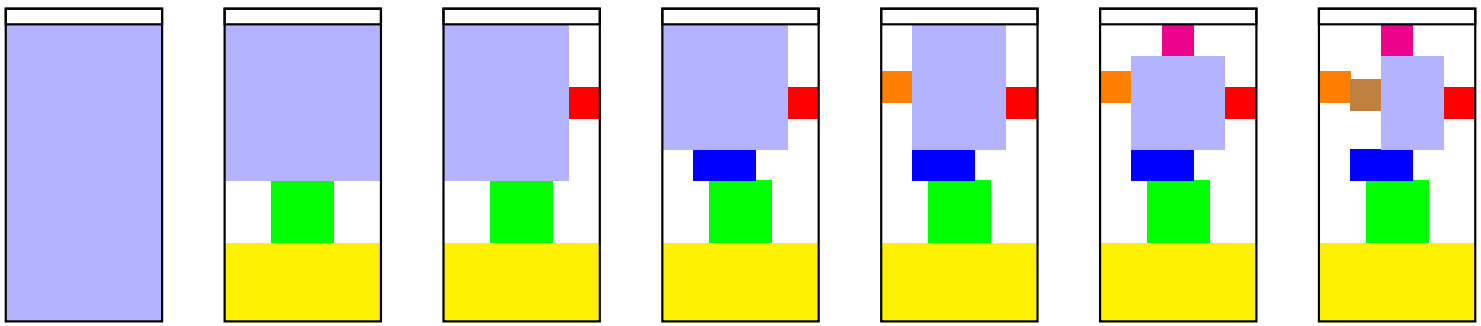
```
six <- tkframe(tt, width = 20, height = 20, bg = "magenta")
tkpack(six, side = "top")
```

The Cavity Model

The packing algorithm of the pack geometry manager is based on the *cavity model*. Widgets are placed into *cavities*, and the primary rule is

a widget occupies one whole side of the cavity

In the following sequence of pictures, the cavity is colored in light blue



- The complete R-code for this example can be found on our webpage.

```
seven <- tkframe(tt, width = 20, height = 20, bg = "brown")
tkpack(seven, side = "left")
```

Try to resize the window and figure out what the pack-geometry manager does.

Packing Space and Display Space

- The packer distinguishes between *packing space* and *display space*.
  - display space: area of requested by a widget for painting itself
  - packing space: area the packer allows for the placement of the widget
- Because of geometry constraints, a widget may be allocated more or less packing space than it needs to display itself
- Extra space, if any, is along the side of the cavity against which the widget was packed

For our previous example, the display space is the 40 times 40 pixels (in green) and the packing space is the gray rectangle (partly covered by the display space of the widget):

The -fill Option

The `fill` packing option causes the widget to fill up the allocated packing space with its display. A widget can fill in the X or Y direction, or both. The default is not to fill, which is why the black background has shown through in some of the examples so far:

```
tt <- tktoplevel(bg = "black")
one <- tkframe(tt, width = 100, height = 50, bg = "grey50")
two <- tkframe(tt, width = 40, height = 40, bg = "white")
tkpack(one, two, side = "bottom", fill = "x")
three <- tkframe(tt, width = 20, height = 20, bg = "red")
tkpack(three, side = "right", fill = "x")
```

The `three` frame does not fill, because the fill does not expand into the packing cavity.

Internal Padding with -ipadx and -ipady

Another way to get more fill space is with the `-ipadx` and `-ipady` packing options. Due to other constraints such a request might not be offered.

```
menubar <- tkframe(tt, bg = "white")
body <- tkframe(tt, width = 150, height = 50, bg = "grey50")
for(b in c("alpha", "beta")){
  assign(b,tkbutton(menubar, text = b))
}
tkpack(alpha, side = "left", ipady = 10)
tkpack(beta, side = "right", ipadx = 10)
tkpack(menubar, side = "top", fill = "x", ipady = 5)
tkpack(body)
```

Note that the white frame and the two buttons have internal padding.

Button padding vs. packer padding

Buttons have their own `-padx` and `-pady` options that give them more display space, too. This padding provided by the button is button is used to keep its text away from the edge of the button.

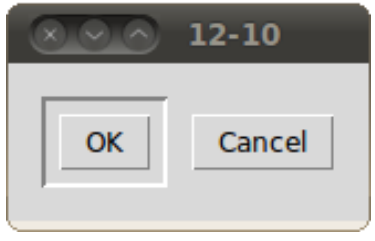
```
foo <- tkbutton(tt, text="Foo", anchor="e", padx=0, pady=0)
tkpack(foo, side = "right", ipadx = 10, ipady = 10)
bar <- tkbutton(tt, text="Bar", anchor="e", pady=10, padx=10)
tkpack(bar, side = "right", ipadx = 0, ipady = 0)
```

Note that the the text orientation of the `Foo` button is set with `anchor = "e"` option.

External Padding with -padx and -pady

The packer can provide external padding that allocates packing space that cannot be filled. The space is outside of the border that widgets use to implement their 3D reliefs.

```
tkconfigure(tt, borderwidth = 10)
ok <- tkframe(tt, borderwidth = 2, relief = "sunken")
ok.b <- tkbutton(ok, text = "OK")
tkpack(ok.b, padx = 5, pady = 5)
cancel <- tkbutton(tt, text = "Cancel")
tkpack(ok, cancel, side = "left", padx = 5, pady = 5)
```

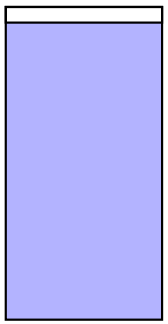


Even if the `ok.b` button were packed with `-fill both`, it would look the same. The external padding provided by a packer will not be filled by the child widgets.

Expand and resizing

The `expand=TRUE` packing option lets a widget expand its packing space into unclaimed space in the packing cavity. This is in contrast to the `fill` option which only expands into non-cavity regions. Most often, you will use the `expand` option when you want to resize a window. In nearly all cases, `fill='both'` is used along with `expand=TRUE`.

As in the previous slide, the cavity is colored light blue.



- The initial toplevel window provides one big cavity

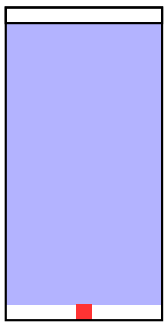
```
tt <- tktoplevel(bg="white", width=100, height=190)
tkpack.propagate(tt,FALSE)
frame1 <- tkframe(tt, bg = 'red')
```

propagation is turned of for the sake visualization here.

Expand and resizing

The `expand=TRUE` packing option lets a widget expand its packing space into unclaimed space in the packing cavity. This is in contrast to the `fill` option which only expands into non-cavity regions. Most often, you will use the `expand` option when you want to resize a window. In nearly all cases, `fill='both'` is used along with `expand=TRUE`.

As in the previous slide, the cavity is colored light blue.



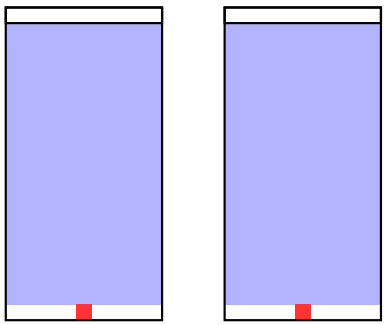
- No `fill` option will reduce the widget to its minimal size

```
tkpack(frame1, side = 'bottom')
```

Expand and resizing

The `expand=TRUE` packing option lets a widget expand its packing space into unclaimed space in the packing cavity. This is in contrast to the `fill` option which only expands into non-cavity regions. Most often, you will use the `expand` option when you want to resize a window. In nearly all cases, `fill='both'` is used along with `expand=TRUE`.

As in the previous slide, the cavity is colored light blue.



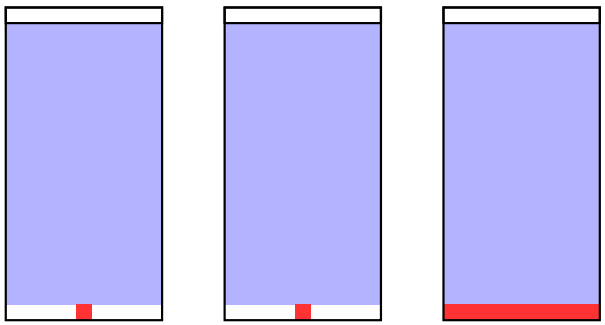
- The `fill` option does not expand the widget into the cavity.

```
tkpack(frame1, fill = 'y', side = 'bottom')
```

Expand and resizing

The `expand=TRUE` packing option lets a widget expand its packing space into unclaimed space in the packing cavity. This is in contrast to the `fill` option which only expands into non-cavity regions. Most often, you will use the `expand` option when you want to resize a window. In nearly all cases, `fill='both'` is used along with `expand=TRUE`.

As in the previous slide, the cavity is colored light blue.



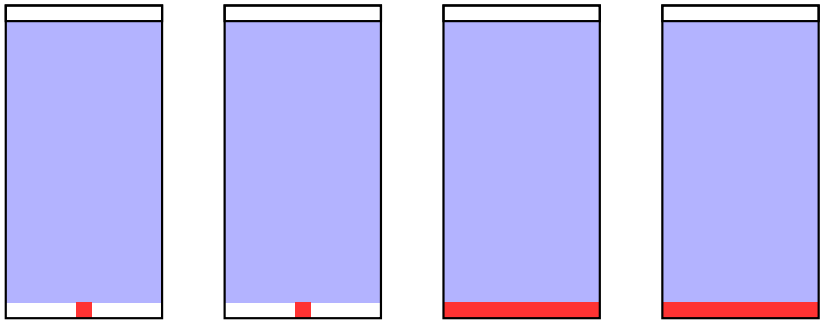
- Here however, the `fill` makes the packer to expand into unclaimed space since it is not in the cavity direction.

```
tkpack(frame1, fill = 'x', side = 'bottom')
```

Expand and resizing

The `expand=TRUE` packing option lets a widget expand its packing space into unclaimed space in the packing cavity. This is in contrast to the `fill` option which only expands into non-cavity regions. Most often, you will use the `expand` option when you want to resize a window. In nearly all cases, `fill='both'` is used along with `expand=TRUE`.

As in the previous slide, the cavity is colored light blue.



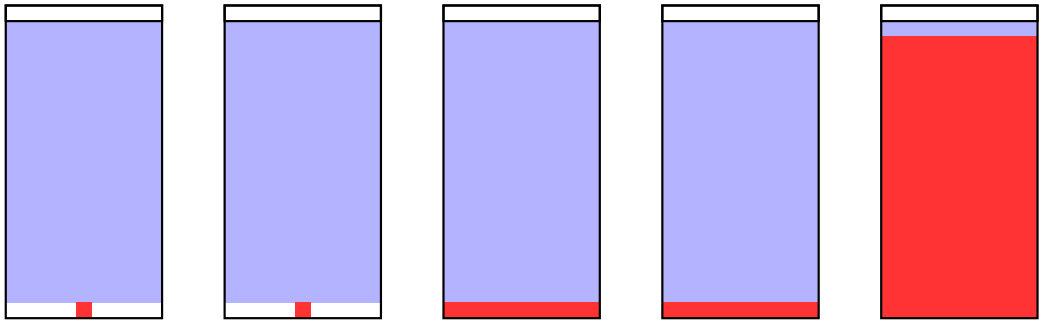
- Again, the fill option only expands only in to free space which does not face towards the cavity.

```
tkpack(frame1, fill = 'both', side = 'bottom')
```

Expand and resizing

The `expand=TRUE` packing option lets a widget expand its packing space into unclaimed space in the packing cavity. This is in contrast to the `fill` option which only expands into non-cavity regions. Most often, you will use the `expand` option when you want to resize a window. In nearly all cases, `fill='both'` is used along with `expand=TRUE`.

As in the previous slide, the cavity is colored light blue.



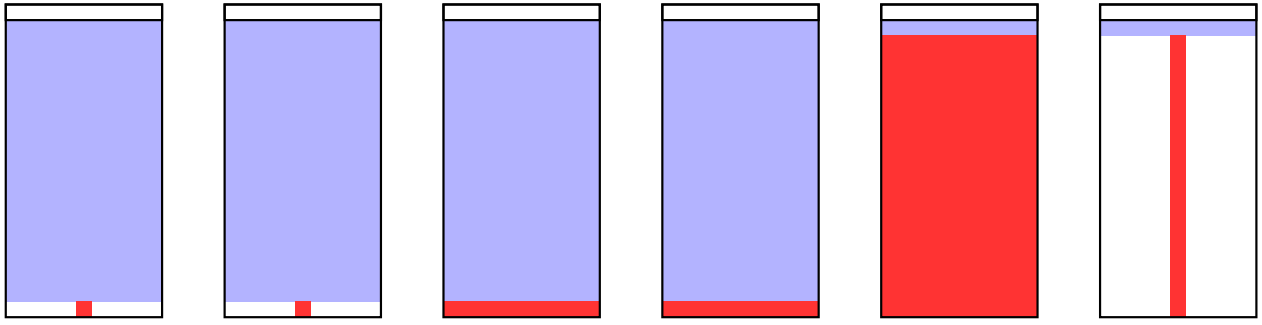
- With both options, `fill='both'` and `expand=TRUE`, the widget expands as much as possible.

```
tkpack(frame1, fill = 'both', expand = TRUE, side='bottom')
```

Expand and resizing

The `expand=TRUE` packing option lets a widget expand its packing space into unclaimed space in the packing cavity. This is in contrast to the `fill` option which only expands into non-cavity regions. Most often, you will use the `expand` option when you want to resize a window. In nearly all cases, `fill='both'` is used along with `expand=TRUE`.

As in the previous slide, the cavity is colored light blue.



- The expansion is in both direction, but the frame only 'fills' into the y-direction.

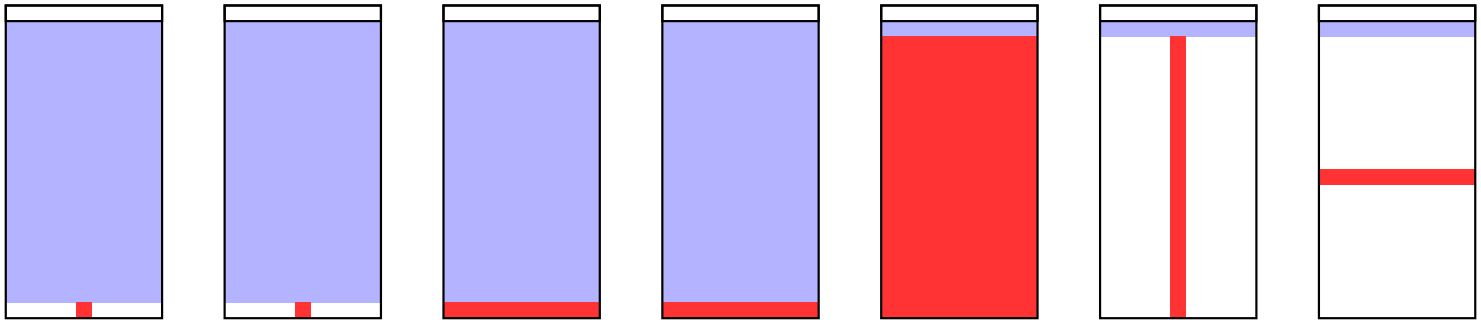
```
tkpack(frame1, fill = 'y', expand = TRUE, side = 'bottom')
```

In that sense, the `fill` option dominates the `expand` option.

Expand and resizing

The `expand=TRUE` packing option lets a widget expand its packing space into unclaimed space in the packing cavity. This is in contrast to the `fill` option which only expands into non-cavity regions. Most often, you will use the `expand` option when you want to resize a window. In nearly all cases, `fill='both'` is used along with `expand=TRUE`.

As in the previous slide, the cavity is colored light blue.



- `expand` always expands the widget towards the cavity.

```
tkpack(frame1, fill = 'x', expand = TRUE, side = 'bottom')
```

Anchoring

If a widget is left with more packing space than display space, you can position it within its packing space using the `anchor` packing option. The default anchor position is `center`. The other options correspond to points on a compass: `n`, `ne`, `e`, `se`, `s`, `sw`, `w` and `nw`.


```
prop <- tkframe(tt, bg = "white", height = 80, width = 20)
base <- tkframe(tt, bg = "gray50", height = 20, width = 120)
tkpack(base, side = "bottom")
foo <- tklabel(tt, text = "Foo")
tkpack(prop, foo, side = "right", expand = TRUE)
```

Notice that the `expand` option gives `prop` and `foo` half of the extra space in x-direction, rather then squeezing them towards the right side.

Anchoring

If a widget is left with more packing space than display space, you can position it within its packing space using the `anchor` packing option. The default anchor position is `center`. The other options correspond to points on a compass: `n`, `ne`, `e`, `se`, `s`, `sw`, `w` and `nw`.

We can change the packing options after a widget is already packed.

```
tkpack(foo, anchor = 'n')
```

Anchoring

If a widget is left with more packing space than display space, you can position it within its packing space using the `anchor` packing option. The default anchor position is `center`. The other options correspond to points on a compass: `n`, `ne`, `e`, `se`, `s`, `sw`, `w` and `nw`.

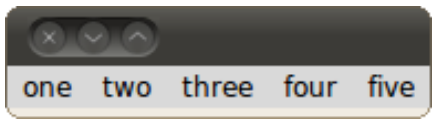
One final example for clarity.

```
tkpack(foo, anchor = 'se')
```

Notice that both `foo` and `prop` occupy half of the x-direction because the `expand` option was set to `TRUE`.

Packing Order

The packer maintains an order among the children that are packed into the frame. By default, each new child is appended to the end of the packing order. The most obvious effect of the order is that the children first in the packing order are closest to the side they are packed against. You can control the packing order with the `before` and `after` packing options, and you can reorganize widgets after they have already been packed.




```
for(label in c("one", "two", "three", "four", "five")) {
  assign(label, tklabel(tt, text = label))
  tkpack(get(label), side = 'left', padx = 5)
}
```


```
tkpack(two, after=four)
```

Pack Slaves, Pack Info and Pack forget

The `tkpack.slaves()` R function returns the list of children in their packing order.

If you need to examine the current packing parameters for a widget use the `tkpack.info()` function.

Finally if you and to unpack a widget (not display it anymore) use the `tkpack.forget()` function. After unpacking a widget, you can always pack (display) it again.