GTKlos extension

Building GUI with STklos

Erick Gallesio

Table of Contents

Chapter 1. Introduction

This extension permits to easily program GUI (Graphical User Interfaces) using the OO model of *STklos*. The model used here, is very similar to the one originally defined in STk Scheme and is discussed in the papers image: - Programming Graphical User Interfaces with Scheme, and in - Designing a Meta Object Protocol to wrap a Standard Graphical Toolkit.

Furthermore, this OO model allows you to define your own widgets thanks to the MOP (Meta Object Protocol) of *STklos*.

1.1. Installation

The extension is in the `extensions/gtklos` directory of *STklos*. It is configured when you run `configure` in the main directory of *STklos*. So, to compile it you just need to run

```
$ make
```

The `demos` directory contains several demos that can be run separately or with the `run-demos` script in this directory. Running this script, with

```
$ cd demos
$ ./run-demos
```

you should obtain something like:

If everything is correct, you can install the *GTklos* extension with a `make install` in the `gtklos` directory. Everything will be installed in a sub-directory of the main *STklos* installation

1.2. Getting started

To use the *GTklos* extension you need to import the `(stklos gtklos)` library. This can be done with:

```
;; Import GTKlos to access its exported symbols
(import (stklos gtklos))
```

1.2.1. A first window

The first thing you must do to make an interface consists to create an instance of the class `<window>`. For instance,

```
stklos> (define w (make <window> #:title "A first window"))
```

will create a window with a title set to `"A first window"`. You can see all the slots that can be set in `w` by using describe:

```
stklos> (describe w)
#[<window> 7b55ac1137b0] is an instance of class <window>.
Slots are:
```

```
      %children = ()
      %data = ()
      %event = ()
      border-width = 0
      can-default = #f
      can-focus = #f
      children = ()
      expand = #f
      focus-on-click = #t
      has-default = #f
      has-focus = #f
      height = 200
      height-request = -1
      modal = #f
      name = ""
      parent = #f
      resizable = #t
      sensitive = #t
      show = #t
      title = "A first window"
      tooltip = #f
      transient = #f
      visible = #t
      wid = #[gtk-window-pointer 57d1624a6df0 @ 7b55ac113780]
      width = 200
      width-request = -1

stklos>
```

Now that the window is created, we need to start the GTK+ interaction loop to see it effectively on our screen. This can be done by calling

- `(start-interactive-gtk)`, or
- `(gtk-main)`

As said by its name, the fist form is preferred when we create an interface interactively in the REPL. This form, call the GTK+ event loop when your keyboard is idle. The second form is generally used when you create a script and don't use the REPL.

As we can see, the *width* and the *height* of this window are reflected in the `width` and `height` of `w`. Hereafter, are some manipulation with the width of `w`:

```
;; Use start-interactive-gtk to develop in the REPL
stklos> (start-interactive-gtk)
;; Setting the width to 400
stklos> (slot-set! w 'width 400)
;; Reading back the value
stklos> (slot-ref w 'width)
400
```

```
;; Since accessors are defined on all slots we can also do
stklos> (width w)
400
stklos> (set! (width w) 300)
stklos> (width w)
300
```

Of course, we can also define the widget size at creation time with a class such as

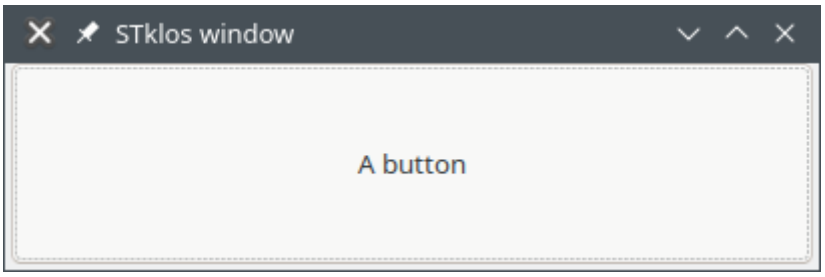
```
stklos> (define w (make <window> #:title "STklos window"
                                 #:width 400
                                 #:height 100))
```

1.2.2. Adding a button

We can add a button to a the previous window bay making an instance of a `<button>`:

```
(define b (make <button> #:parent w #:text "A button"))
```

By saying that the parent of `b` is `w`, the window we have just created just before, this button will be *inside* the `w` window. So, we will obtain:



Using describe, on `b` we have:

```
stklos> (describe b)
#[<button> 7b55ac170480] is an instance of class <button>.
Slots are:
     %children = ()
     %data = ()
     %event = ()
     border-width = 0
     can-default = #f
     can-focus = #t
     children = ()
     command = #f
     expand = #f
     focus-on-click = #t
     has-default = #f
     has-focus = #f
     height = 1
```

```
      height-request = -1
      image = #f
      image-position = left
      name = ""
      parent = #[<window> 7b55ac1137b0]
      relief = normal
      sensitive = #t
      show = #t
      text = "A button"
      tooltip = #f
      use-underline = #t
      visible = #t
      wid = #[gtk-button-pointer 57d162476b20 @ 7b55ac170420]
      width = 1
      width-request = -1
      xalign = 0,5.0
      yalign = 0,5.0

stklos>
```

The slot `command` is particularly important on buttons. It contains the callback function that will be called when we click (with left mouse button) on `b`. The function will be called with two parameters the widget which has been clicked and an event object which contains all the information on the event itself (more on that in Section 4.2.4).

We can add a *command* to the previous button with:

```
stklos> (set! (command b)
              (lambda (w e)
                (printf "Button ~s was clickedn" w)))
```

Now, when clicking the button `b` a message will be printed.

This ends this small introduction on GTKlos.

Chapter 2. Container widgets

A container is a widget that can contain other widgets. It permits to create new windows or organize le layout of a multi widget GUI component.

2.1. Class <window>

The `<window>` class defines a new window. By default, a new window is mapped on the screen when it is created and it's size is 200x200.

Inherited classes:	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	`<dialog>`
	`<vwindow>`
	`<hwindow>`

Direct slots:	height modal resizable title transient width
Direct (non accessor) methods:	realize-widget (<window> <top>)

Slots

- **height** is the height of the window. The default is 200 pixels.
- **modal** is a boolean. It indicates if the window is in modal mode. A modal window prevents interaction with other windows in the same application.
- **resizable** indicates if the window size can be changed.
- **title** indicates the string to display in the window title bar.
- **transient** is a boolean. It indicates if the window is transient.
- **width** is the width of the window. The default is 200 pixels.

Method

- **realize-widget** see general documentation on *realize-widget*.

All these slots have an associated accessor that can be used to read or write the value of the slot. For instance

```
stklos> (define w (make <window> :title "A window"))
;; w
stklos> (title w)
"A window"
stklos> (set! (title w) "Title changed")
stklos> (title w)
"Title changed"
stklos>
```

2.2. Class <vwindow>

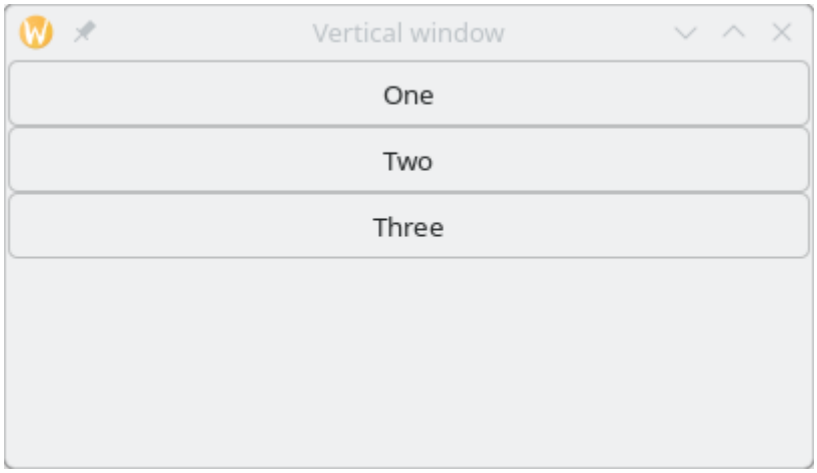
A <vwindow> is a utility class. It is a window which contains a vertical box, to arrange vertically some widgets in it. See example below.

Inherited classes:	<window> <gtk-container> <gtk-widget> <gtk-object>
Directly inheriting classes:	
Direct slots:	
Direct (non accessor) methods:	

Example

```
stklos> (define w (make <vwindow> :title "Vertical window" :width 400))
;; w
stklos> (dolist (b '("One" "Two" "Three")) (make <button> :text b :parent w))
stklos>
```

This will display the following window:



Note

Specifying that a button has `w` as parent permits to embed it in the `w` container.

2.3. Class <hwindow>

A `<hwindow>` is a utility class. It is a window which contains an horizontal box, to arrange horizontally some widgets in it.

Inherited classes:	`<window>`
	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	
Direct (non accessor) methods:	

2.4. Class <gtk-box>

A `<gtk-box>` is a simple container which arranges child widgets into a single row or column, depending upon the value of `orientation` property.

Normally, a class name prefixed by `<gtk-` is an internal class which is not exported bt the GTKlos library. This class name has been choose, because the name `<box>` is already used for the normal ***STklos*** boxes (see SRFI-111).

Inherited classes:	`<gtk-orientable>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	`<vbox>` `<hbox>`
Direct slots:	baseline-position expand fill homogeneous padding spacing
Direct (non accessor) methods:	container-add! (<gtk-box> <gtk-widget> . <top>) realize-widget (<gtk-box> <top>)

Slots

- **baseline-position** indicates the baseline position (see GTK documentation for more information)
- **expand** is a boolean which indicates if children must be expanded by default
- **fill** is a boolean which indicates if children fill the empty space by default.
- **homogeneous** is a boolean. It indicates if all children of the box are given equal space in the box.
- **padding** is teh default value of the padding
- **spacing** is the number of pixels between children of the box (default is 0).

Methods

- **container-add!** accepts a list of keyword parameters after the widget to add to the container. The possible values for these keyword parameters are:
  - `expand` (defaults to the value of slot `expand` slot of the box)
  - `fill` (defaults to the value of slot `fill` slot of the box)
  - `padding` (defaults to the value of slot `padding` slot of the box)
  - `end` add element to the end (default to `#f`)
- **realize-widget** see general documentation on *realize-widget*

Notes

One of the most important slot of this class is the slot `orientation` (not shown here, since it is inherited from the class `<gtk-orientable>`). Its value is a symbol which can be one of the symbols `horizontal` or `vertical`.

2.5. Class <hbox>

This utility class can be used to define a `<gtk-box>` whose orientation is initialized to `horizontal`.

Inherited classes:	`<gtk-box>`
	`<gtk-orientable>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	
Direct (non accessor) methods:	initialize-instance (<hbox> <top>)

Method

- **initialize-instance** see general documentation on *initialize-instance*

2.6. Class <vbox>

This utility class can be used to define a `<gtk-box>` whose orientation is initialized to `vertical`.

Inherited classes:	`<gtk-box>`
	`<gtk-orientable>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	
Direct (non accessor) methods:	initialize-instance (<vbox> <top>)

Method

- **initialize-instance** see general documentation on *initialize-instance*

2.7. Class <frame>

A `<frame>` widget surrounds its child with a decorative frame and an optional label.

Inherited classes:	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	`<vframe>` `<hframe>`
Direct slots:	shadow title xalign yalign
Direct (non accessor) methods:	realize-widget (<frame> <top>)

Slots

- **shadow** can be one of the following symbols `none`, `in`, `out`, `etched-in` or `etched-out`.
- **title** contains the label of the frame
- **xalign** is a float to adjust the position of the title (0: left, 1: right, 0.5:default)
- **yalign** is a float to adjust the position of the title (0: top, 1: bottom, 0.5:default)

2.8. Class <hframe>

This utility class permits to define a frame which contain an horizontal box that can be filled with components that are arranged horizontally.

Inherited classes:	`<frame>`
	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	
Direct (non accessor) methods:	

2.9. Class <vframe>

This utility class permits to define a frame which contain an vframe box that can be filled with components that are arranged vertically.

Inherited classes:	`<frame>`
	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	
Direct (non accessor) methods:	

2.10. Class <grid>

A `<grid>` widget is a container which arranges its child widgets in rows and columns, with arbitrary positions and horizontal/vertical spans.

Inherited classes:	<gtk-container> <gtk-widget> <gtk-object>
Directly inheriting classes:	
Direct slots:	column-homogeneous column-spacing row-homogeneous row-spacing
Direct (non accessor) methods:	container-add! (<grid> <gtk-widget> . <top>) realize-widget (<grid> <top>)

Slots

- **column-homogeneous** is a boolean which indicates whether all columns of the grid will have the same width.
- **column-spacing** is the amount of space between columns of the grid.
- **row-homogeneous** is a boolean which indicates whether all rows of the grid will have the same width.
- **row-spacing** is the amount of space between rows of the grid.

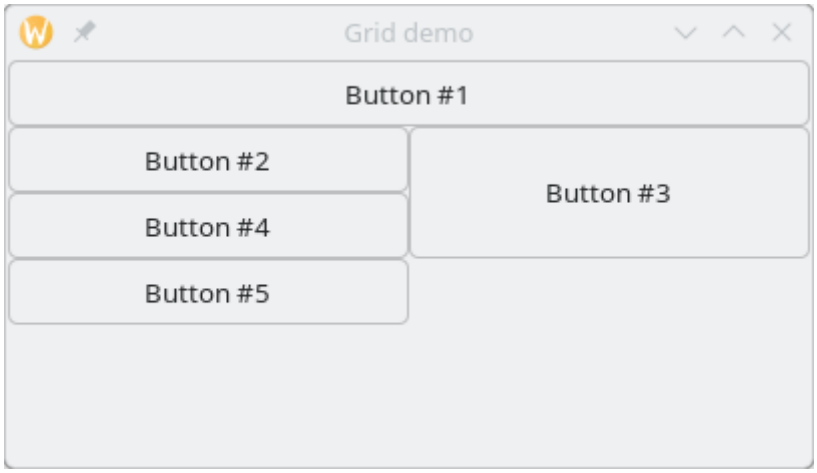
Methods

- **container-add!** accepts a list of keyword parameters after the widget to add to the container. The possible value for these keyword parameters are:
  - `left` is the column position of the added widget (starting from 1)
  - `top` is the line position of the added widget (starting from 1)
  - `width` is the width position of the added widget
  - `height` is the height position of the added widget
- **realize-widget** see general documentation on *realize-widget*

Example

```
stklos> (define w (make <window> :title "Grid demo"))
;; w
stklos> (define g (make <grid> :parent w))
;; g
;; Create 5 buttons
stklos> (define b (map (lambda (x) (make <button> :text x :width 200))
               '("Button #1" "Button #2" "Button #3" "Button #4" "Button #5")))
;; b
;; Add them to the grid
stklos> (container-add! g (list-ref b 0) #:left 0 #:top 0 :width 2)
stklos> (container-add! g (list-ref b 1) #:left 0 #:top 1)
stklos> (container-add! g (list-ref b 2) #:left 1 #:top 1 :height 2)
stklos> (container-add! g (list-ref b 3) #:left 0 #:top 2)
stklos> (container-add! g (list-ref b 4) #:left 0 #:top 3)
```

This will display the following window:



2.11. Class <header-bar>

A `<header-bar>` is similar to a horizontal `<gtk-box>`. Furthermore, this widget can add typical window frame controls, such as minimize, maximize and close buttons, or the window icon. It is often used at the top of a `<vwindow>`

Inherited classes:	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	decoration-layout decoration-layout-set has-subtitle show-close-button subtitle title
Direct (non accessor) methods:	realize-widget (<header-bar> <top>)

Slots

- **decoration-layout** is a string used to indicate the layout of the buttons (see example below)
- **decoration-layout-set** is a boolean used to know if the decoration layout has been set
- **has-subtitle** reserves space for a subtitle (even if not currently set).
- **show-close-button** indicates if the decoration buttons (not only the close button!!) are shown or not. Its default value is `#f`
- **subtitle** indicates the subtitle of the header bar (note that place is always reserved for a subtitle, except is `has-subtitle` is set to `#f`.
- **title** indicates the title of the header bar.

Method

- **realize-widget** see general documentation on *realize-widget*

Exemple

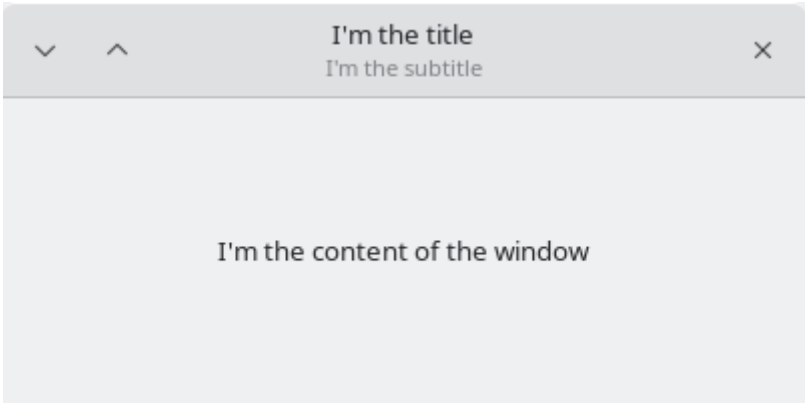
The following example illustrates the use of a header bar.

```
(define w (make <vwindow> :width 400))

(define h (make <header-bar> :title "I'm the title"
                             :subtitle "I'm the subtitle"
                             :parent w
                             :decoration-layout "minimize,maximize:close"
                             :show-close-button #t))

(define l (make <label> :text "I'm the content of the window"
                        :parent (list w :expand #t)))
```

Execution of this code will display the following window



Notes

1. The `decoration-layout` slot is set here to `"minimize, maximize:close"` to place
   - the *minimize* and *maximize* buttons on the left (since they are before the `':'` character)
   - the *close* button on the right (since it is after the `':'` character)
2. The `show-close-button` is set to `#t` so display the control buttons
3. The `parent` is set here to `w` with an indication that it must be expanded into it container (`w` here). See the documentation of `parent` of the `<gtk-widget>` class.

2.12. Class <toolbar>

A `<toolbar>` is container whose constituents are instance of the `<toolbar-item>` class.

Inherited classes:	`<gtk-container>`
	`<gtk-orientable>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	icon-size show-arrow toolbar-style
Direct (non accessor) methods:	add-items-to-toolbar (<toolbar> <top>) container-add! (<toolbar> <toolbar-item> <integer>) container-add! (<toolbar> <toolbar-item>) realize-widget (<toolbar> <top>)

Slots

- **expand** is a boolean. It indicates if toolbar items are expanded or not (default to `#f`)
- **icon-size** can be one of the following symbols `small`, `medium`, `large` or `huge`.
- **show-arrow** is the boolean. It indicates if the toolbar as an overflow menu.
- **toolbar-style**: can be one of the following symbols `icons`, `text`, `both` or `both-horizontal`.

Methods

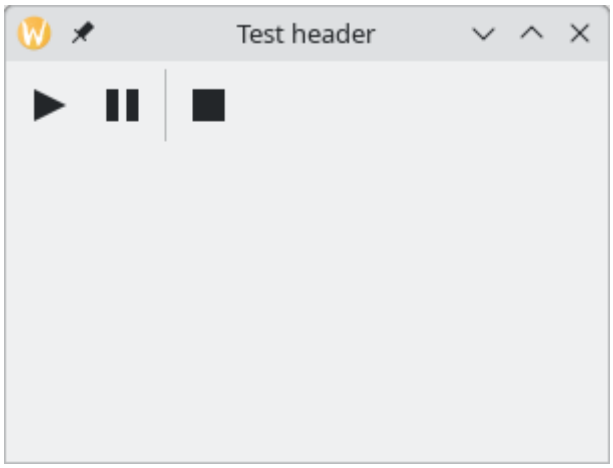
- **add-items-to-toolbar** is a utility method to easily populate the components of a toolbar. It takes a toolbar and a list describing its components with the following convention, for each item of the list:
  1. an empty list specify to add a new separator, that is an instance of `<toolbar-separator-item>`, to the toolbar
  2. a list specifies that a new `<toolbar-icon-item>` must be created and added to the toolbar. The content of the list are the parameters that must be passed during the creation of the icon.
- **container-add!**: see general documentation on *container-add!*.

Exemple

```
(define w (make <vwindow> :title "Test header" :width 300))
(define tb (make <toolbar> :parent w))

(add-items-to-toolbar tb                          ;; populate the toolbar
 '((:text "Play"  :icon-name "media-playback-start")
   (:text "Pause" :icon-name "media-playback-pause")
   ()  ;; <== A separator
   (:text "Stop" :icon-name "media-playback-stop")))
```

Execution of this code will display the following window



Notes

1. For the sake of simplicity, the buttons are inactive here (use the `command` slot to add an action when the toolbar button is clicked).
2. Icons her are stock buttons they are searched by the GTK library in the standard directory (generally `/usr/share/icons` on GNU/Linux).
3. Since `<gtk-toolbar>` inherits from `<gtk-orientable>`, a toobar can be horizontal or vertical.

2.12.1. Class <toolbar-item>

The `<toolbar-item>` class is the parent class of the toolbar items classes that can be added to a to a GTK toolbar. It offers only two methods (**container-add**) to add an item to a toolbar.

Inherited classes:	<gtk-widget> <gtk-object>
Directly inheriting classes:	<toolbar-button-item> <toolbar-separator-item>
Direct slots:	
Direct (non accessor) methods:	container-add! (<toolbar> <toolbar-item> <integer>) container-add! (<toolbar> <toolbar-item>)

Methods

There are two methods of the generic function **container-add!** to add an item to a container:

- with 2 parameters, the methods permit to append the new item at the end of already added items.
- with 3 parameters, the method adds the given item at the position given as third parameter (an integer). If the position is 0 the item is prepended to the start of the toolbar. If it is negative, the item is appended to the end of the toolbar.

2.12.2. Class <toolbar-separator-item>

The class <toolbar-separator-item> permits to define a separator to a toolbar.

Inherited classes:	<toolbar-item> <gtk-widget> <gtk-object>
Directly inheriting classes:	
Direct slots:	
Direct (non accessor) methods:	realize-widget (<toolbar-separator-item> <top>)

Methods

- **realize-widget** see general documentation on *realize-widget*.

2.12.3. Class <toolbar-button-item>

The class <toolbar-separator-item> permits to define a button to a toolbar. The button can have an image and a text.

Inherited classes:	`<toolbar-item>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	command icon-name text
Direct (non accessor) methods:	realize-widget (<toolbar-button-item> <top>)

Slots

- **command** is identical to the command associated to a button (see general documentation on command *slot*)
- **icon-name** is a string which contains is the name of the themed icon displayed on the item. Icons are searched by the GTK library in the standard directory (generally `/usr/share/icons` on GNU/Linux).
- **text** is the text of the button item.

Method

- *realize-widget: see general documentation on *realize-widget*

2.13. Class <scroll>

The `<scroll>` class represents a container that accepts a single child widget and makes that child scrollable using scollbars.

Inherited classes:	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	hpolicy max-content-height max-content-width min-content-height min-content-width overlay-scrolling vpolicy window-placement
Direct (non accessor) methods:	realize-widget (<scroll> <top>)

Slots

- **hpolicy** determines how the size should be computed to achieve the one of the visibility

mode for the horizontal scrollbar. It's value can be one of the symbols `always`, `automatic`, `never` or `external` (default is `automatic`).

- **max-content-height** is the maximum content height of the scrolled window, or -1 if not set.
- **max-content-width** is the maximum content width of the scrolled window, or -1
- **min-content-height** is the minimum content height of scrolled_window, or -1 if not set.
- **min-content-width** is the minimum content width of scrolled_window, or -1 if not set.
- **overlay-scrolling** indicates whether overlay scrolling is enabled or not. If it is, the scrollbars are only added as traditional widgets when a mouse is present. Otherwise, they are overlayed on top of the content, as narrow indicators.
- **vpolicy** determines how the size should be computed to achieve the one of the visibility mode for the vertical scrollbar. It's value can be one of the symbols `always`, `automatic`, `never` or `external` (default is `automatic`).
- **window-placement** indicates where the scollbars are placed. It's value can e one of the symbols `top-left`, `bottom-left`, `top-right` or `bottom-right`.

Method

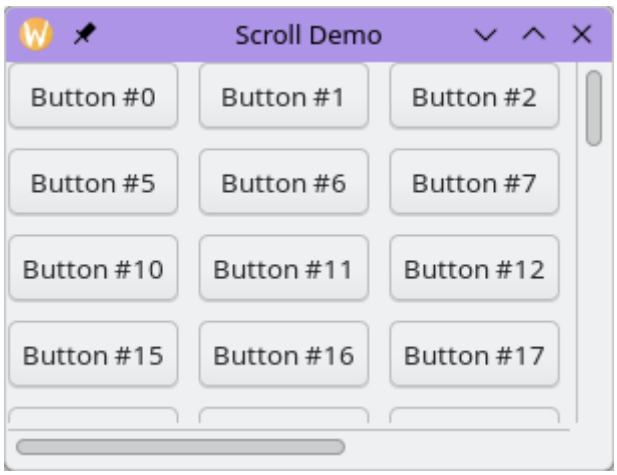
- **realize-widget** see general documentation on *realize-widget*.

Example

```
(let* ((win    (make <vwindow> #:title "Scroll Demo"
                       #:width 300 #:height 200
                       #:expand #t #:fill #t))
       (scroll (make <scroll> #:parent win :overlay-scrolling #f))
       (grid   (make <grid> #:parent scroll #:row-spacing 10 #:column-spacing 10)))

  ;; Build a set of  buttons in the grid contained in <scroll>
  (dotimes (i 100)
    (make <button>
          #:text (format "Button #~A" i)
          #:parent (list grid #:top (quotient i 5) #:left (modulo i 5)))))
```

It will display the following window:



Chapter 3. Display widgets

A display widget is a GUI component which shows some information (a text, an image, or a progress bar for instance)

3.1. Class <image>

The class `<image>` permits to display an image.

Inherited classes:	`<gtk-misc>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	file-name icon-name icon-size
Direct (non accessor) methods:	get-image-pixbuf (<image>) realize-widget (<image> <top>)

Slots

- **file-name** designates a file containing the image to display. If the file cannot be found. a "broken image" icon will be displayed. If the file contains an animation, the image will be animated. The `image-path` parameter object is used to find the image file (see below).
- **icon-name**: is a string which contains is the name of the themed icon displayed on the item. Icons are searched by the GTK library in the standard directory (generally `/usr/share/icons` on GNU/Linux).
- **icon-size** can be one of the following symbols `small`, `medium`, `large` or `huge`.

Methods

- **get-image-pixbuf** return a C pointer on the pixbuf used to represent the image. This can be useful to insert an image in a canvas.
- **realize-widget** see general documentation on *realize-widget*.

Parameter object

- **image-path** is a parameter object which denotes the paths to be searched when specifying a file name with the `file-name` slot. The default path contains the current directory and a directory depending of ***STklos*** installation directory. Alternatively, the default path can also be specified with the shell variable `STKLOS_INDEX_TERM`.

3.2. Class <label>

A `<label>` permits to define a text widget which displays a small text.

Inherited classes:	`<gtk-misc>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	justify selectable text
Direct (non accessor) methods:	realize-widget (<label> <top>)

Slots

- **justify** slot may contain one of the following symbols: `left`, `right`, `center` or `fill`.
- **selectable** isa boolean. If `#t`, the user can select text from the label, for copy-and-paste.
- **text** slot contains the text of the label.

Method

-**realize-widget** see general documentation on *realize-widget*.

3.3. Class <progress-bar>

A `<progress-bar>` can be used to display the progress of a long running operation; it can be used in two different modes: *percentage* mode and *activity mode*.

Inherited classes:	`<gtk-widget>`
	`<gtk-orientable>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	inverted pulse-step show-text text value
Direct (non accessor) methods:	progress-bar-pulse (<progress-bar>) realize-widget (<progress-bar> <top>)

Slots

- **inverted** is a boolean. It permits to invert the growing direction of a progress bar (form top to bottom for vertical progress bars and from left to right for horitontal ones).
- **pulse-step** indicates the fraction of the progress bar used to advance to the next step.
- **show-text** is a boolean used to indicate if a text is shown in the progress bar. The shown text is either the value of the `text` slot or, if not set, the value of the `value` slot, as a percentage.
- **text** is a string. It is the legend of the progress bar.
- **value** denotes the fraction of the task which is already done. Setting this slot permit to use the progress bar in *percentage* mode.

Methods

- **progress-bar-pulse** can be used to indicate that some progress has been made. The progress bar enters "activity mode," where a block bounces back and forth.
- **realize-widget** see general documentation on *realize-widget*.

3.4. Class <scale>

The class `<scale>` defines a slider control used to select a numeric value.

Inherited classes:	`<gtk-widget>` `<gtk-object>`
Directly inheriting classes:	
Direct slots:	command digits draw-value has-origin increment lower orientation upper value value-pos
Direct (non accessor) methods:	realize-widget (<scale> <top>)

Slots

- **command** contains a function that will be called when the value of the scale is changed. See general description on the the *command* slot in Chapter 4.
- **digits** specifies the number of decimal places that are displayed in the value.
- **draw-value** is a boolean. It indicates whether the current value is displayed.
- **has-origin** is a boolean. It indicates if the scale has an origin.

- **increment**
- **lower** is the minimum possible value of the scale.
- **orientation** indicates the orientation of the scale. Warning: it is a **read-only** slot.
- **upper** is the maximum possible value of the scale.
- **value** is the value of the scale. Setting it will move the scale.
- **value-pos** indicates the position where the value is displayed. Its value can be the symbol `top`, `bottom`, `left` or `right`.

Method

- *realize-widget see general documentation on *realize-widget*.

3.5. Class <separator>

A `<separator>` is a horizontal or vertical separator widget, used to group the widgets within a window. It displays a line with a shadow to make it appear sunken into the interface.

Inherited classes:	`<gtk-orientable>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	
Direct (non accessor) methods:	realize-widget (<separator> <top>)

Method

- **realize-widget** see general documentation on *realize-widget*.

Chapter 4. Signal and Events

When a user interacts with a GTklos with an input device (e.g. moves a mouse or presses a key on the keyboard), GTK receives events from the windowing system. Those events are generally directed to a widget (the area under the pointer mouse for mouse events or the focused widget for a keyboard event). For more information on GTK events you can see *The GTK Input and Event Handling Model* chapter in GTK documentation.

4.1. Event primitives

The GTklos ***STklos*** extension defines a new object type called *GTK-event* to handle events. Some primitives are also defined.

STklos procedure

```
(get-gtk-event)
```

This function returns a new *GTK-event* object representing the event currently processed by GTK. If no event is processed this function returns `#f`

STklos procedure

```
(event-type ev)
```

GTK events can have different types, depending of the event occuring on a GTklos widget. The function `event-type` returns an information on the type of the event as a symbol. The possible values returned by this function can be one of the following symbols:

NOTHING	DELETE	DESTROY	EXPOSE
MOTION	BUTTON-PRESS	BUTTON-RELEASE	KEY-PRESS
KEY-RELEASE	ENTER	LEAVE	FOCUS-IN
FOCUS-OUT	CONFIGURE	MAP	UNMAP
PROPERTY	SELECTION-CLEAR	SELECTION-REQUEST	SELECTION-NOTIFY
PROXIMITY-IN	PROXIMITY-OUT	DRAG-ENTER	DRAG-LEAVE
DRAG-MOTION	DRAG-STATUS	DROP-START	DROP-FINISHED
CLIENT-EVENT	VISIBILITY		

See GTK documentation for more information.

STklos procedure

```
(event-x ev)
```

The primitive `event-x` returns the event window relative x coordinates from the `ev` event. It returns `#f` if the the given event does not contain such a coordinate.

STklos procedure

```
(event-y ev)
```

The primitive `event-y` returns the event window relative y coordinates from the `ev` event.It returns `#f` if the the given event does not contain such a coordinate.

STklos procedure

```
(event-char ev)
```

The primitive `event-char` returns the character associated to the event `ev` if it is a key press or key release event (the result is a Scheme character). If a key is not pressed or released, the function returns `#f`.

STklos procedure

```
(event-keyval ev)
```

Returns the *keyval* associated to event `ev` as an integer, or `#f` if it does not exists.

STklos procedure

```
(event-keycode ev)
```

Returns the *keycode* associated to event `ev` as an integer, or `#f` if it does not exists.

STklos procedure

```
(event-modifiers ev)
```

Returns a list of the modifiers associated to the ev event. The components of the list can be the following symbols

- *shift*
- *lock*
- *control*
- *mod1*, *mod2*, *mod3*, *mod4* or *mod5*

If no modifier is pressed. The function returns the empty list.

Example

```
(define w (make <vwindow> :title "Testing events" :width 300 :height 40))

;; define an entry for the test
(define e (make <entry>
            :placeholder-text "Press any mouse button on this entry"
            :parent (list w :expand #t :fill #t)))

;; Connect an event handler to the e entry
(event-connect e "button-press-event"
               (lambda (wid ev)
                 (eprintf "Button pressed. Modifiers: ~Sn"
                          (event-modifiers ev))))
```

⇒

```
Button pressed. Modifiers: ()
Button pressed. Modifiers: (control)
Button pressed. Modifiers: (control shift)
Button pressed. Modifiers: (mod1 control shift)
```

See also the **event-connect** primitive for details.

STklos procedure

```
(event-button ev)
```

Returns the number of the button involved in the event ev. If ev doesn't contain a button, event-button returns #f.

```
(event-describe ev)
```

This primitive prints on the current error port some information on the `ev` event.

Result looks like this

```
vent description #[GTK-event 707c2c5a9ea0]
            type: KEY-PRESS
          button: #f
       modifiers: (shift)
            char: #H
          keyval: 72
         keycode: 43
               x: #f
               y: #f
```

4.2. Signal primitives

4.2.1. Notion of signal

A GTKlos application is, in fact, an event driven application. When an event occurs (a mouse click, a mouse motion, ...), the application may react to this event. If there no event, the application is generally idle. When an event reaches a widget, it may react to this event by emitting a signal. A GTKlos program can connect a specific *callback* to a signal. So, a callback is a kind of handler functions in charge of the signal.

4.2.2. Event callback

A signal name is a string (such as "clicked" "enter", "leave", "destroy", ...). All GTKlos callbacks are Scheme functions with two parameters:

1. the widget receiving the event
2. the event itself

4.2.3. Event connection

To handle a signal, we can add an event handler for this signal. The connection between the signal and the event is done by the `event-connect` primitive. This function takes three parameters

1. the widget ti which we want to connect an handler
2. the name of the signal sent when this event occurs

3. the callback function

Each widget has a given set of possible signal names that it sends to the application (see the GTK+ documentation for more information).

Hereafter is an example which uses two calls to `event-connect`

```
(define win (make <vwindow> :title "Demo window" :width 300 :height 40))

(define but (make <button> :text "Test button" :parent win))

(event-connect but "clicked"
               (lambda (wid ev)
                 (eprintf "Widget ~S was clickedn" wid)
                 (event-describe ev)))

(event-connect but "enter"
               (lambda (wid _) (eprintf "We enter the button ~Sn" wid)))
```

An example of output:

```
Widget #[<button> 7b70477ab0c0] was clicked     <= ①
Event description #[GTK-event 7b70477bd720]     <= ②
             type: BUTTON-RELEASE
           button: 1
        modifiers: ()
             char: #f
           keyval: #f
          keycode: #f
                x: 50.0625
                y: 23.7734375
```

We see here that we have:

1. the mouse enters in the button (event "enter")
2. the button 1 has been pressed (event "clicked")

`event-connect` connects a callback function to a signal for a particular object. The handler will be called **before** the default handler for the signal. To connect a callback that will be called **after** the default handler, you can use `event-connect-after`.

4.2.4. The command slot

Numerous GTKlos widgets have a slot named `command` which permits to fast-connect a "clicked" signal to a widget. This signal is generally sent by reactive widgets when the button 1 or the mouse is clicked over the widget. Hence, the connection of the "clicked" signal of the previous example can

be done at the button definition:

```
(define but (make <button> :text "Test button" :parent win
                  :command (lambda (wid ev)
                             (eprintf "Widget ~S was clickedn" wid)
                             (event-describe ev))))
```

Chapter 5. Button widgets

5.1. Class <button>

A `<button>` widget is generally used to trigger a callback function that is called when the button is pressed.

Inherited classes:	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	`<check-button>`
Direct slots:	command image image-position relief text use-underline xalign yalign
Direct (non accessor) methods:	realize-widget (<button> <top>)

Slots

- **command** denotes the callback called when the button is clicked (see command *slot*).
- **image** contains the child widget to appear next to the button text.
- **image-position** is a symbol used to define the position of the image relative to the text inside the button. It's value can be one of the symbols `left`, `right`, `top`, `bottom`.
- **relief** is a symbol describing the relief style of the button. Its value can be the `normal` or `none`.
- **text** is a string which contains the button label.
- **use-underline** if true, an underline in the text of the button label indicates the next character should be used for the mnemonic accelerator key.
- **xalign** is a float that can be used to control the horizontal position of button content (0.0 is left and 1.0 is right). The default value is 0.5.
- **yalign** is a float that can be used to control the vertical position of button content (0.0 is top and 1.0 is bottom). The default value is 0.5.

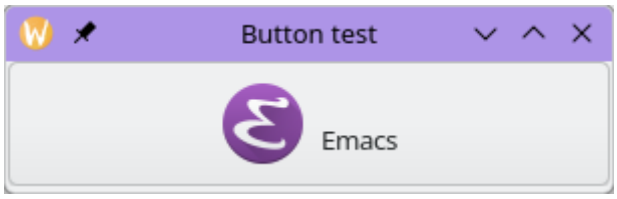
Method

- **realize-widget** see general documentation on *realize-widget*.

Here is a simple button with a text and an image. Clicking on the button, will launch the *emacs* editor:

```
(let ((w (make <vwindow> :title "Button test" :width 300 :height 40)))
  ;; create a button with a text and an image
  (make <button> :text " Emacs "
        :image (make <image> :icon-name "emacs")
        :parent w
        :command (lambda (e w) (system "emacs &"))))
```

This code displays a window with a the *emacs* button:



5.2. Class <check-button>

A <check-button> is a button with a toggle button displayed next to the button label.

Inherited classes:	<button> <gtk-container> <gtk-widget> <gtk-object>
Directly inheriting classes:	<radio-button>
Direct slots:	value
Direct (non accessor) methods:	realize-widget (<check-button> <top>)

Slot

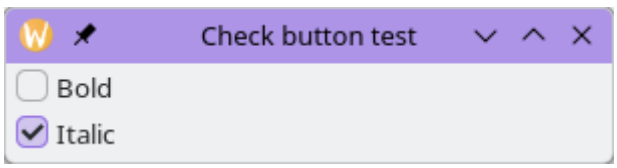
- **value** is a boolean. It indicates if the check button is on or off.

Method

- **realize-widget*** see general documentation on *realize-widget*.

```
(let ((w (make <vwindow> :title "Check button test" :width 300 :height 40)))
  (make <check-button> :text "Bold"   :parent w)
  (make <check-button> :text "Italic" :parent w :value #t))
```

This code displays the following window:



5.3. Class <radio-button>

A `<radiobutton>` is similar to a check button. Radio buttons are generally pcas in a group of radio buttons. When one is selected, all other radio buttons in the same group are deselected. It is one way of giving the user a choice from many options.

Inherited classes:	`<check-button>`
	`<button>`
	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	radio-selected sibling
Direct (non accessor) methods:	initialize-instance (<radio-button> <top>) radio-group (<radio-button>) realize-widget (<radio-button> <top>)

Slots

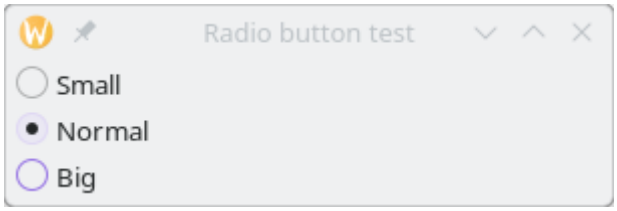
- **radio-selected** contains the radio button which is selected in the radio-button's group. For-instance the expression `(text (radio-selected b))` can be used to find the label string of the radio group to which the button `b` belongs. Note any button of the group can be used here to find the selected radio button.
- **sibling** is a read-only slot. It contains the value of the `#:sibling` option passed when the widget was initialized. See `initialize-instance` below.

Methods

- **initialize-instance** for radio buttons takes care of the `#:sibling` option which can be passed during their creation. If no sibling is passed (or is `#f`), the radio button creates a new group and register it in this group. If a sibling is passed, it must be a radio button and the new created button will be registered in the same group of this sibling radio button.
- **radio-group** returns a list of all the buttons of the group to which belongs the radio button passed as parameter.
- **realize-widget** see general documentation on *realize-widget*.

```
(let* ((w  (make <vwindow> :title "Radio button test" :width 300 :height 40))
       (b1 (make <radio-button> :text "Small"  :parent w))
       (b2 (make <radio-button> :text "Normal" :parent w :sibling b1))
       (b3 (make <radio-button> :text "Big"    :parent w :sibling b1)))
    (set! (radio-selected b1) b2)) ;; Select button b2 instead of the group leader b1
```

The evaluation of this code will display the following window.



5.4. Class <combobox>

A <combobox> permits the user to choose from a list of valid choices. It displays the selected choice. When activated, the combo box displays a popup with a list of valid choices.

Inherited classes:	<gtk-container> <gtk-widget> <gtk-object>
Directly inheriting classes:	<entry-combobox>
Direct slots:	command items value
Direct (non accessor) methods:	container-add! (<combobox> <string>) container-add! (<combobox> <top>) realize-widget (<combobox> <top>)

Slots

- **command** contains the callback function that will be used when an entry is selected.
- **items** is a list of strings. It represents the correct values for the combo box.
- **value** is the current selected value of in the combo box. If no value is selected, this slot is #f.

Methods

- **container-add!** permits to append a value to the end of the list of items. If the value given is not a string, it is converted before to a string.
- **realize-widget** see general documentation on *realize-widget*.

Example

```
(let ((w  (make <vwindow> :title "Combo box test" :width 300 :height 40)))
  (make <combobox> :parent w  :items '("XS" "S" "M" "L" "XL")
        :value "M"
        :command (lambda (w e)
                   (printf "You have selected ~Sn" (value w)))))
```

5.5. Class <entry-combobox>

An `<entry-combobox>` is similar to a `<combo-box>`, excepts that its value can be edited.

Inherited classes:	`<combobox>`
	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	has-frame value
Direct (non accessor) methods:	realize-widget (<entry-combobox> <top>)

Slots

- **has-frame** is a boolean. It indicates whether a frame is drawn around the entry.
- **value** is the current selected value of in the combo box. If no value is selected, this slot is `#f`.

Method

- **realize-widget** see general documentation on *realize-widget*.

5.6. Menus

A GTKlos drop down menu is a special kind of button composed of menu items (of class `<menu-item>`). The menu buttons are generally arranged in a menu bar (of class `menu-bar>`).

5.6.1. Class <menu-bar>

A `<menu-bar>` is a container whose children are menu items. It permits to implement a menu bar with several sub-menus.

Building a menu bar and its components generally takes a lot of lines of code. Using the `add-items-to-menubar` method can be used to simplify this task (see below).

Inherited classes:	<gtk-menu-shell> <gtk-container> <gtk-widget> <gtk-object>
Directly inheriting classes:	
Direct slots:	chiild-pack-direction pack-direction
Direct (non accessor) methods:	add-items-to-menubar (<menu-bar> <top>) realize-widget (<menu-bar> <top>)

Slots

- **child-pack-direction** is a symbol which describes how the children menu items composing the menu in the bar are added. It's value is one of the symbols 'left→right' (the default), `right→left`, `top→bottom` or `bottom→top`.
- **pack-direction** is a symbol which describes how the menu items composing the menu in the bar are added. It's value is one of the symbols 'left→right' (the default), `right→left`, `top→bottom` or `bottom→top`.

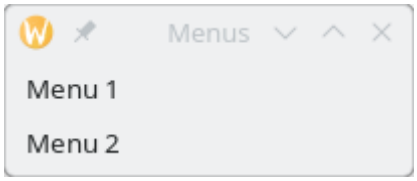
Methods

- **add-items-to-menubar** permit to create a menu bar and fill its components in a declarative way. A description of this method is given in Section 5.6.7
- **realize-widget** see general documentation on *realize-widget*

In the following example, we create a menu bar with two menu items arranged vertically, thanks to the `pack-direction` slot which is set to `top→bottom`

```
(let* ((win (make <vwindow>  :title "Menus" :height 20))
       (mb  (make <menu-bar>  :parent win :pack-direction 'top->bottom)))
  (make <menu-item> :text "Menu 1" :parent mb)
  (make <menu-item> :text "Menu 2" :parent mb))
```

The menu bar:



5.6.2. Class <menu>

A `<menu>` implements a drop down menu consisting of a list of **menu items**. A **<menu>** is most commonly dropped down by activating a menu item in a **<menu-bar>**

Inherited classes:	`<gtk-menu-shell>`
	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	active reserve-toggle-size
Direct (non accessor) methods:	container-add! (<menu-item> <menu> . <top>) realize-widget (<menu> <top>)

Slots

- **active** contains the index of the currently selected menu item, or -1 if no menu item is selected.
- **reserve-toggle-size** is a boolean that indicates whether the menu reserves space for toggles and icons, (even if not present).

Methods

- **container-add!** see the description of this method in Section 5.6.3
- **realize-widget** see general documentation on *realize-widget*.

5.6.3. Class <menu-item>

A `<menu-item>` (or a derived class) is the only possible class of the components of a **<menu>**.

Inherited classes:	`<gtk-menu-item>`
	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	`<menu-check-item>`
Direct slots:	right-justified text use-underline
Direct (non accessor) methods:	container-add! (<menu-item> <menu> . <top>) realize-widget (<menu-item> <top>)

Slots

- **right-justified** is boolean which indicates whether the menu item appears justified at the right side of a menu bar.
- **text** contains the text displayed in the menu item.
- **use-underline** indicates if an underline character the menu item's text consists in a

accelerator key.

Methods

- **container-add!** permits to add a sub-menu (second parameter) to a menu item (first parameter. Other arguments are ignored.
- **realize-widget** see general documentation on *realize-widget*.

5.6.4. Class <menu-check-item>

A `<menu-check-item>` is a menu item that maintains the state of a boolean value in addition to a **<menu-item>** usual role in activating application code.

A check box is displayed at the left side of the menu item to indicate the boolean state. Activating the menu item toggles the boolean value.

Inherited classes:	`<menu-item>`
	`<gtk-menu-item>`
	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	`<menu-radio-item>`
Direct slots:	draw-as-radio inconsistent value
Direct (non accessor) methods:	realize-widget (<menu-check-item> <top>)

Slots

- **draw-as-radio** is a boolean. It indicates if the check box is drawn as a `<menu-radio-item> .
- **inconsistent** is a boolean. It graphically indicates that the check button is in an inconsistent state (check button is not empty and not checked). This can be useful if clicking this value is not coherent with the current state of the application.
- **value** is a boolean. It indicates if the box is checked or not.

Methods

- **realize-widget** see general documentation on *realize-widget*.

5.6.5. Class <menu-radio-item>

A `<menu-radio-item>` is a check menu item that belongs to a group. At each instant exactly one of the radio menu items from a group is selected.

Inherited classes:	<menu-check-item> <menu-item> <gtk-menu-item> <gtk-container> <gtk-widget> <gtk-object>
Directly inheriting classes:	
Direct slots:	sibling
Direct (non accessor) methods:	initialize-instance (<menu-radio-item> <top>) realize-widget (<menu-radio-item> <top>)

Slot

- **sibling**

Methods

- **initialize-widget** accepts a list of keyword arguments. You can use the `#sibling` argument to set join the newly created radio button to an already created menu radio item.
- **realize-widget** see general documentation on *realize-widget*.

Example

```
(let* ((win (make <vwindow>  :title "Menus" :width 300 :height 150))
       (mb  (make <menu-bar>  :parent win))
       (mi  (make <menu-item> :text "A menu" :parent mb))
       ;; Add a menu to mi menu item
       (m   (make <menu> :parent mi))
       ;; Add 3 menu-radio-items to m (in the same group (group leader is c1)
       (c1  (make  <menu-radio-item> :text "Check 1" :parent m :value #t))
       (c2  (make  <menu-radio-item> :text "Check 2" :parent m :sibling c1))
       (c3  (make  <menu-radio-item> :text "Check 3" :parent m :sibling c1)))
  'done)
```

Here, we create a menu bar `mb` with only one menu item `mi`. This menu item contains a sub-menu `m`. In `m`, we have added 3 radio buttons in the same group (they all belong to the group of `c1`). Note that `c1` is clicked since its value is `#t`.

5.6.6. Class <menu-separator-item>

The `<menu-separator-item>` is a separator used to group items within a menu. It displays a horizontal line in the interface.

Inherited classes:	<gtk-menu-item> <gtk-container> <gtk-widget> <gtk-object>
Directly inheriting classes:	
Direct slots:	
Direct (non accessor) methods:	realize-widget (<menu-separator-item> <top>)

Method

- **realize-widget** see general documentation on *realize-widget*.

5.6.7. Method add-items-to-menubar

As you have probably seen in previous examples, building a menu bar an all its sub-menus is quite complex, and easily error prone. The helper method `add-items-to-menubar` permits to have a mod declarative approach. This method takes two parameters: a menu bar and a list specifying the components to embed in this menu bar.

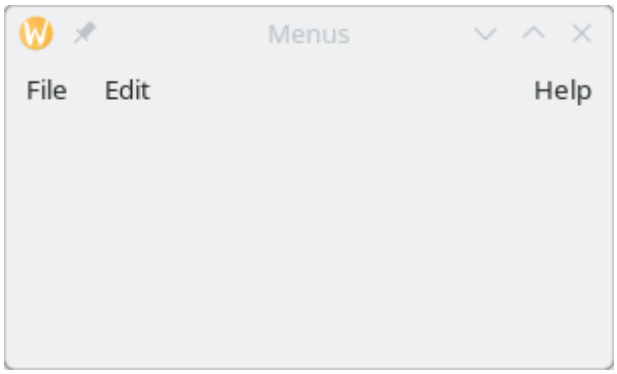
The specification list components are all Scheme lists specifying a child menu item of the menu bar. By convention, an empty list indicates that the next menu items are placed on the right of the menu bar.

A child menu item is described also by a list whose, first item is the text displayed in its sub menu.

As a first example, we can construct a simple interface:

```
(let* ((win (make <vwindow>  :title "Menus" :width 300 :height 150))
       (mb  (make <menu-bar>  :parent win)))
  (add-items-to-menubar mb
                        `(("File")
                          ("Edit")
                          ()
                          ("Help"))))
```

It will produce the following window on screen:



We can now populate each menu item of this menu bar. by specifying their components in each

sub-list. For instance, to add several sub-menus in the previous "File" menu item, we can replace `("File")` by the following list:

```
("File"
    ("Load")
    ("Save")
    ("") ;; <== We want a separator here
    ("Quit"))
```

For each menu item of this sub menu, we can specify which kind of menu item we have with the `:type` key. This key accepts the following values: `:item` (the default), `:check`, `:radio`, `:separator`, `:cascade` (for a cascading menu). Each kind of menu item accepts the `:command` key to specify the callback that mus be run when the menu item is clicked. For radio buttons, the boolean key `:first` can be used to specify that this menu item is the fist one of a new group of radio button.

For the previous example, we could have:

```
("File"
    ("Load" :command ,do-load )   ; do-load is a callback function
    ("Save" :command ,do-save )   ; do-save too
    ("" :type :separator )
    ("Quit" :command ,(lambda ignore (exit 0))))
```

For cascading menu, the sub menu mus be given with the `:menu` key. For instance, we could replace the previous example with

```
("File"
    ("Load"   :command ,do-load )   ; do-load is a callback function
    ("Save"   :command ,do-save )   ; do-save too
    ("Export" :type :cascade
              :menu (("PDF" :command ,export-pdf)   ; a callback
                     ("PNG" :command ,export-png))) ; another one
    ("" :type :separator )
    ("Quit"   :command ,(lambda ignore (exit 0))))
```

To conclude, hereafter is an example using all the possible menu items:

```
(define (action w e)
  (eprintf "You have clicked ~Sn" (text w)))

(let* ((win (make <vwindow>  :title "Menus" :width 300 :height 150))
       (mb  (make <menu-bar>  :parent win)))

  (add-items-to-menubar
        mb
        `(("File"
           ("Load"   :command ,action)
           ("Save"   :command ,action)
           (""       :type    :separator)
           ("Quit"   :command ,(lambda _ (exit 0))))
          ("Edit"
           ("Copy"   :command ,action)
           ("Cut"    :command ,action)
           ("Paste"  :command ,action))
          ("Cascade"
           ("  1  " :type :cascade
            :menu (("One"  :command ,action)
                   ("Un"   :command ,action)
                   ("Eins" :command ,action)))
           ("  2  " :type :cascade
            :menu (("Two"  :command ,action)
                   ("Deux" :command ,action)
                   ("Zwei" :command ,action)))
           ("  3  " :command ,action)
           ("  4  " :command ,action))
          ("Check"
           ("option1" :type :check :command ,action)
           ("option2" :type :check :command ,action :value #t))
          ("Radio"
           ("radio1 group1" :type :radio :command ,action)
           ("radio2 group1" :type :radio :command ,action :value #t)
           (""       :type :separator)
           ("radio1 group2" :type :radio :command ,action :first #t)
           ("radio2 group2" :type :radio :command ,action))
          ()  ;; Add an empty list to make space
          ;; Now "Help" will be on the right part of the tool-bar
          ("Help"
           ("About"     :command ,action)
           ("More Info" :command ,action)))))
```

Chapter 6. Entries and Text widgets

6.1. Class <entry>

An `<entry>` widget is a single line text entry widget.

Inherited classes:	`<gtk-widget>` `<gtk-object>`
Directly inheriting classes:	
Direct slots:	activates-default caps-lock-warning cursor-position has-frame max-length placeholder-text primary-icon-name secondary-icon-name text-editable text-overwrite text-visibility value
Direct (non accessor) methods:	realize-widget (<entry> <top>)

Slots

- **activates-default** is a boolean which tells if the entry will activate the default widget.
- **caps-lock-warning** is a boolean which tells if a password entry will show a warning when Caps Lock is on.
- **cursor-position** indicates the position of the cursor in the entry.
- **has-frame** is a boolean which tells if the entry has a frame.
- **max-length** is the maximum length of the entry.
- **placeholder-text** is the string which is displayed in the entry when it is empty and unfocused.
- **primary-icon-name** is the name of the icon to use for the primary icon for the entry. Icons are searched by the GTK library in the standard directory (generally `/usr/share/icons` on GNU/Linux).
- **secondary-icon-name** is the name of the icon to use for the secondary icon for the entry.
- **text-editable** is a boolean which indicates if the text entry is modifiable.
- **text-overwrite** is a boolean which tells if the text is overwritten when typing in the entry.

- **text-visibility** is a boolean which tells if the text in the entry is visible. Setting the visibility to false is useful to read secrets.
- **value** is a string which contains the text in the entry.

Method

- **realize-widget** see general documentation on *realize-widget*

6.2. Class <text>

The `<text>` widget permits to create an editable text widget.

Inherited classes:	`<gtk-widget>` `<gtk-object>`
Directly inheriting classes:	
Direct slots:	accepts-tab cursor-visible justify text-buffer text-editable text-indent text-monospace text-overwrite text-wrap value
Direct (non accessor) methods:	event-connect (<text> <top> <top>) event-connect-after (<text> <top> <top>) realize-widget (<text> <top>) text-copy-clipboard (<text>) text-cut-clipboard (<text>) text-paste-clipboard (<text>)

Slots

- **accepts-tab** indicates if the text widget insert a tab character when the `Tab` key is pressed.
- **cursor-visible** indicates if the cursor is displayed.
- **justify** is one of the following symbols: `left`, `right`, `center` or `fill`.
- **text-buffer** contains a C pointer to the representation of the internal GTK buffer (handle with care).
- **text-editable** is a boolean which indicates if the text widget is modifiable.
- **text-indent** contains the value of the default indentation for paragraphs.
- **text-monospace** is a boolean which indicates that the text content should use monospace

fonts.

- **text-overwrite** is a boolean which tells if the text is overwritten when typing in the text widget.
- **text-wrap** is a boolean. It indicates if text must be wrapped in the widget
- **value** contains the characters of the text widget

Methods

- **event-connect** permits to connect a signal to the text widget (see the Chapter 4)
- **realize-widget** see general documentation on *realize-widget*
- **text-copy-clipboard** copies the content of text clipboard
- **text-cut-clipboard** cuts the selected clipboard
- **text-paste-clipboard** pastes the clipboard in the text widget

Chapter 7. Dialog box widgets

A dialog box is a graphical interface component consisting of a window displayed by a program for:

- inform the user of an event, or
- obtain user information.

7.1. Class <dialog>

A `<dialog>` is a a window split vertically. The top section is the place where widgets such as a label or an entry should be packed. The bottom area is generally used for packing buttons into the dialog which may perform functions such as cancel, ok, or apply.

Inherited classes:	`<window>`
	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	`<font-dialog>`
	`<file-dialog>`
	`<color-dialog>`

Direct slots:	
Direct (non accessor) methods:	container-add! (<dialog> <gtk-widget> . <top>) dialog-run (<dialog>) realize-widget (<dialog> <top>)

Methods

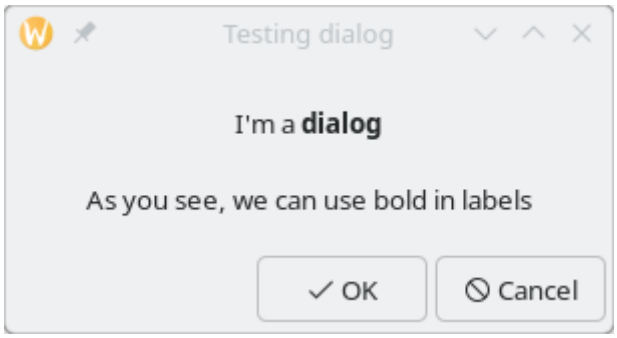
- **container-add!** permits to add a widget to the top area of the dialog window (above the buttons).
- **dialog-run** is a function that must be run to wait the user response. When the user chooses a button, the string associated to this button is returned as result. If the user clicks the close button, the value returned is `#f`.
- **realize-widget** permits to specify at creation time the buttons used by the dialog with the `:buttons` valued parameter. the default buttons are a "OK and " and "Cancel" buttons. The specified value can be the name of a GTK stock item and an icon will be associated to the text. For instance "gtk-ok" will add a button with text "Ok" and a check. So the default value for the `:buttons` parameter is the list `("gtk-ok" "gtk-cancel")`. See official GTK documentation for the possible names for GTK stock items. **Note that the usage of GTK stock items is now deprecated**. You can also see general documentation on *realize-widget*.

Exemple

```
(let ((d (make <dialog> :width 300 :title "Testing dialog")))
  ;; Add two labels to this dialog
  (container-add! d (make <label>
                      :text "nI'm a <b>dialog</b>"))
  (container-add! d (make <label>
                      :text "nAs you see, we can use bold in labelsn"))

  (let ((res (dialog-run d)))
    (if res
        (eprintf "You clicked the button ~sn" res)
        (eprintf "Dialog box was closed by the usern"))
    (destroy d)))
```

Executing the previous code will create the following window:



The make-simple-message-dialog procedure

The `make-simple-message-dialog` permits to easily build dialog window. It takes three mandatory parameters.

1. the title of the dialog window
2. the type of the dialog. It is a symbol and can be `error`, `info password`, `question`, `warning`.
3. the message that must be displayed in above the buttons

The following parameters are also accepted:

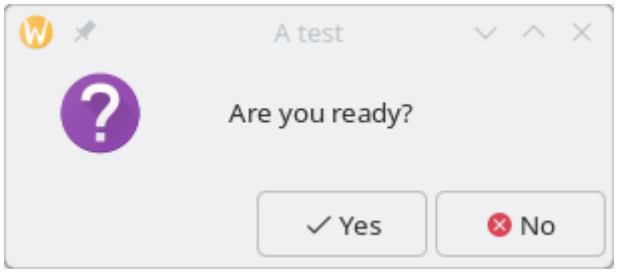
- **buttons** is a list of buttons (by default, this list contain a OK and Cancel buttons).
- **width** the width of the dialog window (default 300)
- **height** the height of the dialog window (default 100)

This procedure also runs the dialog, and waits for the user response. The result is the name of the clicked button (or `#f` is destroyed). See below:

```
(make-simple-message-dialog "A test" 'question "Are you ready?"
                            :buttons '("gtk-yes" "gtk-no"))
```

Executing the previous code will create the following window, and wait until the user responds.

When execution continues, the result will be `"gtk-yes"`, `"gtk-no"` of `#f`



7.2. Class <color-dialog>

A `<color-dialog>` is a specialized dialog box for choosing a color.

Inherited classes:	`<dialog>`
	`<window>`
	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	show-editor value
Direct (non accessor) methods:	dialog-run (<color-dialog>) realize-widget (<color-dialog> <top>)

Slots

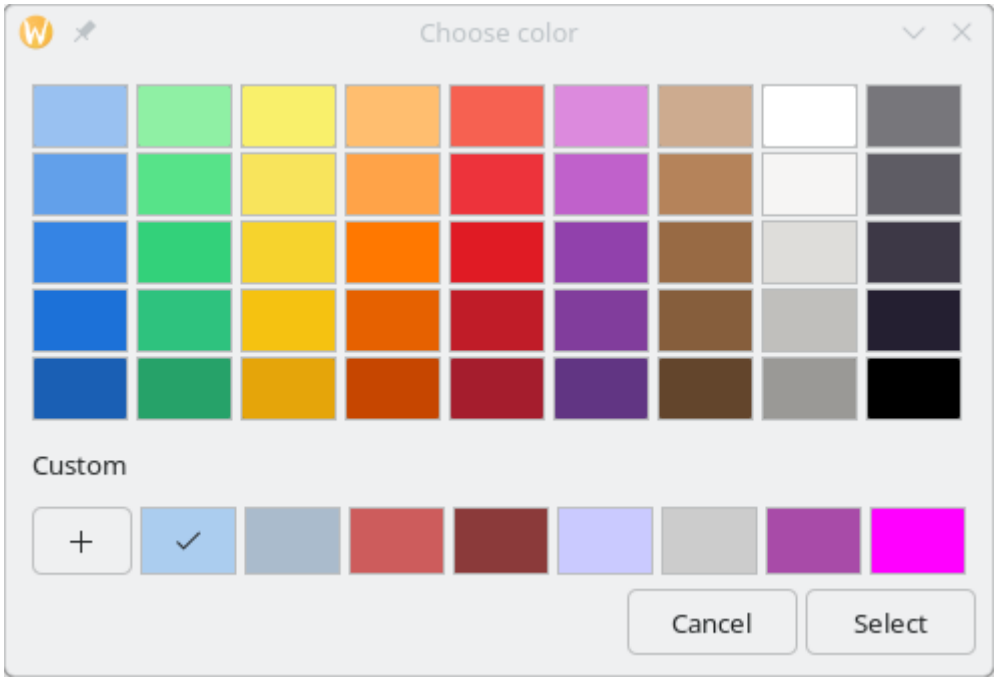
- **show-editor** indicates if the dialog will use a standard palette or if we want to see a color editor (default is `#f`).
- **value** is a starting value used when the dialog is initialized.

Methods

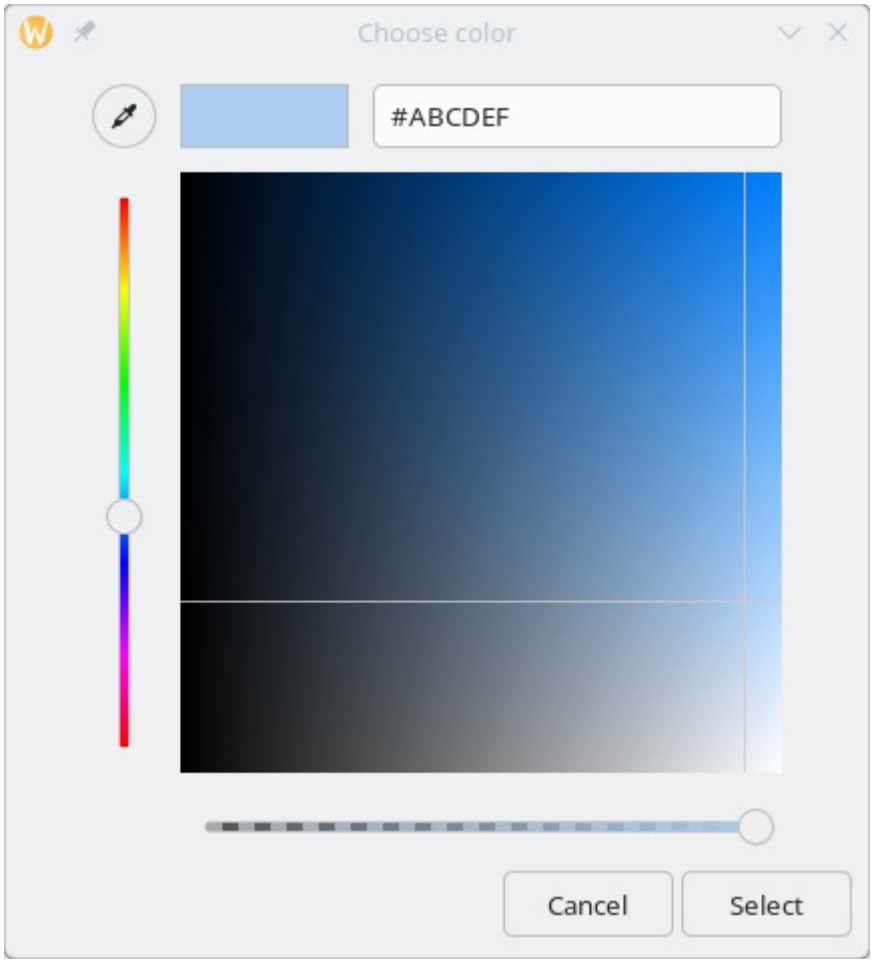
- **dialog-run** run the color dialog (see documentation on *dialog-run* for dialogs). When it terminates it will return the color as a string., or `#f` if cancelled
- **realize-widget** see see documentation on *realize-widget* for dialogs

Example

```
(dialog-run (make <color-dialog> :title "Choose color"
                  :show-editor #f :value "#abcdef"))
```



If `show-editor` is set to `#t` we'll have



7.3. Class <file-dialog>

A `<color-dialog>` is a specialized dialog box for choosing one or several files.

Inherited classes:	`<dialog>`
	`<window>`
	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	dialog-type select-multiple show-hidden value
Direct (non accessor) methods:	dialog-run (<file-dialog>) realize-widget (<file-dialog> <top>)

Slots

- **dialog-type** indicates the reason of file selection. It can be the symbols `open`, `save`, `open-folder`, `create-folder`. The buttons displayed at the bottom of the dialog depends of the dialog type. The default value for this slot is `open`.
- **select-multiple** indicates if multiple selection ar possible
- **show-hidden** indicates if hidden files are shown.
- **value** is the initial chosen file or directory.

Methods

- **dialog-run** permits to run the file dialog (see documentation on *dialog-run* for dialogs).
- **realize-widget** see see documentation on *realize-widget* for dialogs.

7.4. Class <font-dialog>

A `<font-dialog>` is a specialized dialog box for choosing a font.

Inherited classes:	`<dialog>`
	`<window>`
	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	preview-text show-preview-entry value
Direct (non accessor) methods:	dialog-run (<font-dialog>) realize-widget (<font-dialog> <top>)

Slots

- **preview-text** is the text to show in the preview window (by default it is the classical "The quick brown fox jumps over the lazy dog.")
- **show-preview-entry** indicates if the preview entry is displayed or not.
- **value** is the font used for the preview

Methods

- **dialog-run** permits to run the font dialog (see documentation on *dialog-run* for dialogs). The value returned is the selected font as a string, or `#f` if cancelled.
- **realize-widget** see general documentation on *realize-widget*.

Chapter 8. Basic Classes

This section describes the basic classes which are inherited by the high level GTKlos widgets. These classes are not exported by the GTKlos library. However, since the slots (an their accessor function) are available in the library widgets, they are exposed here. Furthermore, all the methods described here are also available in user programs, once the library has been imported.

8.1. Class <gtk-object>

The `<gtk-object>` class is the root of the hierarchy of all the GTK object.

Inherited classes:	
Directly inheriting classes:	`<gtk-canvas-item>` `<gtk-widget>`
Direct slots:	
Direct (non accessor) methods:	event-connect (<gtk-object> <top> <top>) event-connect-after (<gtk-object> <top> <top>)

Methods

- **event-connect** see Events.
- **event-connect-after** see Events.

8.2. Class <gtk-destroyed-object>

. The `<gtk-destroyed-object>` class is the class given to a `<gtk-object>` which has been destroyed with the **destroy** method.

Inherited classes:	
Directly inheriting classes:	
Direct slots:	
Direct (non accessor) methods:	

8.3. Class <gtk-widget>

The `<gtk-widget>` is the base class all widgets in GTK derive from.

Inherited classes:	`<gtk-object>`

Directly inheriting classes:	`<text>`
	`<entry>`
	`<scale>`
	`<progress-bar>`
	`<toolbar-item>`
	`<gtk-container>`
	`<gtk-orientable>`
	`<gtk-misc>`

Direct slots:	can-default can-focus expand focus-on-click has-default has-focus height height-request name parent sensitive show tooltip visible wid width width-request
Direct (non accessor) methods:	container-add! (<grid> <gtk-widget> . <top>) container-add! (<gtk-container> <gtk-widget> . <top>) container-add! (<dialog> <gtk-widget> . <top>) container-add! (<gtk-box> <gtk-widget> . <top>) container-info (<gtk-widget>) container-remove! (<gtk-container> <gtk-widget>) destroy (<gtk-widget>) initialize-instance (<gtk-widget> <top>) internal-arrange-widget (<gtk-widget> <top>) realize-widget (<gtk-widget> <top>)

Slots

- **can-default** specifies whether widget can be a default widget.
- **can-focus** specifies whether widget can own the input focus. _ **expand** indicates if widget is expanded or not (default to #f).
- **focus-on-click** indicates whether the widget should grab focus when it is clicked with the mouse (this slot is only relevant for widgets that can take focus).
- **has-default** indicates if the widget is the default widget.
- **has-focus** indicates if the widget has the focus.
- **height** contains the actual widget height.
- **height-request** contains the requested widget height.
- **name** denotes the name of the widget. The name of the widget allows you to refer to it from a CSS file. See GTK documentation for more information.
- **parent** denotes the parent of the container widget which contain this window. A list can be used when setting the parent of a widget. In this case, the first element of the list must be the container in which the widget must be added, the rest are the parameters that would be used when using the **container-add!** method (see below).
- **sensitive** indicates the if the user can interact with it. If the widget is non sensitive, it is grayed out.
- **show** is a read only slots. It indicates if the widget is shown when realized. The default value of this slot is `#t`.
- **tooltip** is a string that can the text to be used as a tooltip for the created widget.
- **visible** is a boolean to set/unset the visibility of the widget.
- **wid** is a ***STklos*** slot. It contains the low level GTK widget which implements the high level GTKlos object. Its value is generally set in the **realize-widget** method. Normal user program shouldn't change the value of this slot.
- **width** contains the actual widget width.
- **with-request** contains the requested widget width.

Methods

- **container-add!** see general documentation on *container-add!*
- **container-info** returns some information on the way the widget has been added to its container as a list. If the widget has no parent, it returns `#f`.
- **container-remove!** see general documentation on *container-add!*
- **destroy** permits to destroy the widget. When a widget is destroyed, its class is changed to `<destroyed-gtk-object>`.
- **internal-arrange-widget** is a hook called when the widget is initialized. Most of the time it does nothing.
- **realize-widget** is the method called to create a low level GTK widget, and initialize it properly. Each widget has it own `realize-widget`. For `<gtk-widget>`, it does nothing.

8.4. Class <gtk-container>

The `<gtk-container>` class is inherited by all the container widgets.

Inherited classes:	`<gtk-widget>` `<gtk-object>`

Directly inheriting classes:	`<canvas>`
	`<gtk-menu-item>`
	`<gtk-menu-shell>`
	`<combobox>`
	`<button>`
	`<scroll>`
	`<toolbar>`
	`<header-bar>`
	`<grid>`
	`<frame>`
	`<window>`

Direct slots:	border-width children
Direct (non accessor) methods:	container-add! (<gtk-container> <gtk-widget> . <top>) container-remove! (<gtk-container> <gtk-widget>) destroy (<gtk-container>)

The direct methods of this class are described in the section about **<gtk-widget>** class.

Methods

- **container-add!** is the method used to add a widget to a container. Its first argument is the container widget and its second argument is the widget to add to the container. Subsequent parameters depend of the container (each container has its own conventions to add a component to it).
- **container-remove!** permit to remove the widget form it container. The widget is not destroyed.
- **destroy** permits to destroy the widget (and all its children). When a widget is destroyed, its class is changed to `<destroyed-gtk-object>`.

8.5. Class <gtk-misc>

This class factorizes properties which are common between the label and image widgets.

Inherited classes:	`<gtk-widget>` `<gtk-object>`
Directly inheriting classes:	`<label>` `<image>`
Direct slots:	xalign xpad yalign ypad
Direct (non accessor) methods:	

Slots

- **xalign** is the horizontal alignment of the widget, from 0.0 (left) to 1.0 (right).
- **xpad** is the horizontal amount of padding for the widget.
- **yalign** is the vertical alignment of the widget , from 0.0 (top) to 1.0 (bottom).
- **ypad** is the vertical amount of padding for the widget.

8.6. Class <gtk-orientable>

The class `<gtk-orientable>` is inherited by classes which can be horizontally or vertically oriented.

Inherited classes:	`<gtk-widget>` `<gtk-object>`

Directly inheriting classes:	`<separator>`
	`<progress-bar>`
	`<toolbar>`
	`<gtk-box>`

Direct slots:	orientation
Direct (non accessor) methods:	

Method

- **orientation** indicates the orientation of the widget. It's value is a symbol whose value can be `horitontal` or `vertical`.

8.7. Class <gtk-menu-shell>

The class `<gtk-menu-shell>` is a base class. It is the ancestor of the classes `<menu>` and `<menu-bar>`.

Inherited classes:	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	`<menu>` `<menu-bar>`
Direct slots:	
Direct (non accessor) methods:	

8.8. Class <gtk-menu-item>

The class `<gtk-menu-item>` is inherited by menu item classes which can hav an associated callback .

Inherited classes:	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	`<menu-separator-item>` `<menu-item>`
Direct slots:	command
Direct (non accessor) methods:	

Method

- **command** contains the callback associated to the menu item (see Events).

Chapter 9. Canvases

GTKlos permits to access to the GTK canvas widget if the library `GooCanvas` is installed ***STklos*** is configured. `GooCanvas` is a canvas widget for GTK+ that uses the Cairo 2D library for drawing. It has a model/view split, and uses interfaces for canvas items and views.

This documentation is built from the original `GooCanvas` documentation available at the following URL: https://lazka.github.io/pgi-docs/GooCanvas-2.0

9.1. Class <canvas>

Inherited classes:	`<gtk-container>`
	`<gtk-widget>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	automatic-bounds background-color bounds-from-origin bounds-padding canvas-x1 canvas-x2 canvas-y1 canvas-y2 clear-background integer-layout resolution-x resolution-y scale scale-x scale-y
Direct (non accessor) methods:	initialize-instance (<canvas> <top>) realize-widget (<canvas> <top>)

Slots

- **automatic-bounds** indicates if the bounds are automatically calculated based on the bounds of all the items in the canvas.
- **background-color** is a strings that contains the color to use for the canvas background.
- **bounds-from-origin** indicates if the automatic bounds are calculated from the origin.
- **bounds-padding** is a float. It indicates the padding added to the automatic bounds.
- **canvas-x1** is the x coordinate of the left edge of the canvas bounds, in canvas units.

- **canvas-x2** is the x coordinate of the right edge of the canvas bounds, in canvas units.
- **canvas-y1** is the y coordinate of the top edge of the canvas bounds, in canvas units.
- **canvas-y2** is the y coordinate of the bottom edge of the canvas bounds, in canvas units.
- **clear-background** indicates if the background is cleared before the canvas is painted.
- **integer-layout** indicates if all item layout is done to the nearest integer.
- **resolution-x** is the horizontal resolution of the display, in dots per inch
- **resolution-y** is the vertical resolution of the display, in dots per inch
- **scale** is the magnification factor of the canvas.
- **scale-x** is the horizontal magnification factor of the canvas.
- **scale-y** is the vertical magnification factor of the canvas.

Methods

- **initialize-instance** initializes the given canvas.
- **realize-widget** see general documentation on *realize-widget*.

9.2. Canvas Items

Canvas items (rectangles, ellipses, ...) are the components that are displayed in a canvas widget. To add a canvas item in a given canvas, you just need to set the parent of the canvas item to this canvas.

9.2.1. Class <canvas-rectangle>

A `<canvas-rectangle>` represents a rectangle item. Since it is a subclass *<gtk-canvas-item>*, it inherits all of the style properties such as `x`, `y`, `fill-color` or `line-width`.

Inherited classes:	`<gtk-canvas-item-simple>`
	`<gtk-canvas-item>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	radius-x radius-y
Direct (non accessor) methods:	make-canvas-item (<canvas-rectangle> <top>)

Slots

- **radius-x** is the horizontal radius to use for rounded corners.
- **radius-y** is the vertical radius to use for rounded corners

Method

- make-canvas-item see documentation on *make-canvas-item* method

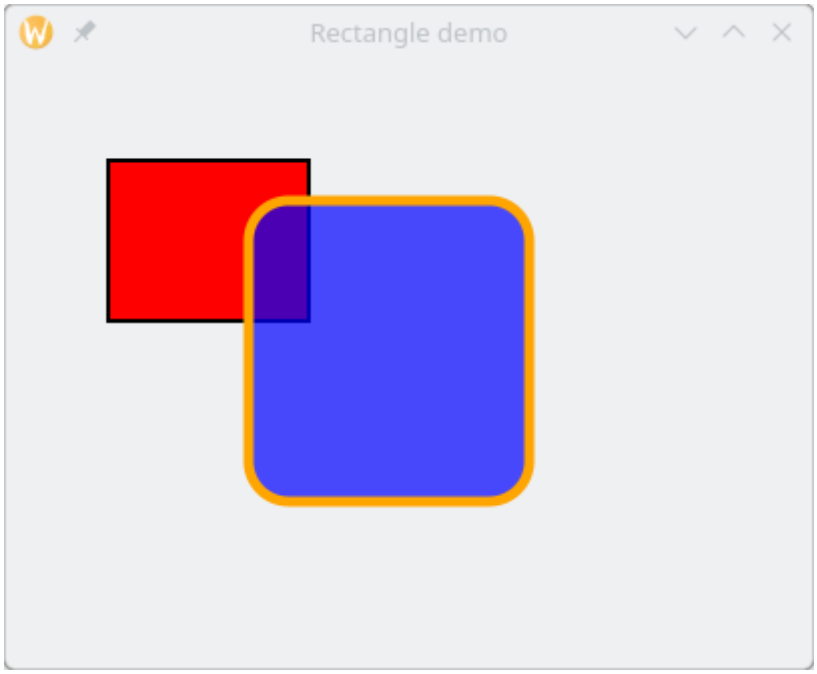
Example

```
(define w (make <vwindow> :title "Rectangle demo"))
(define c (make <canvas> :parent w :width 400 :height 300))

(define r1 (make <canvas-rectangle> :parent c
                 :x 50 :y 50 :width 100 :height 80 :fill-color "red"))

(define r2 (make <canvas-rectangle> :parent c
                 :x 120 :y 70 :width 140 :height 150
                 :stroke-color "orange" :radius-x 20 :radius-y 20
                 :line-width 5
                 :fill-color "rgba(0,0,255,0.7)"))
```

Here, we define two rectangles, a red one `r1` and a (partially transparent) blue one `r2`. We obtain:



Canvas items receive event signals as standard GTK widgets. For instance we can define a behavior when entering in `r1` or when cliking `r2`

```
(event-connect r1 "enter-notify-event"   (lambda ignore (eprintf "Entering r1n")))
(event-connect r2 "button-release-event" (lambda ignore (eprintf "Clicked r2n")))
```

9.2.2. Class <canvas-ellipse>

A <canvas-ellipse> represents an ellipse item.

Inherited classes:	<gtk-canvas-item-simple> <gtk-canvas-item> <gtk-object>
Directly inheriting classes:	
Direct slots:	center-x center-y radius-x radius-y
Direct (non accessor) methods:	make-canvas-item (<canvas-ellipse> <top>)

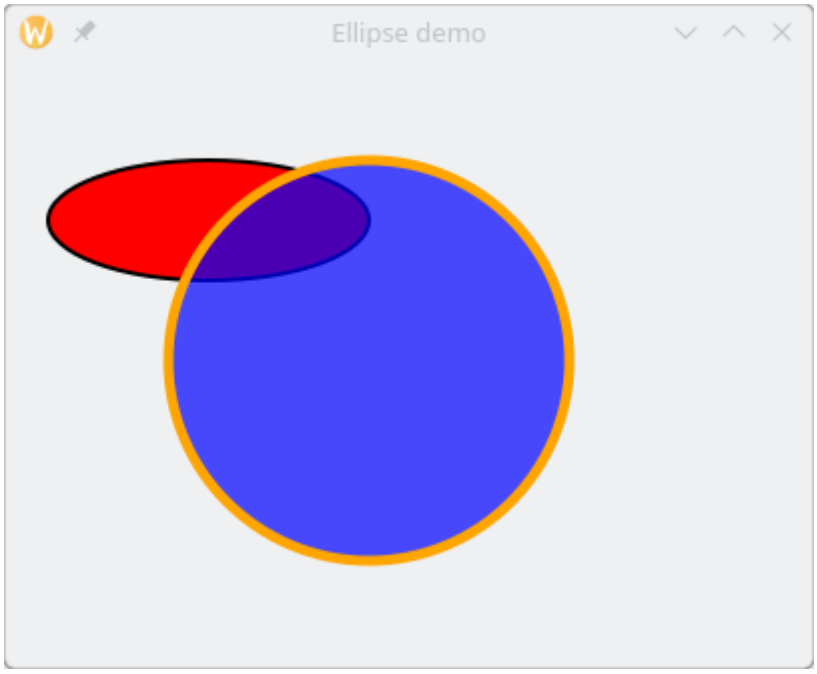
Slots

- **center-x** is the x coordinate of the center of the ellipse
- **center-y** is the y coordinate of the center of the ellipse
- **radius-x** is the horizontal radius of the ellipse
- **radius-y** is the vertical radius of the ellipse

Method

- **make-canvas-item** see documentation on *make-canvas-item* method.

```
(define w (make <vwindow> :title "Ellipse demo"))
(define c (make <canvas> :parent w :width 400 :height 300))
(define e1 (make <canvas-ellipse> :parent c
                 :center-x 100 :center-y 80 :radius-x 80 :radius-y 30
                 :fill-color "red"))
(define e2 (make <canvas-ellipse> :parent c
                 :center-x 180 :center-y 150 :radius-x 100 :radius-y 100
                 :line-width 5 :stroke-color "orange"
                 :fill-color "rgba(0,0,255,0.7)"))
```



9.2.3. Class <canvas-line>

A `<canvas-line>` represents a line item. More exactly,it is a poly-line item which is a series of one or more lines, with optional arrows at either end.

Inherited classes:	`<gtk-canvas-item-simple>`
	`<gtk-canvas-item>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	arrow-length arrow-tip-length arrow-width close-path end-arrow points start-arrow
Direct (non accessor) methods:	make-canvas-item (<canvas-line> <top>)

Slots

- **arrow-length** is a float which represents the length of the arrows, as a multiple of the line width.
- **arrow-tip-length** is a float which represents the length of the arrow tip, as a multiple of the line width.
- **arrow-width** is a float which represents the width of the arrows, as a multiple of the line width.

- **close-path** indicates if the last point should be connected to the first.
- **end-arrow** indicates if an arrow should be displayed at the end of the poly-line
- **points** is the list of points of the poly-line.
- **start-arrow** indicates if an arrow should be displayed at the start of the poly-line.

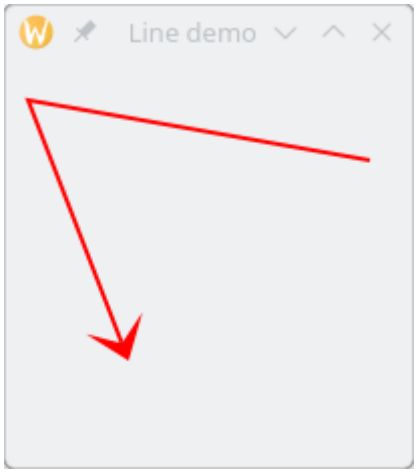
Method

- **make-canvas-item** see documentation on *make-canvas-item* method.

Example

```
(define w (make <vwindow> :title "Line demo"))
(define c (make <canvas> :parent w :width 100 :height 200))

(define l1 (make <canvas-line> :parent c :points '(180 50 10 20 60 150)
                 :stroke-color "red" :end-arrow #t
                 :arrow-length 10 :arrow-tip-length 5 :arrow-width 15))
```



9.2.4. Class <canvas-text>

A `<canvas-text>` represents a text canvas item.

Inherited classes:	`<gtk-canvas-item-simple>`
	`<gtk-canvas-item>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	alignment anchor ellipsize use-markup value wrap
Direct (non accessor) methods:	make-canvas-item (<canvas-text> <top>)

Slots

- **alignment** indicates how to align the text. Its value can be one of the symbols `left`, `center` or `right`.
- **anchor** indicates how to position the text relative to the given `x` and `y` coordinates . Its value can be one of the symbols `center`, `north`, `north-west`, `north-east`, `south`, `south-west`, `south-east`, `west` or `east`.
- **ellipsize** indicates the preferred place to ellipsize the string, if the label does not have enough room to display the entire string. It's value can be one of the symbols `none`, `start`, `middle` or `end`.
- **use-markup** indicates whether we use *PangoMarkup* in the text, to support different styles.
- **value** contains the text to display.
- **wrap** indicates the preferred method of wrapping the string if a width has been set. Its value can be one of the symbols `word`, `char` or `word-char`. `

Method`

- **make-canvas-item** see documentation on *make-canvas-item* method.

9.2.5. Class <canvas-image>

A `<canvas-image>` permits to add an image as a canvas item in a canvas.

Inherited classes:	`<gtk-canvas-item-simple>`
	`<gtk-canvas-item>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	alpha image pixbuf scale-to-fit
Direct (non accessor) methods:	make-canvas-item (<canvas-image> <top>)

Slots

- **alpha** is the opacity of the image (0.0 is fully transparent, and 1.0 is opaque).
- **image** contains the GTKlos image to display (see [class-image])
- **pixbuf** contains the pixbuf to display (see [class-image])
- **scale-to-fit** indicate if the image is scaled to fit the width and height settings.

Method

- **make-canvas-item** see documentation on *make-canvas-item* method.

9.2.6. Class <canvas-path>

A `<canvas-path>` represents a path item, which is a series of one or more lines, bezier curves, or elliptical arcs, using SVG canvas path notation.

Inherited classes:	`<gtk-canvas-item-simple>`
	`<gtk-canvas-item>`
	`<gtk-object>`

Directly inheriting classes:	
Direct slots:	value
Direct (non accessor) methods:	make-canvas-item (<canvas-path> <top>)

Slot

- **value** is the sequence of path commands as a string.

Method

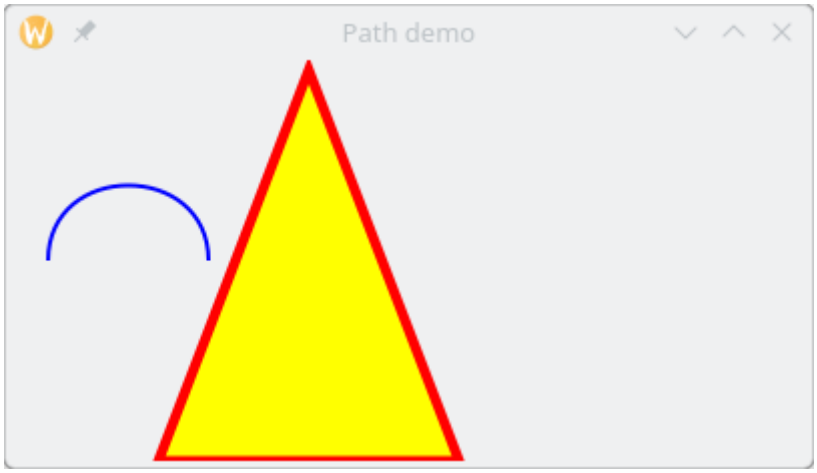
- **make-canvas-item** see documentation on *make-canvas-item* method.

Example

```
(define w (make <vwindow> :title "Path demo"))
(define c (make <canvas> :parent w :width 400 :height 200))

(define p1 (make <canvas-path> :parent c :stroke-color "blue"
                 :value "M20,100 C20,50 100,50 100,100"))

(define p2 (make <canvas-path> :parent c :fill-color "yellow"
                 :stroke-color "red" :line-width 5
                 :value "M150 5 L75 200 L225 200 Z"))
```



9.3. Canvas Base Classes

There are two canvas base classes: `<gtk-canvas-item>` and `<gtk-canvas-item-simple>`. They are not exported by the GTKlos library. However, since the slots (an their accessor function) are available in the library widgets, they are exposed here. Furthermore, all the methods described here are also available in user programs, once the library has been imported.

9.3.1. Class <gtk-canvas-item>

The `<gtk-canvas-item>` is the base class of all the items that can be added in a canvas object.

Inherited classes:	`<gtk-object>`
Directly inheriting classes:	`<gtk-canvas-item-simple>`
Direct slots:	can-focus description parent pointer-events title tooltip visibility visibility-threshold wid
Direct (non accessor) methods:	canvas-item-animate (<gtk-canvas-item> . <top>) canvas-item-lower (<gtk-canvas-item> . <top>) canvas-item-raise (<gtk-canvas-item> . <top>) canvas-item-remove (<gtk-canvas-item>) canvas-item-rotate (<gtk-canvas-item> <top> <top> <top>) canvas-item-scale (<gtk-canvas-item> <top> <top>) canvas-item-stop-animation (<gtk-canvas-item>) canvas-item-translate (<gtk-canvas-item> <top> <top>) initialize-instance (<gtk-canvas-item> <top>) make-canvas-item (<gtk-canvas-item>)

Slots

- **can-focus** indicates if the item can take the keyboard focus.
- **description** contains a description of the item for use by assistive technologies.
- **parent** contains the parent of the canvas item
- **pointer-events** specifies when the item receives pointer events. This is a mask value which can be built with the constants
  - **CANVAS_EVENTS_NONE** the item doesn't receive events at all.
  - **CANVAS_EVENTS_VISIBLE_MASK** a mask indicating that the item only receives *events

when it is visible.

  - **CANVAS_EVENTS_PAINTED_MASK** the item receives events in its painted areas, whether it is visible or not.
  - **CANVAS_EVENTS_FILL_MASK** a mask indicating that the filled part of the item receives events.
  - **CANVAS_EVENTS_STROKE_MASK** a mask indicating that the stroked part of the item receives events.
- **title** contains a short context-rich description of the item for use by assistive technologies.
- **tooltip** contains the tooltip to display for the item.
- **visibility** indicates when the canvas item is visible. Its value can be one of the symbols
  - **hidden** the item is invisible, and is not allocated any space
  - **invisible** the item is invisible, but it is still allocated space
  - **visible** the item is visible.
  - **visible-above-threshold** the item is visible when the canvas scale setting is greater than or equal to the item's visibility threshold setting.
- **visibility-threshold** contains the scale threshold at which the item becomes visible
- **wid** is a ***STklos*** slot. It contains ta pointer to the GTK object used to implement the canvas item . Its value is generally set in the *make-canvas-item* method. Normal user program shouldn't change the value of this slot.

Methods

- **canvas-item-animate** animates an item from its current position to the given offsets, scale and rotation. This method takes several keyword arguments:
  - **x** (default 0.0): the final x coordinate.
  - **y**: (default 0.0): the final y coordinate.
  - **scale** (default 1.0): the final scale.
  - **degrees** (default: 360.0) the final rotation. This can be negative to rotate anticlockwise, and can also be greater than 360 to rotate a number of times.
  - **absolute** (default `#t`) indicates if the x, y, scale and degrees values are absolute, or relative to the current transform.
  - **duration** (default 1000): the duration of the animation, in milliseconds.
  - **step-time** (default 1): the time between each animation step, in milliseconds.
  - **animation-type** (default `freeze`) specifies what happens when the animation finishes. This value may be one of the symbols
    - `freeze`: the item remains in the final position,
    - `reset`: the item is moved back to the initial position,
    - `restart`: the animation is restarted from the initial position.
    - `bounce`: the animation bounces back and forth between the start and end positions.

- **canvas-item-lower** lowers the given item in the stacking order.
- **canvas-item-raise** raises the given item in the stacking order.
- **canvas-item-remove** removes the item from its parent.
- **canvas-item-rotate** rotates the item's coordinate system by the given amount, about the given origin. This method takes four for parameters:
  - the item to rotate
  - **degrees** (a float) which is the clockwise angle rotation
  - **cx** (a float) which is the `x` coordinate of the origin of the rotation.
  - **cy** (a float) which is the `y` coordinate of the origin of the rotation.
- **canvas-item-scale** scales the item's coordinate system by the given amounts. This method takes three for parameters:
  - the item to rotate
  - `sx` (a float) is the amount to scale the horizontal axis.
  - `sy` (a float) is the amount to scale the vertical axis.
- **canvas-item-stop-animation** stops any current animation for the given item, leaving it at its current position.
- **canvas-item-translate** translates the origin of the item's coordinate system by the given amounts. This method takes three for parameters:
  - the item to translate
  - `tx` (a float) is the amount to move the origin in the horizontal direction
  - `ty` (a float) is the amount to move the origin in the vertical direction
- **initialize-instance** verifies that a parent is given, and calls the *make-canvas-item* method.à
- **make-canvas-item** creates the GTK canvas item object. This method is redefined for each GTKlos descendant of the class `<gtk-canvas-item>`.

9.3.2. Class <gtk-canvas-item-simple>

Inherited classes:	`<gtk-canvas-item>` `<gtk-object>`

Directly inheriting classes:	`<canvas-image>`
	`<canvas-path>`
	`<canvas-line>`
	`<canvas-text>`
	`<canvas-rectangle>`
	`<canvas-ellipse>`

Direct slots:	antialias clip-fill-rule clip-path fill-color fill-rule font height hint-metrics line-cap line-join line-join-miter-limit line-width operator stroke-color width x y
Direct (non accessor) methods:	

Slots

- **antialias** indicates the anti-aliasing mode to use. The value is one of the symbols `default`, `none`, `gray` or `subpixel`.
- **clip-fill-rule** is the fill rule used to determine which parts of the item are clipped. It's value is either the symbol `winding` or `even-odd` (see the cairo.FillRule for more information).
- **clip-path** is a string representing the sequence of path commands specifying the clip path.
- **fill-color** is he color to use to paint the interior of the item
- **fill-rule** is the fill rule used to determine which parts of the item are filled (see the cairo.FillRule for more information)
- **font** is the base font to use for the text
- **height** is the height of the object in pixels.

- **hint-metrics** is the hinting to be used for font metrics.its value is one of the symbols `default`, `on` or `off` (see the cairo.HintMetrics for more information).
- **line-cap** is the the line cap style to use. Its value is one of the symbols `butt`, `round` or `square`(see cairo.LineCap for more information.
- **line-join** is the line join style to use. Its value is one of the symbols `mitter`, `round` or `bevel` (see cairo.LineJoin for more information.
- **line-join-miter-limit** is a float representing the smallest angle to use with miter joins, in degrees. Bevel joins will be used below this limit
- **line-width** is a float representing the line width to use for the item's perimeter.
- **operator** is the compositing operator to use. Its value is one of the symbols `clear`, `source`, `dest-atop`, `xor`, `add`, `saturate`, `over`, `in`, `out`, `atop`, `dest`, `dest-over`, `dest-in` or `dest-out` (see caito.Operator for more information).
- **stroke-color** is the color to use for the item's perimeter
- **width** is the width of the object in pixels.
- **x** is the x coordinate of the canvas item
- **y** is the y coordinate of the canvas item

Chapter 10. Misc. functions

This part of the document describes the public functions which have not detailed before and are exported by the `(stklos gtklos)` library.

==== timeout

The function `timeout` permits to add a callback which is called periodically. The callback is integrated in the GTK event loop. It takes two parameters:

1. a delay (in millisecond) which is the period of the repetition
2. a thumb procedure which is the callback. If this procedure returns `#f` the callback is destroyed. Otherwise, it will be called after the given delay.

This function returns an integer which is the event-loop ID of the callback. It can be used to destroy a callback with the function `kill-idle-callback` (see below).

==== timeout-seconds

The function `timeout` permits to add a callback which is called periodically. The callback is integrated in the GTK event loop. This function is preferred if you want to have a timer in the "seconds" range and do not care about the exact time of the first call of the timer. It allows for more optimizations and more efficient system power usage. It takes two parameters:

1. a delay (in second) which is the period of the repetition
2. a thumb procedure which is the callback. If this procedure returns `#f` the callback is destroyed. Otherwise, it will be called after the given delay.

This function returns an integer which is the event-loop ID of the callback. It can be used to destroy a callback with the function `kill-idle-callback` (see below).

==== when-idle

The function `when-idle` takes one parameter (a function without parameter) to be called whenever there are no higher priority events pending to the default main loop. If this function returns `#t` it is automatically removed from the list of event sources and will not be called again.

This function returns an integer which is the event-loop ID of the callback. It can be used to destroy a callback with the function `kill-idle-callback` (see below).

==== kill-idle-callback

The `kill-idle-callback` takes one parameter: the callback ID to remove form the event loop (a callback ID, is the value returned by a function such as `timeout` or `when-idle`).

==== gtk-main

The function `gtk-main` run the main event loop of GTK. This function should be called at the end of a GUI program to take into account the signals and event of this program and, consequently, interact with the user.

==== start-interactive-gtk

This function is useful when developing an application in the ***STklos*** REPL. This function tries to connect the REPL to the GTK main event loop. Hence, when your keyboard is idle, GTK event are processed.

Index