Tutorial for beginners (version 2016/06/02)

By following these tutorials you will learn the basics of GUI programming using GTK+ in Perl.

If you have never programmed before, or are not familiar with the concepts of object oriented programming, you may need to learn a few basics first. The book "Perl in 21 Tagen", "Learning Perl" or a Perl Tutorial may be a better place for you to start. Once you master the basics, be sure to come back and check out these tutorials.

To run the code samples

To run the code samples in the tutorial:

1. Type or copy and paste the code into a file, and save the file with a name like filename.pl
2. To execute the code, type in the terminal:

```
perl filename.pl
```

After executing the code, you will either see the widget on your screen, or (if you have mistyped some of the code) you will see an error message that will help you identify the problem.

A path through the widgets

The tutorial will walk you through increasingly complex examples and programming theory, but you can also feel free to just go directly to the tutorial that is most helpful to you.

Tutorial

1. Basic windows

1.1. Window

2. Images and labels

2.1. Image

2.2. Strings

2.3. Label

3. Introduction to properties

3.1. Properties

4. Grid, separator and scrolling

4.1. Grid

4.2. Separator

4.3. ScrolledWindow

4.4. Paned

5. Signals, callbacks and buttons

5.1. Signals and callbacks

5.2. Button

5.3. LinkButton

5.4. CheckButton

5.5. ToggleButton

5.6. Switch

5.7. RadioButton

6. ButtonBox

6.1. ButtonBox

7. Other display widgets

7.1. Statusbar

7.2. Spinner

7.3. ProgressBar

8. Entry widgets

8.1. SpinButton

8.2. Entry

8.3. Scale

9. A widget to write and display text

9.1. TextView

10. Dialogs

10.1. Dialog

10.2. AboutDialog

10.3. MessageDialog

11. Menus, Toolbars and Tooltips (also: using Glade and GtkBuilder)

11.1. Menu

11.2. MenuButton

11.3. Toolbar

11.4. Tooltip **(to do)**

11.5. Toolbar created using Glade

11.6. MenuBar created using XML and GtkBuilder

12. Selectors

12.1. ColorButton ***(to do)***

12.2. FontChooserWidget ***(to do)***

12.3. FileChooserDialog

13. TreeViews and ComboBoxes (using the M/V/C design)

13.1. ComboBox (one column)

13.2. Simple TreeView with ListStore

13.3. Simpler TreeView with TreeStore

13.4. The Model/View/Controller design ***(to do)***

13.5. ComboBox (two columns)

13.6. More Complex Treeview with ListStore

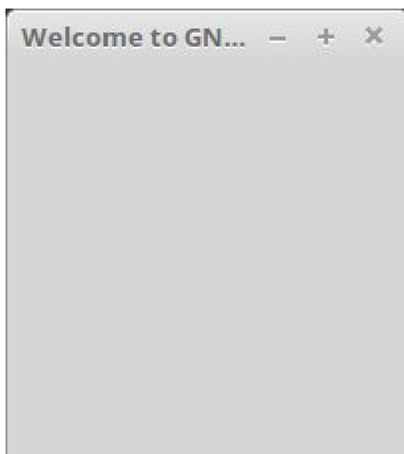
13.7. More Complex TreeView with TreeStore

14. Custom widget

14.1. Widget

1. Basic Windows

1. 1. Window



This GtkApplication displays an image file from the current directory

Code used to generate this example

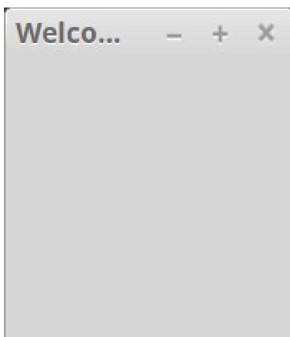
```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;
```

```
# create a Gtk Window
my $window = Gtk3::Window->new('toplevel');

# set the title
$window->set_title('Welcome to GNOME')

# exit the program if user close the Application
$window->signal_connect('delete_event'=>sub{Gtk3->main_quit()});

# show the window
$window->show_all;

# create and run the application
Gtk3->main();
```

Useful methods for a Window widget

- *set_default_size(200, 100)* sets the default size of the window to a width of 200 and a height of 100; if instead of a positive number we pass -1 we have the default size.
- *set_position("center")* centers the window. Other options are *"none"*, *"mouse"*, *"center-always"*, *"center-on-parent"*.

1. 2. Application Window[1]



The simplest GtkApplication Window which can support Gmenu.

NOTE: The structure of a GtkApplication program is different than that of a "Gtk3 mainloop"-style Application. You should't call Gtk3::init - that is handled automatically. All of the program initialization is supposed to be done inside of the 'startup' and 'activate' callbacks. Instead of calling Gtk3::main and Gtk3::quit, you call theGApplication 'run' and 'quit' methods.

Therefore we need to import **Glib::IO**, a per binding for the Gio API. There are two

1 This chapter was made possible alone by Jeremy Volkening who explained me creating a Gtk3::ApplicationWindow. He has also written the code of this example! Thank you very much for that, Jeremy Volkening!

ways:

1) We can make a bindinding to the Glib::IO API manually in our Perl program with the following code at the begin of our script:

```
BEGIN {
use Glib::Object::Introspection;
Glib::Object::Introspection->setup(
basename => 'Gio',
version => '2.0',
package => 'Glib::IO');
}
```

This is the way we want to go in this Tutorial.

2) Alternatively you can find an early implementation as a Perl module on https://git.gnome.org/browse/perl-Glib-IO (not yet published on CPAN!). After downloading you can just copy the file *lib/Glib/IO.pm* to *directory_of_your script/Glib/IO.pm* or install the module system-wide with the following commands:

```
perl ./Makefile.PL
make
make install
```

After installing you have to load the Glib::IO module as usual with

```
use Glib::IO;
```

Hopefully this module simplifies the use of the Gio API in the future (for example the converting of the bytes in a bytestring, when you read a file or write to a file with the Gio API [see the problem in the FileChooserDialog example], could be in later versions of the module redundant).

Code used to generate this example

```
#!/usr/bin/perl

# Make a binding to the Gio API in the Perl program (just copy&paste ;-))
# This is necessary mainly for Gtk3::Application
# Alternatively you find an early implementation as a Perl module
# on https://git.gnome.org/browse/perl-Glib-IO (not yet published on CPAN!)
# Hopefully this module simplifies the use of the Gio API in the future
# (see also the notes above).
BEGIN {
  use Glib::Object::Introspection;
  Glib::Object::Introspection->setup(
    basename => 'Gio',
    version => '2.0',
    package => 'Glib::IO');
}

use strict;
```

```
use warnings;

use Gtk3;
use Glib qw/TRUE FALSE/;

# The MAIN FUNCTION should be as small as possible and do almost nothing except creating
# your Gtk3::Application and running it
# The "real work" should always be done in response to the signals fired by Gtk3::Application.
# see below
my $app = Gtk3::Application->new('app.test', 'flags-none');

$app->signal_connect('startup'  => \&_init     );
$app->signal_connect('activate' => \&_build_ui );
$app->signal_connect('shutdown' => \&_cleanup  );

$app->run(\@ARGV);

exit;


# The CALLBACK FUNCTIONS to the SIGNALS fired by the main function.
# Here we do the "real work" (see above)
sub _init {
	my ($app) = @_;

	# Handle program initialization
	print "Hello world!\n";

 }

sub _build_ui {

	my ($app) = @_;

	my $window = Gtk3::ApplicationWindow->new($app);
	$window->set_title ('Welcome to GNOME');
	$window->set_default_size (200, 200);
	$window->signal_connect( 'delete_event' => sub {$app->quit()} );
	$window->show();

}

sub _cleanup {

	my ($app) = @_;

	# Handle cleanup
	print "Goodbye world!\n";

}
```

Useful methods for a Gtk3::ApplicationWindows

- *set_default_size(200, 100)* sets the default size of the window to a width of 200 and a height of 100; if instead of a positive number we pass -1 we have the default size.
- *set_position("center")* centers the window. Other options are *"none"*, *"mouse"*, *"center-always"*, *"center-on-parent"*.

For more details on creating and running Gtk3::Applications I highly recommend to read the most useful explanations about this on https://wiki.gnome.org/HowDoI/GtkApplication!

Whereas the Python Tutorial, on which this tutorial is based, for the below presented widget always uses a Gtk3::Application Program, we want to use in the following normally a "main-loop"-style Application. We want to use the the Gtk3::Application interface only if it is really necessary (i.e. especially if the Application has a Menu)!

The reasons for this decission are:

1) a practical: Jeremy Volkening explained me creating a Gtk3::ApplicationWindow, only after the most Tutorials was already written.

2) The above written "difficulties" with the non CPAN module Glib::IO

3) I think binding the Glib::IO API and building a seperate class for Gtk3::ApplicationWindow are for the mostly simple examples a little bit overdone and make them too complicated. The features of the specific element can be explained more clearly in the "main-loop"-Style

But you can easily translate the "mainloop"-style Application in a Gtk3::Application program with the following steps:

1) Implement a binding to the Gio API in the perl Program as described above.

2) Create a Gtk3::ApplicationWindows class to build and show the Gtk3::ApplicationWindows an its content as follows. Here you have to put the main content of the specific Tutorial chapter. But note to use *"$app->quit"* instead of *"Gtk3->main_quit"* and import Gtk3 without the -init argument!

```
package MyWindow;
use strict;
use warnings;
use Gtk3;
use Glib qw/TRUE FALSE/;
# Our class must be a subclass of Gtk3::ApplicationWindow to inherit
# the methods, properties etc.pp. of Gtk3::ApplicationWindow
use base 'Gtk3::ApplicationWindow' ;

sub new {
	my ($window, $app) = @_;
	$window = bless Gtk3::ApplicationWindow->new($app);
	$window->set_title ("MenuButton Example");
	$window->set_default_size(600,400);
	$window->signal_connect( "delete_event" => sub {$app->quit()} );
```

```
# Here add the widgets how it is described in the specific chapter
[...]
}

# Define also the callback functions as described in the chapter
sub on_click_cb { [...] }
```

3) Last create the Gtk3::Application and run it in your main class. Againg don't call Gtk3::init and use *"$app->run;"* instead of *"Gtk3->main();"*. Usually the main part looks like this:

```
package main;
use strict;
use warnings;

# NO -init!!!
use Gtk3;
use Glib qw/TRUE FALSE/;

# Creating and run the Gtk3::Application
my $app = Gtk3::Application->new('app.id', 'flags-none');

$app->signal_connect('startup'  => \&_init     );
$app->signal_connect('activate' => \&_build_ui );
$app->signal_connect('shutdown' => \&_cleanup  );

$app->run(\@ARGV);

exit;

# The CALLBACK FUNCTIONS to the SIGNALS fired by the main function.
sub _init {
	my ($app) = @_;

	# Handle program initialization
	print "Hello world!\n";
}

sub _build_ui {
	my ($app) = @_;
	my $window = MyWindow->new($app);
	$window->show_all();
}

sub _cleanup {
	my ($app) = @_;

	# Handle cleanup
	print "Goodbye world!\n";
}
```

2. Images and Labels

2. 1. Image

This GtkApplication displays an image file from the current directory.

If the image file is not loaded successfully, the image will contain a "broken image" icon. filename.png needs to be in the current directory for this code to work.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;


# create a window
my $window = Gtk3::Window->new('toplevel');
$window->set_title('Welcome to GNOME');
$window->set_default_size(300,300);
$window->signal_connect('delete_event'=>sub{Gtk3->main_quit()});

# create an image
my $image = Gtk3::Image->new();

# set the content of the image as the file filename.png
$image->set_from_file('gnome-image.png');

# add the image to the window
$window->add($image);
```

```
# show the window and run the app
$window->show_all;
Gtk3->main();
```

Useful methods for an Image widget

- To load an image over a network use *set_from_pixbuf(pixbuf)*, where pixbuf is a GdkPixbuf:

```
#!/usr/bin/perl
use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;

# create a window
[… see below!]

# create a pixbuf from file filename='gnome-image.png', with width=32
# and height=64 amd boolean preserve_aspect_ratio=False.
my $pixbuf = Gtk3::Gdk::Pixbuf->new_from_file_at_scale('gnome-image.png', 64, 128,
FALSE);

# create an image
my $image = Gtk3::Image->new();

# set the content of the image as the file filename.png
$image->set_from_pixbuf($pixbuf);

# add the image to the window
$window->add($image);

[… see below!]
```

  If *preserve_aspect_ratio=TRUE* we can use *new_from_file_at_size(filename, width, height)*. If *width* or *height* is -1, it is not constrained.

  For loading from an input stream, see *new_from_stream()* and *new_from_stream_at_scale()* in the documentation.

2. 2. Strings

[***python specific – to do!]***

GTK+ uses UTF-8 encoded strings for all text. If umlauts (ä, ö, ü, ß etc) are displayed wrongly, you have to set the pragma "*use utf8;"* at the beginning of the script. If umlauts are nevertheless wrongly displayed, you possibly have to decode the string, which shall be displayed, to the utf8 format with *Encode::decode('utf-8', $string)* or better open already a inputfile with the option '<:encoding(UTF-8)'!

If you also want to display something on the terminal (e.g. with the *"print"*-methode), you have to set the "Line discipline" of the standard output onto the UTF 8 mode. This prevent that the terminal

will not try to convert the string again to Latin-1. See a example of the problem in the chapter about the toolbar.

```
# set the pragma utf8 to prevent that umlauts of in the script created strings are displayed wrongly
use utf8;
# set the "Line Discipline" of the standard output into the UTF 8 Mode.  Thereby the terminal
# doesn't try to convert the string again to Latin-1
binmode STDOUT, ':utf8';
```

2. 4. Label

A simple label

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;

# create the window
my $window = Gtk3::Window->new('toplevel');
$window->set_title('Welcome to GNOME');
$window->set_default_size(200,100);
$window->signal_connect('delete_event'=>sub{Gtk3->main_quit()});

# create a label
my $label = Gtk3::Label->new();

# set the text of the label
$label -> set_text('Hello GNOME!');

# add the label to the window
$window->add($label);

# show the window and run the application
$window->show_all;
Gtk3->main();
```

Uselful methods for a Label widget

An explanation of how to work with strings in GTK+ can be found in the chapter "Strings" (see below).

- *set_line_wrap(TRUE)* breaks lines if the text of the label exceeds the size of the widget.
- *set_justify("left")* (or "*right*", "*center*", "*fill*")) sets the alignment of the lines in the text of the label relative to each other. The method has no effect on a single-line label.
- For decorated text we can use *set_markup('text')*, where *'text'* is a text in the Pango Markup Language. An example:[2]

```
$label->set_markup  ('Text can be <small>small</small>, <big>big</big>,'.
                     '<b>bold</b>, <i>italic</i> and even point to somewhere'.
                     'in the <a href=\"http:://www.gtk.org\" '.
                     'title=\"Click to find out more\">internets</a>');
```

- To break lines if text exceeds the widget's size, you have to set line wrap to "TRUE" with

```
$label->set_line_wrap(TRUE);
```

3. Properties

Overview

Properties describe the configuration and state of widgets. Each widget has its own particular set of properties. For example, a widget such as a button has the property "label" which contains the text of the widget. You can specify the name and value of any number of properties by using the method associated with it. For example, to create a label with the text "Hello World", an angle of 25 degrees, and aligned to the right, you can use:

```
My $label = Gtk3::Label-new();
$label->set_label('Hello World');
$label->set_angle(25);
$label->set_halign(Gtk.Align.END);
```

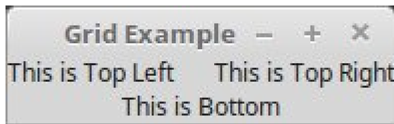
Once you have created such a label, you can get the text of the label with $label->get_label(), and analogously for the other properties.

Instead of using getters and setters you can also get and set the properties with get_property('*prop-*name') and set_property('*prop-name*', *value*), respectively.

2 Note: At line-breaks inside quotation marks the lines are wraped in the label, too. To prohibit this while at the same time keeping legibility inside the code, the quotation marks are closed and connected to the operator ".".

4. Grid, separator and scrolling

4. 1. Grid



Some labels in a grid

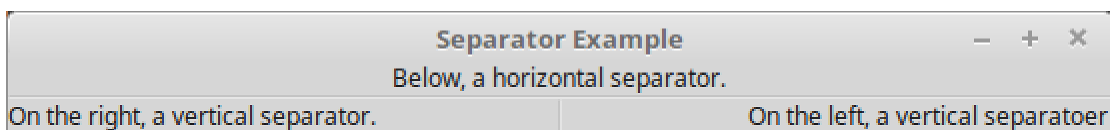
Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;

my $window = Gtk3::Window->new('toplevel');
$window->set_title('Grid Example');
$window->signal_connect('delete_event' => sub {Gtk3->main_quit});

# three labels
my $label_top_left = Gtk3::Label->new('This is Top Left');
my $label_top_right = Gtk3::Label->new('This is Top Right');
my $label_bottom = Gtk3::Label->new('This is Bottom');

# a grid
my $grid = Gtk3::Grid->new();

# some space between the columns of the grid
$grid->set_column_spacing(20);

# in the grid:
# attach the first label in the top left corner
$grid->attach($label_top_left,0,0,1,1);

# attach the second label
$grid->attach($label_top_right,1,0,1,1);

# attach the third label below the first label
$grid->attach_next_to($label_bottom, $label_top_left, 'bottom', 2, 1);

# add the grid to the window
$window->add($grid);

# show the window and run the Application
$window->show_all;
Gtk3->main();
```

Useful methods for a Grid

- To attach a widget *child* in position *left*, *top* in a slot of given *width*, *height* use *attach*(*child*, *left*, top, *width*, *height*). If a widget sibling is already in place, we can also use *attach_next_to(child, sibling, side, width, height)*, where side is one of *"left"*, *"right"*, *"top"*, *"bottom"*.
- *insert_row(position)* and *insert_column(position)* do exactly what they say; children which are attached at or below this position are moved one row down, and children which span across this position are grown to span the new row. *insert_next_to(sibling, side)* inserts a row or column at the specified position. The new row or column is placed next to *sibling*, on the side determined by side; if side is *"top"* or *"bottom"*, a row is inserted, if side is *"left"* or *"right"*, a column is inserted.
- *set_row_homogeneous(TRUE)* and *set_column_homogeneous(TRUE)* ensure that (respectively) every row or every column has the same width or height.
- *set_row_spacing(spacing)* and *set_column_spacing(spacing)* force a spacing between (respectively) rows or columns. The value of *spacing* can be between 0, which is the default value, and 32767.

4. 2. Separator



A horizontal and a vertical separator divide some labels.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;

my $window = Gtk3::Window->new('toplevel');
$window->set_title ('Separator Example');
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# three labels
my $label1 = Gtk3::Label->new();
$label1->set_text('Below, a horizontal separator.');

my $label2 = Gtk3::Label->new();
$label2->set_text('On the right, a vertical separator.');

my $label3 = Gtk3::Label->new();
$label3->set_text('On the left, a vertical separatoer');
```

```
# a horizotal separator
my $hseparator = Gtk3::Separator->new('horizontal');

# a vertical separator
my $vseparator = Gtk3::Separator->new('vertical');

# a grid to attach labels and separators
my $grid = Gtk3::Grid->new();
$grid -> attach ($label1, 0, 0, 3, 1);
$grid -> attach ($hseparator, 0, 1, 3, 1);
$grid -> attach ($label2, 0, 2, 1, 1);
$grid -> attach ($vseparator, 1, 2, 1, 1);
$grid -> attach ($label3, 2, 2, 1, 1);
$grid -> set_column_homogeneous(TRUE);

# add the grid to the window
$window -> add($grid);

# show the window and run the Application
$window -> show_all();
Gtk3->main();
```

4. 3 ScrolledWindow

An image in a scrolled window.

Code used to generate this example

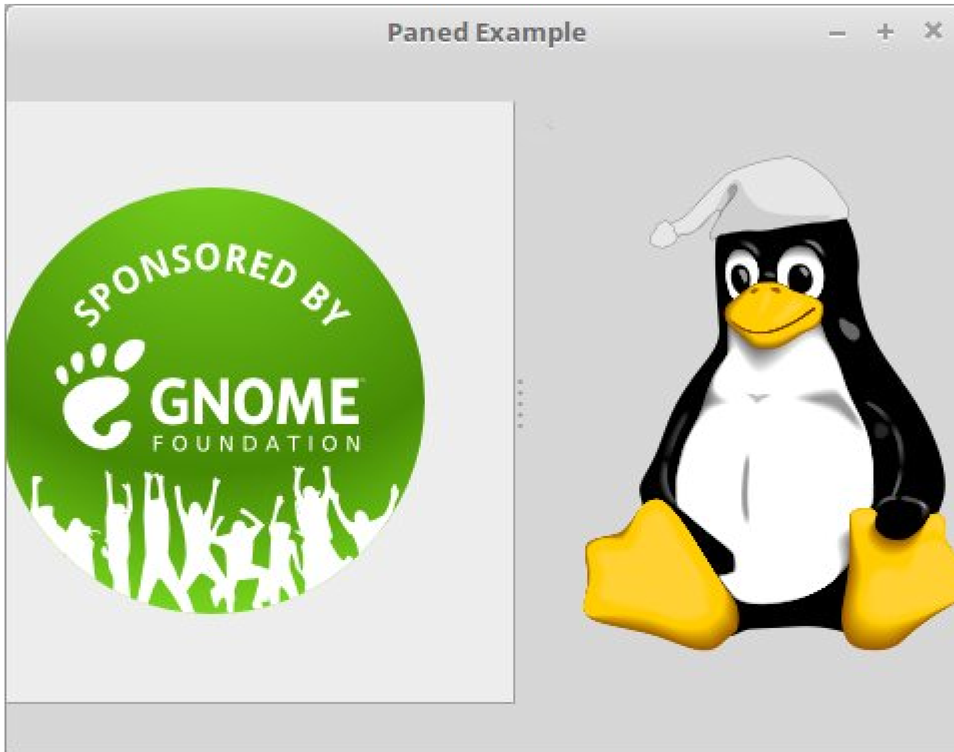
```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;

my $window=Gtk3::Window->new('toplevel');
$window->set_title('Scrolled Window Example');
$window->set_default_size(200,200);
```

```
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# the scrolled window
my $scrolled_window=Gtk3::ScrolledWindow->new();
$scrolled_window->set_border_width(10);

# there is always the scrollbar (otherwise: automatic - only if needed - or never
$scrolled_window->set_policy('always', 'always');

# an image - slightly arger than the window...
my $image = Gtk3::Image->new();
$image->set_from_file('gnome-image.png');

# add the image to the scrolled window
$scrolled_window->add_with_viewport($image);

# add the scrolled window to the window
$window->add($scrolled_window);

# show window and start MainLoop
$window->show_all();
Gtk3->main;
```

Useful methods for a ScrolledWindow widget

- *set_policy(hscrollbar_policy, vscrollbar_policy)* where each of the arguments is one of *"automatic"*, *"always"*, *"never"* regulates whether the horizontal and vertical scrollbars should appear: with *"automatic"* they appear only if needed, *"always"* and *"never"* are self-explanatory.
- *add_with_viewport($widget)* is used to add the Gtk3::Widget $widget without native scrolling capabilities inside the window.
- *set_placement(window_placement)* sets the placement of the contents with respect to the scrollbars for the scrolled window. The options for the argument are "*top-left"* (default: the scrollbars are on the bottom and on the right of the window), *"top-right"*, *"bottom-left"*, *"bottom-right"*.
- *set_hadjustment($adjustment)* and *set_vadjustment($adjustment)* set the Gtk3::Adjustment $*adjustment*. This is the representation of a value with a lower and upper bound, together with step and page increments, and a page size, and it is constructed as *"my $adjustment=Gtk3::Adjustment->new(initial value, minimum value, maximum value, step_increment, page_increment, page_size)"* where the fields are of type float. (Note that step_increment is not used in this case, it can be set to 0.)

4. 4 Paned



Two images in two adjustable panes, horizontally aligned

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;

my $window = Gtk3::Window->new('toplevel');
$window->set_title ('Paned Example');
$window->set_default_size(450, 350);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# a new widget with two adjustable panes
# on on the left and one on the right
my $paned = Gtk3::Paned->new('horizontal');

# two images
my $image1 = Gtk3::Image->new();
$image1 -> set_from_file('gnome-image.png');

my $image2 = Gtk3::Image->new();
$image2 -> set_from_file('tux.png');

# add the first image to the left pane
$paned -> add1($image1);
```

```
# add the second image to the right pane
$paned -> add2($image2);

# add the panes to the window
$window -> add($paned);

# show the window and run the Application
$window -> show_all();
Gtk3->main();
```

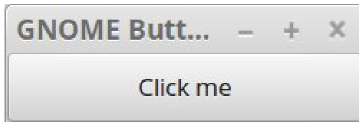
Useful methods for a Paned widget

To have two vertically aligned panes, use *"vertical"* instead of *"horizontal"*. The method *add1($widget1)* will add the $*widget1* to the top pane, and *add2($widget2)* will add the $*widget2* to the bottom pane.

5. Signals, callbacks and buttons

5. 1. Signals and callbacks (to do)

5. 2. Button



A button widget connected to a simple callback function.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;

# a window
my $window = Gtk3::Window->new('toplevel');
$window->set_title ('GNOME Button');
$window->set_default_size(250,50);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# a button
my $button = Gtk3::Button->new();
# with a label
$button->set_label('Click me');

# connect the signal 'clicked' emitted by the button
# to the callback function do_clicked
$button->signal_connect('clicked' => \&do_clicked);

# add the button to the window
$window -> add ($button);

# show the window and run the Application
$window -> show_all();
Gtk3->main();

# callback function connected to the signal 'clicked' of the button
sub do_clicked {
	print("You clicked me! \n");
	}
```

Useful methods for a Button widget

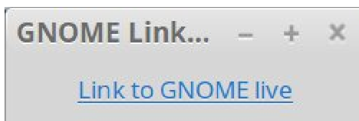
In line 20 the 'clicked' signal from the button is connected to the callback function do_clicked() using *$widget->signal_connect(signal, callback function)*. See Signals and callbacks for a more detailed explanation.

- *set_relief("none")* sets to none the relief style of the edges of the Gtk.Button - as opposed to *"normal"*.
- If the label of the button is a stock icon, *set_use_stock(TRUE)* sets the label as the name of the corresponding stock icon.
- To set an image (e.g. a stock image) for the button button:

```
my $image = Gtk3::Image->new();
# USAGE: set_from_stock(stock-id, Gtk3::IconSize[3]);
$image -> set_from_stock("gtk-about", "4");
$button-> set_image($image);
```

- If we use *set_focus_on_click(FALSE)* the button will not grab focus when it is clicked by the mouse. This could be useful in places like toolbars, so that the keyboard focus is not removed from the main area of the application.

5. 3. LinkButton



A button that links to a web page.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;

# a window
my $window = Gtk3::Window->new('toplevel');
$window->set_title ('GNOME LinkButton');
$window->set_default_size(250,50);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# a button
my $button = Gtk3::LinkButton->new('http:://live.gnome.org');
# with a label
$button->set_label('Link to GNOME live');

# add the button to the window
$window -> add ($button);

# show the window and run the Application
```

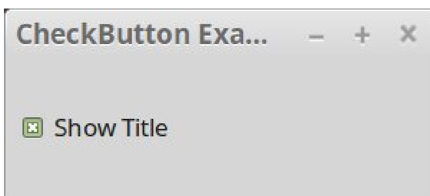
3 Unfortunately the nichnames for Gtk3::IconSize seems not to work at the moment because Perl/Gtk3 expects a integer. For this reason (I believe!) you have to use the following options:
'0' = 'invalid', '1' = 'menu', '2' = 'small-toolbar', '3' = 'large-toolbar, '4' = 'button',
'5' = 'dnd' '6' = 'dialog'

```
$window -> show_all();
Gtk3->main();
```

Useful methods for a LinkButton widget

- *get_visited()* returns the *'visited'* state (*TRUE* or *FALSE*) of the URI where the LinkButton points. The button becomes visited when it is clicked.
- *set_visited(TRUE)* sets the *'visited'* state of the URI where the LinkButton points as *TRUE* (analogously for *FALSE*).
- Each time the button is clicked, the signal *'activate-link'* is emitted. For an explanation of signals and callback functions, see Signals and callbacks.

5. 4. CheckButton



This CheckButton toggles the title.

Code used to generate this example

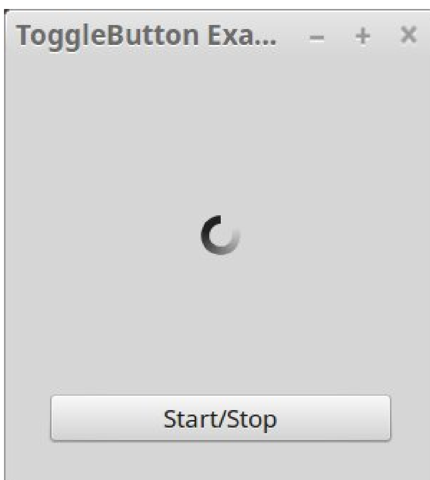
```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;

# a window
my $window = Gtk3::Window->new('toplevel');
$window->set_title ('CheckButton Example');
$window->set_default_size(300,100);
$window->set_border_width(10);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# a button
my $button = Gtk3::CheckButton->new();
# with a label
$button->set_label('Show Title');

# connect the signal 'toggled' emitted by the checkbutton
# with the callback function toggled_cb
$button->signal_connect('toggled' => \&toggled_cb);

# by default, the checkbutton is active
$button->set_active(TRUE);

# add the button to the window
$window -> add($button);
```

```
# show the window and run the Application
$window -> show_all();
Gtk3->main();

sub toggled_cb {
	# if the toggle button is active, set the title of the window
	# as 'Checkbutton Example'
	if ($button->get_active()) {
		$window->set_title ('CheckButton Example');
		}
	else {
		$window->set_title ('');
		}
	}
```

Useful methods for a CheckButton widget

In line 18 the *'toggled'* signal is connected to the callback function *toggled_cb()* using *$widget->signal_connect("signal", \&callback_function)*. See Signals and callbacks for a more detailed explanation.

5. 5. ToggleButton



When this ToggleButton is in an active state, the spinner spins.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Gtk3 -init;
use Glib ('TRUE','FALSE');

# a window
my $window = Gtk3::Window->new('toplevel');
$window->set_title ('ToggleButton Example');
$window->set_default_size(300,300);
$window->set_border_width(30);
```

```
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# a spinner animation
my $spinner = Gtk3::Spinner->new();
# with extra horizontal space
$spinner->set_hexpand(TRUE);
# with extra vertical space
$spinner->set_vexpand(TRUE);

# a toggle button
my $button = Gtk3::ToggleButton->new_with_label('Start/Stop');
# connect the signal 'toggled' emitted by the togglebutton
# when its state is changed to callback function toggled_cb
$button->signal_connect('toggled' => \&toggled_cb);

# a grid to allocate the widgets
my $grid = Gtk3::Grid->new();
$grid->set_row_homogeneous(FALSE);
$grid->set_row_spacing(15);
$grid->attach($spinner, 0, 0, 1, 1);
$grid->attach($button, 0, 1, 1, 1);

# add the grid to the window
$window->add($grid);

# show the window and run the Application
$window -> show_all();
Gtk3->main();

# callback function for the signal 'toggled'
sub toggled_cb {
	# if the togglebutton is active, start the spinner
	if ($button->get_active()) {
		$spinner->start();
		}
	# else, stop it
	else	{
		$spinner->stop();
		}
	}
```

Useful methods for a ToggleButton widget

In line 25 the *'toggled'* signal is connected to the callback function *toggled_cb()* using *$widget->signal_connect("signal", \&callback_function)*. See Signals and callbacks for a more detailed explanation.

5. 6. Switch

This Switch makes the title appears and disappear.

Code used to generate this example

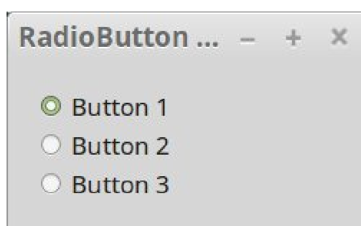
```
#!/usr/bin/perl

use strict;
use Gtk3 -init;
use Glib ('TRUE','FALSE');

# a window
my $window = Gtk3::Window->new('toplevel');
$window->set_title ('Switch Example');
$window->set_default_size(300,100);
$window->set_border_width(10);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# a switch
my $switch = Gtk3::Switch->new();
# turn on by default
$switch->set_active(TRUE);
# connect the signal notify::active emitted by the switch
# to the callback function activate_cb
$switch->signal_connect('notify::active' => \&activate_cb);

# a label
my $label = Gtk3::Label->new();
$label->set_text('Title');

# a grid to allocate the widgets
my $grid = Gtk3::Grid->new();
$grid->set_column_spacing(10);
$grid->attach($label, 0, 0 , 1, 1);
$grid->attach($switch, 1, 0 , 1, 1);

# add the grid to the window
$window->add($grid);

# show the window and run the Application
$window -> show_all();
Gtk3->main();

# Callback function. Since the signal is notify::activv
# we need the argument 'active'
sub activate_cb {
```

```
	if ($switch->get_active) {
		$window->set_title('Switch Example');
		}
	else {
		$window->set_title('');
		}
	}
```

Useful methods for a Switch widget

In line 20 the *'notify::active'* signal is connected to the callback function activate*_cb()* using *$widget->signal_connect("signal", \&callback_function)*. See Signals and callbacks for a more detailed explanation.

5. 7. RadioButton



Three RadioButtons. You can see in the terminal if they are turned off or on.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Gtk3 -init;
use Glib ('TRUE','FALSE');

# a window
my $window = Gtk3::Window->new('toplevel');
$window->set_title ('RadioButton Example');
$window->set_default_size(250,100);
$window->set_border_width(20);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# a new radiobutton with a label
my $button1 = Gtk3::RadioButton->new('Button1');
# with label 'Button 1'
$button1->set_label('Button 1');
# connect the signal 'toggled' emitted by the radiobutton
# with the callback function toggled_cb
$button1->signal_connect('toggled' => \&toggled_cb);

# another radio button, in the same group as button1
my $button2 = Gtk3::RadioButton->new_from_widget($button1);
# with label 'Button 2'
$button2->set_label('Button 2');
# connect the signal 'toggled' emitted by the radiobutton
```

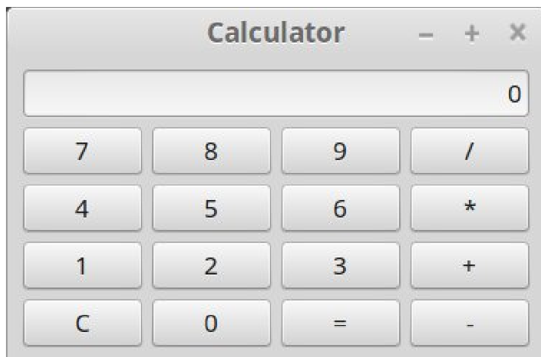
```
# with the callback function toggled_cb
$button2->signal_connect('toggled' => \&toggled_cb);
# set button2 not active by default
$button2->set_active(FALSE);

# another radio button, in the same group as button1
# with label 'Button 3'
my $button3 = Gtk3::RadioButton->new_from_widget($button1);
# with label 'Button 3'
$button3->set_label('Button 3');
# connect the signal 'toggled' emitted by the radiobutton
# with the callback function toggled_cb
$button3->signal_connect('toggled' => \&toggled_cb);
# set button2 not active by default
$button3->set_active(FALSE);

# a grid to place the buttons
my $grid = Gtk3::Grid->new();
$grid->attach($button1, 0, 0, 1, 1);
$grid->attach($button2, 0, 1, 1, 1);
$grid->attach($button3, 0, 2, 1, 1);
# add the grid to the window
$window->add($grid);

# show the window and run the Application
$window -> show_all();
Gtk3->main();

# callback function
sub toggled_cb {
	# The first value of the variable given to the function (=@_) always contains a
	# reference to the widget, which emits the signal.
	my ($button) = @_;
	# a string to describe the state of the button
	my $state = 'unknown';
	# whenever the button is turned on, state is on
	if ($button->get_active()) {
		$state = 'on';
		}
	# else state is off
	else {
		$state = 'off';
		}
	# whenever the function is called (a button is turned on or off)
	# print on the terminal which button was turned on/off
	my $button_label = $button->get_label();
	print ("$button_label was turned $state \n");
	}
```

Useful methods for a RadioButton widget

In line 20 the *'toggled'* signal is connected to the callback function toggle_*cb()* using *$widget->signal_connect("signal", \&callback_function)*. See Signals and callbacks for a more detailed explanation.

Another way to create a new RadioButton with a label is *"my $button1 = Gtk3::RadioButton->new_with_label("", "Button1");"*. (the first argument is the group of the radiobuttons, which we can get with *get_group()*, the second argument is the label.

6. ButtonBox



A calculator – the buttons are enclosed in horizontal ButtonBoxes.

Code used to generate this example[4]

```
#!/usr/bin/perl

use strict;
use Gtk3 -init;
use Glib ('TRUE','FALSE');

# a window
my $window = Gtk3::Window->new('toplevel');
$window->set_title ('Calculator');
$window->set_default_size(350,200);
$window->set_border_width(10);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# an entry
my $entry = Gtk3::Entry->new();
# with an initial text
$entry->set_text('0');
# text aligned on the rigtht
$entry->set_alignment(1);
# the text in the entry cannot be modifier writing in it
$entry->set_can_focus(FALSE);

# a grid
my $grid = Gtk3::Grid->new();
```

4 Die Berechnungen, die ja reines Perl und daher nicht Hauptgegenstand dieses Tutorials sind, sind noch nicht vollständig programmiert

```
$grid->set_row_spacing(5);

# to attach the entry
$grid->attach($entry, 0,0,1,1);

# the labels for the button
my @buttons =         (7, 8, 9, '/',
                4, 5, 6, '*',
                1, 2, 3, '+',
                'C', 0, '=', '-');

# each row is a ButtonBox, attached to the grid
foreach my $i (0..3) {
        my $hbox = Gtk3::ButtonBox->new('horizontal');
        $hbox->set_spacing(5);
        $grid->attach($hbox, 0, $i+1, 1, 1);
        # each ButtonBox has 4 buttons, connected to the callback function
        foreach my $j (0..3) {
                my $button=Gtk3::Button->new();
                $button->set_label("$buttons[$i*4+$j]");
                $button->set_can_focus(FALSE);
                $button->signal_connect('clicked' => \&button_clicked);
                $hbox->add($button);
                }
        }

# some variables for the calculations
my $first_number = 0;
my $second_number = 0;
my $counter = 0;
my $operation = '';

# add the grid to the window
$window->add($grid);

# show the window and run the Application
$window -> show_all();
Gtk3->main();

# callback function for all the buttons
sub button_clicked {
        # Die erste übergebene Variable enthält immer eine Referenz auf den
        # geklickten Button
        my $button = $_[0];

        # Erhalte das Label des geklickten Button
        my $label = $button->get_label();

        # for the operations
        if ($label eq '+') {
                $counter += 1;
                if ($counter > 1) {
```

```
			do_operation();
			}
		$entry->set_text('0');
		$operation = 'plus';
		}
	elsif ($label eq '-') {
		$counter += 1;
		if ($counter > 1) {
			do_operation();
			}
		$entry->set_text('0');
		$operation = 'minus';
		}
	elsif ($label eq '*') {
		$counter += 1;
		if ($counter > 1) {
			do_operation();
			}
		$entry->set_text('0');
		$operation = 'multiplication';
		}
	elsif ($label eq '/') {
		$counter += 1;
		if ($counter > 1) {
			do_operation();
			}
		$entry->set_text('0');
		$operation = 'division';
		}
	 # for '='
	elsif ($label eq '=') {
		do_operation();
		$entry->set_text("$first_number");
		$counter = 1;
		}
	# for Cancel
	elsif ($label eq 'C') {
		$first_number = 0;
		$second_number = 0;
		$counter = 0;
		$entry->set_text('0');
		$operation = '';
		}
	# for a digit button - für Zahl-Buttons
	else {
		my $new_digit = $button->get_label();
		# ZERO DIVISI ERROR -> TO DO!!!
		my $number = $entry->get_text();
		$number = $number * 10 + $new_digit;
		if ($counter eq '0') {
			$first_number = $number;
			}
```

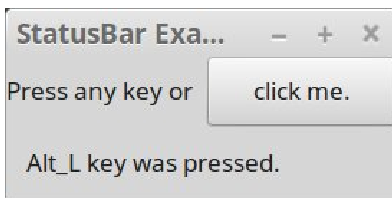
```
        else {
                $second_number = $number;
                }

        $entry->set_text("$number");
        }
    }

sub do_operation {
    if ($operation eq 'plus') {
    $first_number += $second_number;
        }
    elsif ($operation eq 'minus') {
    $first_number -= $second_number;
        }
    elsif ($operation eq 'multiplication') {
    $first_number *= $second_number;
        }
    elsif ($operation eq 'division') {
    $first_number = $first_number / $second_number;
    # ZERO DIVISI ERROR -> TO DO!!!
        }
    }
```

Useful methods for a ButtonBox widget

- The layout of the ButtonBox are set with *set_layout(layout)*, where layout can be *"spread"* (buttons are evenly spread across the box), *"edge"* (buttons are placed at the edges of the box), *"start"* (buttons are grouped towards the start of the box), *"end"* (buttons are grouped towards the end of the box), "*center*" (buttons are centered in the box), "expand".
- *set_child_secondary($button, TRUE)* sets whether button should appear in a secondary group of children. A typical use of a secondary child is the help button in a dialog. This group appears after the other children if the style is *"start"*, "spread" or *"edge"*, and before the other children if the style is *"end"*. If the style is *"start"* or *"end"*, then the secondary children are aligned at the other end of the button box from the main children. For the other styles, they appear immediately next to the main children.
- *set_child_non_homogeneous($button, TRUE)* sets whether the child is exempted from homogeneous sizing. Default is FALSE.
- *set_spacing(spacing)* sets the spacing in pixels between the buttons of the box.

7. Other display widgets

7. 1. Statusbar

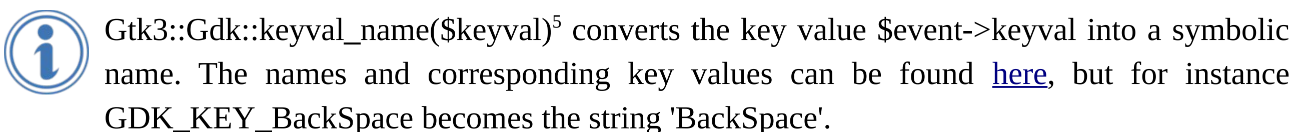


The statusbar tells you if you click the button or if you press any key (and which key!)

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Gtk3 -init;
use Glib ('TRUE','FALSE');

# a window
my $window = Gtk3::Window->new('toplevel');
$window->set_title ('StatusBar Example');
$window->set_default_size(200,100);
# a key press event for the keyboard input
$window->signal_connect('key-press-event' => \&do_key_press_event);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# a label
my $label = Gtk3::Label->new('Press any key or ');

# a button
my $button = Gtk3::Button->new('click me.');
# connected to a callback
$button->signal_connect('clicked', \&button_clicked_cb);

# the statusbar
my $statusbar = Gtk3::Statusbar->new();
# its $context_id - not shown in the UI but needed to uniquely identify
# the source of a message
my $context_id = $statusbar->get_context_id('example');
# we push a message onto the statusbar's stack
$statusbar->push($context_id, 'Waiting for you to do something...');

# a grid to attach the widgets
my $grid = Gtk3::Grid->new();
$grid->set_column_spacing(5);
$grid->set_column_homogeneous(TRUE);
$grid->set_row_homogeneous(TRUE);
$grid->attach($label, 0, 0, 1, 1);
$grid->attach_next_to($button, $label, 'right', 1, 1);
$grid->attach($statusbar, 0, 1, 2, 1);
```

```
# add the grid to the window
$window->add($grid);

# show the window and run the Application
$window -> show_all();
Gtk3->main();

# callback function for the  button clicked
# if the button is clicked the event is signaled to the statusbar
# onto which we push a new status
sub button_clicked_cb {
	$statusbar->push($context_id, 'You clicked the button.');
	}

# event handler
# any signal from the keyboard is signaled to the statusbar
# onto which we push a new status with the symbolic name
# of the key pressed
sub do_key_press_event {
	my ($widget, $event) = @_;
	my $keyval = $event->keyval;
	# !!! important: The perlish diction Gtk3::Gdk->->keyval_name($event);
	# doesn't work  !!!
	my $key = Gtk3::Gdk::keyval_name($keyval);
	$statusbar->push($context_id, "$key key was pressed.");

	# stop the signal emission
	return TRUE;
	}
```



Gtk3::Gdk::keyval_name($keyval)[5] converts the key value $event->keyval into a symbolic name. The names and corresponding key values can be found here, but for instance GDK_KEY_BackSpace becomes the string 'BackSpace'.

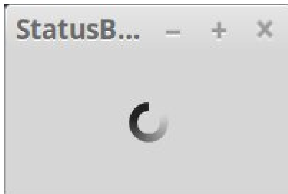
Useful methods for a Statusbar widget

In line 21 the *'clicked'* signal is connected to the callback function button_clicked*_cb()* using *$widget->signal_connect("signal", \&callback_function)*. See Signals and callbacks for a more detailed explanation.

- *pop($context_id)* removes the first message in the statusbar stack with the given context_id
- *remove_all($context_id)* removes all the messages in the statusbar stack with the given *context_id*
- *remove($context_id, $message_id)* removes the message with the given *message_id* in the statusbar stack with the given *context_id*. The *message_id* returned by my message_id = *push($context_id, "the message")* when pushing the message on stack of the statusbar

5 The more perlish Schreibweise *"Gtk3::Gdk->keyval_name"* seems not to work correctely

7. 2 Spinner



This Spinner is stopped and started by pressing the spacebar.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Gtk3 -init;
use Glib ('TRUE','FALSE');

# a window
my $window = Gtk3::Window->new('toplevel');
$window->set_title ('StatusBar Example');
$window->set_default_size(200,100);
$window->signal_connect('key-press-event' => \&do_key_press_event);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# a spinner
my $spinner = Gtk3::Spinner->new();
# that by default spins
$spinner -> start();
# add the spinner to the window
$window->add($spinner);

# show the window and run the Application
$window -> show_all();
Gtk3->main();

# event handler
# a signal from the keyboard (space) controls if the spinner stops/starts
sub do_key_press_event {
	# $keyname is the symbolic name of the key value given by the event
	my ($widget, $event) = @_;
	my $keyval = $event->keyval;
	# !!! IMPORTANT: The perlish diction Gtk3::Gdk->->keyval_name($event);
	# doesn't work !!!
	my $keyname = Gtk3::Gdk::keyval_name($keyval);

	# if  it is 'space'
	if ($keyname == 'space') {
		# an the spinner ist active
		if ($spinner->get_property('active')) {
			# stop the spinner
			$spinner->stop();
			}
```

```
                # if the spinner is not active
                else {
                        # start it again
                        $spinner->start();
                        }
                }
        # stop the signal emission
        return TRUE;
}
```

Gtk3::Gdk::keyval_name($keyval)[6] converts the key value $event->keyval into a symbolic name. The names and corresponding key values can be found here, but for instance GDK_KEY_BackSpace becomes the string 'BackSpace'.

7. 3. ProgressBar

This ProgressBar is stopped and started by pressing any key.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Gtk3 -init;
use Glib ('TRUE','FALSE');

# a window
my $window = Gtk3::Window->new('toplevel');
$window->set_title ('ProgressBar Example');
$window->set_default_size(220,20);
$window->signal_connect('key-press-event' => \&do_key_press_event);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# a progressbar
my $progressbar = Gtk3::ProgressBar->new();
# add the progressbar to the window
$window->add($progressbar);

# the method pulse is called each 100 milliseconds
# and $source_id is set to be the ID of the event source
# (i.e. the bar changes position every 100 milliseconds)
my $source_id = Glib::Timeout->add(100, \&pulse);

# show the window and run the Application
$window -> show_all();
```

6 The more perlish Schreibweise *"Gtk3::Gdk->keyval_name"* seems not to work correctely

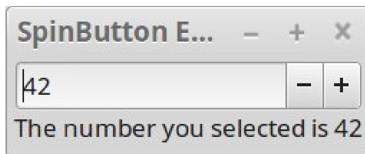
```
Gtk3->main();

# event handler
# any signal from the keyboard controls if the progressbar stops/starts
sub do_key_press_event {
	# if the progressbar has been stopped (therefore $source_id == 0 - see
	# 'else' below) turn it back on
	if ($source_id == 0) {
		$source_id = Glib::Timeout->add(100, \&pulse);
		}
	# if the bar is moving, remove the source with the ID of $source_id
	# from the main context (i.e. stop the bar) and set the source id to 0
	else {
		Glib::Source->remove($source_id);
		$source_id = 0;
		}
	# stop the signal emission
	return TRUE;
	}

# source function
# the progressbar is in 'activity mode' when this method is called
sub pulse {
	$progressbar->pulse();
	# call the function again
	return TRUE;
	}
```

Useful methods for a ProgressBar widget

- Instead of *pulse()*, that makes the bar goes back and forth, if we want the ProgressBar to "fill in" a fraction (a float between 0.0 und 1.0 included) of the bar that has been completed, use *set_fraction(fraction)*
- To set a text and show it (superimposed over the bar) use *set_text("text")* and *set_show_text(TRUE)*. If a text is not set and *set_show_text(TRUE)* the text will be the percentage of the work that has been completed.

8. Entry Widgets

8. 1. SpinButton



Choose a number, by entering it or by clicking on the -/+ buttons!

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Gtk3 -init;
use Glib ('TRUE','FALSE');

# a window
my $window = Gtk3::Window->new('toplevel');
$window->set_title ('SpinButton Example');
$window->set_default_size(210,70);
$window->set_border_width(5);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# an adjustment (initial value, max value,
# step increment - press cursor keys or +/- buttons to see!,
# page increment  not used here,
# page size - not used here)
my $ad = Gtk3::Adjustment->new(0, 0, 100, 1, 0, 0);

# a spin button for integers (digits=0)
# options ($adjustment, climb_rate, digits)
my $spin = Gtk3::SpinButton->new($ad, 1, 0);
# as wide as possible
$spin->set_hexpand(TRUE);

# we connect the signal 'value-changed' emitted by the spinbutton with the callback
# function spin_selected
$spin->signal_connect('value-changed' => \&spin_selected);

# a label
my $label = Gtk3::Label->new();
$label -> set_text('Choose a number');

# a grid to attach the widgets
my $grid = Gtk3::Grid->new();
$grid->attach($spin, 0, 0, 1, 1);
$grid->attach($label, 0, 1, 2, 1);

# add the grid to the window
```

```
$window->add($grid);

# show the window and run the Application
$window -> show_all();
Gtk3->main();

# callback function: the signal of the spinbutton is used to change the
# text of the label
sub spin_selected {
	my $number = $spin->get_value_as_int();
	$label->set_text("The number you selected is $number");
	}
```

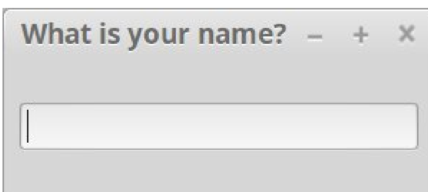
Useful methods for a SpinButton widget

A Gtk3::Adjustment is needed to construct the Gtk3::SpinButton. This is the representation of a value with a lower and upper bound, together with step and page increments, and a page size, and it is constructed as *Gtk3::Adjustment->new(value, lower, upper, step_increment, page_increment, page_size)* where the fields are of type float; *step_increment* is the increment/decrement that is obtained by using the cursor keys or the buttons of the spinbutton. Note that *page_increment* and *page_size* are not used in this case, and they should be set to 0.

In line 27 the signal *'value-changed'* is connected to the callback function *spin_selected()* using $widget->signal_connect("signal" => \&callback function). See Signale und Callbacks for a more detailed explanation.

- If you want the value of the spinbutton to wrap around when they exceed the maximum or the minimum, set *set_wrap(TRUE)*. The 'wrapped' signal is emitted when this happens.
- *set_digits(digits)* sets the precision of the digits to be displayed by the spinbutton (up to 20 digits).
- To get the value of the spinbutton as an integer, use *get_value_as_int()*.

8. 2. Entry



This application greets you in the terminal with the name you provide.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Gtk3 -init;
```

```
use Glib ('TRUE','FALSE');

# a window
my $window = Gtk3::Window->new('toplevel');
$window->set_title ('What is your name?');
$window->set_default_size(300,100);
$window->set_border_width(10);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# a single line entry
my $name_box = Gtk3::Entry->new();
# emits a signal when Enter key is pressed, connected to the
# callback function cb_activate
$name_box->signal_connect('activate', \&cb_activate);

# add the Entry to the window
$window->add($name_box);

# show the window and run the Application
$window -> show_all();
Gtk3->main();

# the content of the entry is used to write in the terminal
sub cb_activate {
	# retrieve the content of the widget
	my $entry = $_[0];
	my $name = $entry->get_text();
	# print it in a nice form in the terminal
	#(ohne "\n" kommt keine Ausgabe auf dem Terminal, bis das Programm beendet wird!
	print "Hello $name! \n";
	}
```

Useful methods for Entry widget

In line 12 the signal *'activate'* is connected to the callback function *cb_activate()* using *$widget->signal_connect("signal", \&callback function)*. See Signale und Callbacks for a more detailed explanation. Some of the signals that a *Gtk3::Entry* widget can emit are: *'activate'* (emitted when the user activates the Entry key); *'backspace'* (emitted when the user activates the Backspace or Shift-Backspace keys); *'copy-clipboard'* (Ctrl-c and Ctrl-Insert); *'paste-clipboard'* (Ctrl-v and Shift-Insert); *'delete-from-cursor'* (Delete, for deleting a character; Ctrl-Delete, for deleting a word); *'icon-press'* (emitted when the user clicks an activatable icon); *'icon-release'* (emitted on the button release from a mouse click over an activatable icon); *'insert-at-cursor'* (emitted when the user initiates the insertion of a fixed string at the cursor); *'move-cursor'* (emitted when the user initiates a cursor movement); *'populate-popup'* (emitted before showing the context menu of the entry; it can be used to add items to it).

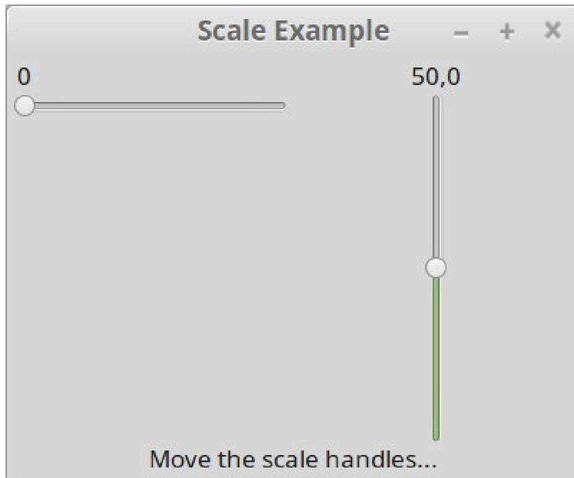
- *get_buffer()* and *set_buffer($buffer)*, where $buffer is a Gtk3::EntryBuffer object, can be used to get and set the buffer for the entry.
- *get_text()* and *set_text('some text')* can be used to get and set the content for the entry.

- *get_text_length()* is self-explanatory.
- *get_text_area()* gets the area where the entry's text is drawn.
- If we set *set_visibility(FALSE)* the characters in the entry are displayed as the invisible char. This is the best available invisible character in the current font, but it can be changed with *set_invisible_char(ch)*, where ch is a Unicode charcater. The latter method is reversed by *unset_invisbile_char()*.
- *set_max_length(int)*, where int is an integer, truncates every entry longer than int to have the desired maximum length.
- By default, if you press the Entry key the *Gtk3::Entry* emits the signal *'activate'*. If you would like to activate the default widget for the window (set using *set_default($widget)* on the window), then use *set_activates_default(TRUE)*.
- To set a frame around the entry: *set_has_frame(TRUE)*.
- *set_placeholder_text('some text')* sets the text to be displayed in the entry when it is empty and unfocused.
- *set_overwrite_mode(TRUE)* and *set_overwrite_mode(FALSE)* are self-explanatory.
- If we have *set_editable(FALSE)* the user cannot edit the text in the widget.
- *set_completion($completion)*, where *$completion* is a *Gtk3::EntryCompletion*, sets completion - or disables it if *$completion* is 'None'.
- An Entry widget can display progress or activity information behind the text. We use *set_progress_fraction(fraction)*, where fraction is a float between 0.0 and 1.0 inclusive, to fill in the given fraction of the bar. We use *set_progress_pulse_step()* to set the fraction of total entry width to move the progress bouncing block for each call to *progress_pulse()*. The latter method indicates that some progress is made, and causes the progress indicator of the entry to enter 'activity mode', where a block bounces back and forth. Each call to *progress_pulse()* causes the block to move by a little bit (the amount of movement per pulse is determined, as said before, by *set_progress_pulse_step()*).
- An Entry widget can also show icons. These icons can be activatable by clicking, can be set up as drag source and can have tooltips.

  To add an icon, you can use *set_icon_from_stock(icon_position, stock_id)*, or one of *set_icon_from_pixbuf(icon_position, pixbuf)*, *set_icon_from_icon_name(icon_position, icon_name)*, where *icon_position* is one of *"primary"*[7] (to set the icon at the beginning of the entry) *"secondary"*[8] (to set the icon at the end of the entry). To set a tooltip on an icon, use *set_icon_tooltip_text('tooltip text')* or *set_icon_tooltip_markup('tooltip text in Pango markup language')*.

7 Or Gtk3::EntryIconPosition::PRIMARY
8 Or Gtk3::EntryIconPosition::SECONDARY

8. 3. Scale



Slide the scales!

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Gtk3 -init;
use Glib ('TRUE','FALSE');

# a window
my $window = Gtk3::Window->new('toplevel');
$window->set_title ('Scale Example');
$window->set_default_size(400,300);
$window->set_border_width(5);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# two adjustments (initial value, min value, max value,
# step incremnt - press cursor keys to see!,
# page increment - click around the handle to see!,
# page size - not used here)
my $ad1 = Gtk3::Adjustment->new(0,0,100,5,10,0);
my $ad2 = Gtk3::Adjustment->new(50,0,100,5,10,0);

# a horizontal scale
my $h_scale = Gtk3::Scale->new('horizontal',$ad1);
# of integers (no digits)
$h_scale->set_digits(0);
# that can expand horizontally if there is space in the grid (see
# below)
$h_scale->set_hexpand(TRUE);
# that is aligned at the top of the space allowed in the grid (see
# below)
$h_scale->set_valign('start');

# we connect the signal 'value-changed' emittet by the scale with the
# callback function scale_moved
```

```
$h_scale->signal_connect('value-changed' => \&scale_moved);

# a vertical scale
my $v_scale = Gtk3::Scale->new('vertical',$ad2);
# that can expand vertically if there is space in the grid (see
# below)
$v_scale->set_vexpand(TRUE);

# we connect the signal 'value-changed' emittet by the scale with the
# callback function scale_moved
$v_scale->signal_connect('value-changed' => \&scale_moved);

# a label
my $label = Gtk3::Label->new();
$label->set_text('Move the scale handles...');

# a grid to attach the widgets
my $grid = Gtk3::Grid->new();
$grid ->set_column_spacing(10);
$grid->set_column_homogeneous(TRUE);
$grid->attach($h_scale, 0, 0, 1, 1);
$grid->attach_next_to($v_scale, $h_scale, 'right', 1, 1);
$grid->attach($label, 0, 1, 2, 1);

$window->add($grid);

# show the window and run the Application
$window -> show_all();
Gtk3->main();

# any signal from the scales is signaled to the label the text of which is
# changed
sub scale_moved {
	my ($widget, $event) = @_;
	my $h_value = $h_scale->get_value();
	my $v_value = $v_scale->get_value();
	$label->set_text("Horizontal scale is $h_value; vertical scale is $v_value.");
	}
```

Useful methods for a Scale widget

A *Gtk3::Adjustment* is needed to construct the *Gtk3::Scale*. This is the representation of a value with a lower and upper bound, together with step and page increments, and a page size, and it is constructed as *Gtk3::Adjustment->new(value, lower, upper, step_increment, page_increment, page_size)* where the fields are of type float; *step_increment* is the increment/decrement that is obtained by using the cursor keys, *page_increment* the one that is obtained clicking on the scale itself. Note that *page_size* is not used in this case, it should be set to 0.

In line 34 the signal *'value-changed'* is connected to the callback function *scale_moved()* using *$widget->signal_connect("signal" => \&callback function)*. See Signale und Callbacks for a more

detailed explanation.

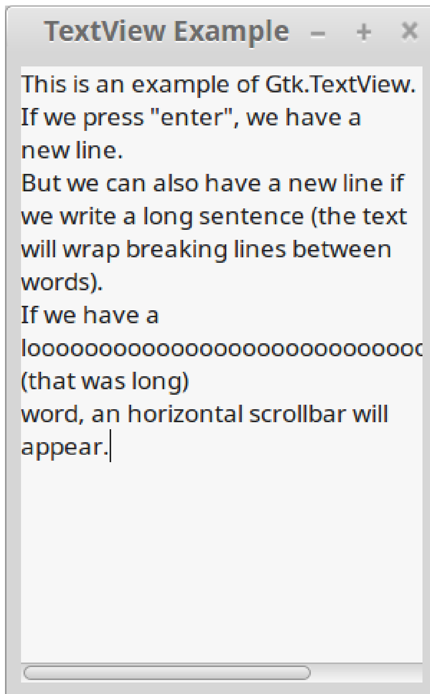
- *get_value()* retrieves the current value of the scale; *set_value(value)* sets it (if the value, of type float, is outside the minimum or maximum range, it will be clamped to fit inside them). These are methods of the class Gtk3::Range.
- Use *set_draw_value(FALSE)* to avoid displaying the current value as a string next to the slider.
- To highlight the part of the scale between the origin and the current value (not tested):

```
$h_scale->set_restrict_to_fill_level(FALSE);
$h_scale->set_fill_level($h_scale->get_value());
$h_scale->set_show_fill_level(TRUE);
```

  in the callback function of the *'value-changed'* signal, so to have the new filling every time the value is changed. These are methods of the class Gtk3::Range.
- *add_mark(value, position, markup)* adds a mark at the *value* (float or int if that is the precision of the scale), in *position* ("left", "right", "top", "bottom") with text *Null* or *markup* in the Pango Markup Language. To clear marks, *clear_marks()*.
- *set_digits(digits)* sets the precision of the scale at digits digits.

9. A widget to write and display text

9. 1. TextView



Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Gtk3 -init;
use Glib ('TRUE','FALSE');

# a window
my $window = Gtk3::Window->new('toplevel');
$window->set_title ('TextView Example');
$window->set_default_size(300, 450);
$window->set_border_width(5);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# a scrollbar for the child widget (that is going to be the textview!)
my $scrolled_window = Gtk3::ScrolledWindow->new();
$scrolled_window->set_border_width(5);
# we scroll only if needed
$scrolled_window->set_policy('automatic','automatic');

# a text buffer (stores text)
my $buffer1 = Gtk3::TextBuffer->new();

# a textview
my $textview = Gtk3::TextView->new();
# displays the buffer
```

```
$textview->set_buffer($buffer1);
# wrap the text, if needed, breaking lines in between words
$textview->set_wrap_mode('word');

# textview is scrolled
$scrolled_window->add($textview);

$window->add($scrolled_window);

# show the window and run the Application
$window -> show_all();
Gtk3->main();
```

Useful methods for a TextView widget

A *Gtk3::TextView* displays the text stored in a *Gtk3::TextBuffer*. However, most text manipulation is accomplished with iterators, represented by a *Gtk3::TextIter*[9] - a position between two characters in the text buffer. Iterators are not valid indefinitely; whenever the buffer is modified in a way that affects the contents of the buffer, all outstanding iterators become invalid. Because of this, iterators can’t be used to preserve positions across buffer modifications. To preserve a position, we use a *Gtk3::TextMark*, that can be set visible with *visible(TRUE)*. A text buffer contains two built-in marks; an '*insert*' mark (the position of the cursor) and the *'selection_bound'* mark.

Methods for a TextView widget:

- The TextView widget is by default editable. If you prefer otherwise, use *set_editable(FALSE)*. If the buffer has no editable text, it might be a good idea to use *set_cursor_visible(FALSE)* as well.
- The justification of the text is set with *set_justification(Gtk.Justification)* where *Gtk3.Justification* is one of *"left"*, *"right"*, *"center"*, *"fill"*.
- The line wrapping of the text is set with s*et_wrap_mode(Gtk.WrapMode)* where *Gtk.WrapMode* is one of *"none"* (the text area is made wider), *"char"* (break lines anywhere the cursor can appear), *"word"* (break lines between words), *"word_char"* (break lines between words, but if that is not enough between characters).

Methods for a TextBuffer widget:

- *get_insert()* returns the *Gtk3::TextMark* that represents the cursor, that is the insertion point.
- *get_selection_bound()* returns the *Gtk3::TextMark* that represents the selection bound.
- *set_text('some text', length)* where *length* is a positive integer or -1, sets the content of the buffer as the first length characters of the 'some text' text. If *length* is omitted or -1, the text is inserted completely. The content of the buffer, if there is any, is destroyed.
- *insert($iter, 'some text', length)* where *$iter* is a text iterator and *length* is a positive integer

9 A Gtk3::TextIter is initialized with the $textbuffer->get_iter_* methods (for more informations see the API Reference)

or -1, inserts in the buffer at iter the first length characters of the 'some text' text. If *length* is omitted or -1, the text is inserted completely.

- *insert_at_cursor('some text', length)* does the same as *insert($iter, 'some text', length)*, with the current cursor taken as iter.
- *create_mark('mark_name', $iter, left_gravity)* where iter is a *Gtk3::TextIter* and *left_gravity* is a boolean, creates a *Gtk3::TextMark* at the position of iter. If *'mark_name'* is None, the mark is anonymous; otherwise, the mark can be retrieved by name using *get_mark()*. If a mark has *left gravity*, and text is inserted at the mark's current location, the mark will be moved to the left of the newly-inserted text. If *left_gravity* is omitted, it defaults to False.
- To specify that some text in the buffer should have specific formatting, you must define a *tag* to hold that formatting information, and then apply that tag to the region of text using *create_tag('tag name', property)* and *apply_tag(tag, start_iter, end_iter)* as in, for instance:

```
tag = $textbuffer->create_tag('orange_bg', background='orange');
$textbuffer->apply_tag(tag, $start_iter, $end_iter);
```

The following are some of the common styles applied to text:

  - Background colour ('background' property)
  - Foreground colour ('foreground' property)
  - Underline ('underline' property)
  - Bold ('weight' property)
  - Italics ('style' property)
  - Strikethrough ('strikethrough' property)
  - Justification ('justification' property)
  - Size ('size' and 'size-points' properties)
  - Text wrapping ('wrap-mode' property)

You can also delete particular tags later using *remove_tag()* or delete all tags in a given region by calling *remove_all_tags()*.

Methods for a TextIter widget

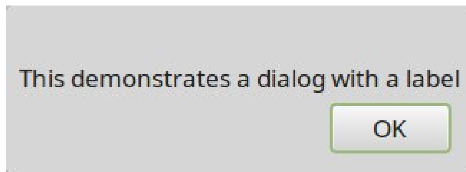
- forward_search(needle, flags, limit) searches forward for needle. The search will not continue past the Gtk.TextIter limit. The flags can be set to one of the following, or any combination of it by concatenating them with the bitwise-OR operator |: 0 (the match must be exact); Gtk.TextSearchFlags.VISIBLE_ONLY (the match may have invisible text interspersed in needle); Gtk.TextSearchFlags.TEXT_ONLY (the match may have pixbufs or child widgets mixed inside the matched range); Gtk.TextSearchFlags.CASE_INSENSITIVE (the text will be matched regardless of what case it is in). The method returns a tuple containing a Gtk.TextIter pointing to the start and to the first character after the match; if no

match is found, None is returned.

- backward_search(needle, flags, limit) does the same as forward_search(), but moving backwards.

10. Dialoge

10. 1. Dialog



A dialog with the response signal connected to a callback function.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Gtk3 -init;
use Glib ('TRUE','FALSE');

# construct a window (the parent window)
my $window = Gtk3::Window->new('toplevel');
$window->set_title ('GNOME Button');
$window->set_default_size(250,50);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit()});

# a button on the parent window
my $button = Gtk3::Button->new('Click me');
# connect the signal 'clicked' of the button with the function
# on_button_click()
$button->signal_connect('clicked', \&on_button_click);
# add the button to the window
$window->add($button);

# show the window and run the Application
$window -> show_all();
Gtk3->main();

# callback function for the signal 'clicked' of the button in the parent
# window
sub on_button_click {
	# create a Dialog
	my $dialog = Gtk3::Dialog->new();
	$dialog->set_title('A Gtk+ Dialog');

	# the window defined in the constructor ($window) is the parent of
	# the dialog.
	# Furthermore, the dialog is on top of the parent window
	$dialog->set_transient_for($window);

	# set modal true: no interaction with other windows of the application
	$dialog->set_modal(TRUE);
```

```
        # add a button to the dialog window
        $dialog->add_button('OK','ok');

        # connect the 'response' signal (the button has been clicked) to the
        # function on_response()
        $dialog->signal_connect('response' => \&on_response);

        # get the content area of the dialog, add a label to it
        my $content_area = $dialog->get_content_area();
        my $label = Gtk3::Label->new('This demonstrates a dialog with a label');
        $content_area->add($label);

        # show the dialog
        $dialog->show_all();
        }

sub on_response {
        my ($widget, $response_id) = @_;
        print "response_id is $response_id \n";
        # destroy the widget (the dialog) when the function on_response() is called
        # (that is, when the button of the dialog has been clicked)
        $widget->destroy();
        }
```

Useful methods for a Dialog widget

In line 17 the signal *'clicked'* is connected to the callback function *on_button_click()* using *$widget->signal_connect("signal" => \&callback function)*. See Signale und Callbacks for a more detailed explanation.

- Instead of *set_modal(TRUE)* we could have *set_modal(FALSE)* followed by *set_destroy_with_parent(TRUE)* that would destroy the dialog window if the main window is closed.
- *add_button(button_text, response_id)*, where *response_id* is any integer, is an alternative to add_button(button_text, Gtk.ResponseType), where *Gtk.ResponseType* could be one of "ok", "cancel", "close", "yes", *"no"*, *"apply"*, *"help"*, which in turn correspond to the integers -5, -6,..., -11.

10. 2. AboutDialog

An AboutDialog example using Gtk3::Window and Menu (the 'about' is displayed if 'About' in the menu is selected).

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Gtk3 -init;
use Glib ('TRUE','FALSE');

my $window = Gtk3::Window->new('toplevel');
$window->set_title('AboutDialog Example');
$window->set_default_size(200,200);
$window->signal_connect('destroy'=>sub {Gtk3->main_quit;});

# a menubar created in the method create_menubar (see below)
my $menubar = create_menubar();

# add the menubar to a grid
my $grid=Gtk3::Grid->new();
$grid->attach($menubar,0,0,1,1);

# add the grid to window
$window->add($grid);

$window->show_all();
Gtk3->main();

sub create_menubar {
	# create a menubar
	my $menubar=Gtk3::MenuBar->new();

	# create a menubar item
	my $menubar_item=Gtk3::MenuItem->new('Anwendung');

	# add the menubar item to the menubar
	$menubar->insert($menubar_item,0);
```

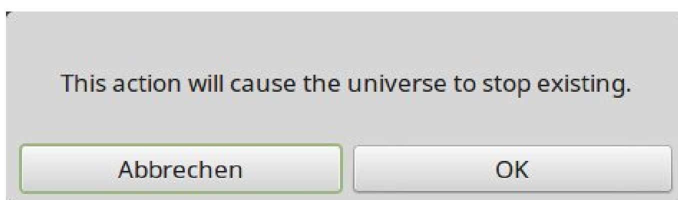
```
	# create a menu
	my $menu=Gtk3::Menu->new();

	# add 2 items to the menu
	my $item1=Gtk3::MenuItem->new('About');
	$item1->signal_connect('activate'=>\&about_cb);

	my $item2=Gtk3::MenuItem->new('Quit');
	$item2->signal_connect('activate'=> sub {Gtk3->main_quit();});


	$menu->insert($item1,0);
	$menu->insert($item2,1);

	# add the menu to the menubar(_item!)
	$menubar_item->set_submenu($menu);

	# return the complete menubar
	return $menubar;
	}

sub about_cb {
	# a Gtk3::AboutDialog
	my $aboutdialog = Gtk3::AboutDialog->new();

	# lists of authors and documenters (will be used later)
	my @authors = ('GNOME Documentation Team');
	my @documenters = ('GNOME Documentation Team');

	# we fill in the aboutdialog
	$aboutdialog->set_program_name('AboutDialog Example');
	$aboutdialog->set_copyright(
		"Copyright \xa9 2012 GNOME Documentation Team");
	# important: set_authors and set_documenters need an array ref!
	# with a normal array it doesn't work!
	$aboutdialog->set_authors(\@authors);
	$aboutdialog->set_documenters(\@documenters);
	$aboutdialog->set_website('http://developer.gnome.org');
	$aboutdialog->set_website_label('GNOME Developer Website');

	# we do not want to show the title, which by default would be 'About AboutDialog Example'
	# we have to reset the title of the messagedialog window after setting
	# the prorgam name
	$aboutdialog->set_title('');

	# to close the aboutdialog when 'close' is clicked we connect
	# the 'response' signal to on_close
	$aboutdialog->signal_connect('response'=>\&on_close);
	# show the aboutdialog
	$aboutdialog->show();
	}
```

```
# destroy the aboutdialog
sub on_close {
	my ($aboutdialog) = @_;
	$aboutdialog->destroy();
	}
```

Useful methods for an AboutDialog widget

In line 41 the signal *'activate'* is connected to the callback function *about_cb()* using *$widget->signal_connect("signal" => \&callback function)*. See Signale und Callbacks for a more detailed explanation.

10. 3. Message Dialog



A message dialog which prints messages on the terminal, depending on your choices.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Gtk3 -init;
use Glib ('TRUE','FALSE');

my $window = Gtk3::Window->new('toplevel');
$window->set_title('MessageDialog Example');
$window->set_default_size(400,200);
$window->signal_connect('destroy'=>sub {Gtk3->main_quit;});

my $label = Gtk3::Label->new();
$label->set_text('This appliccation goes boom');

# a menubar created in the method create_menubar (see below)
my $menubar = create_menubar();

# pack the menubar and the label in a vertical box
my $vbox = Gtk3::Box->new('vertical', 0);
# Pack a menubar always in a vertical box with option Expand and Fill false!
# alternatively you can use a grid (see the other examples)
$vbox->pack_start($menubar,FALSE,FALSE,0);
$vbox->pack_start($label,TRUE,TRUE,0);

$window->add($vbox);
```

```
$window->show_all();
Gtk3->main();

sub create_menubar {
	# create a menubar
	my $menubar=Gtk3::MenuBar->new();

	# create a menubar item
	my $menubar_item=Gtk3::MenuItem->new('Anwendung');

	# add the menubar item to the menubar
	$menubar->insert($menubar_item,0);

	# create a menu
	my $menu=Gtk3::Menu->new();

	# add 2 items to the menu
	my $item1=Gtk3::MenuItem->new('Message');
	$item1->signal_connect('activate'=>\&message_cb);

	my $item2=Gtk3::MenuItem->new('Quit');
	$item2->signal_connect('activate'=> sub {Gtk3->main_quit();});

	$menu->insert($item1,0);
	$menu->insert($item2,1);

	# add the menu to the menubar(_item!)
	$menubar_item->set_submenu($menu);

	# return the complete menubar
	return $menubar;
	}

# callback function for the signal 'activate' from the message_action
# in the menu of the parent window
sub message_cb {
	# a Gtk3::MessageDialog
	# the options are (parent,flags,MessageType,ButtonType,message)
	my $messagedialog = Gtk3::MessageDialog->new($window,
							'modal',
							'warning',
							'ok_cancel',
							'This action will cause the universe to stop
existing.');
	# connect the response (of the button clicked to the function
	# dialog_response
	$messagedialog->signal_connect('response'=>\&dialog_response);
	# show the messagedialog
	$messagedialog->show();
	}
```

```
sub dialog_response {
	my ($widget, $response_id) = @_;

	# if the button clicked gives response OK (-5)
	if ($response_id eq 'ok') {
		print "boom \n";
		}

	# if the button clicked gives response CANCEL (-6)
	elsif ($response_id eq 'cancel') {
		print "good choice \n";
		}

	# if messagedialog is destroyed (by pressing ESC)
	elsif ($response_id eq 'delete-event') {
		print "dialog closed or cancelled \n";
		}

	# finally, destroy the messagedialog
	$widget->destroy();
	}
```

Useful methods for a MessageDialog widget

In line 45 the signal *'activate'* is connected to the callback function *message_cb()* using *$widget->signal_connect("signal" => \&callback function)*. See Signale und Callbacks for a more detailed explanation.

- In the constructor of MessageDialog we could set flags as *"destroy_with_parent"* (to destroy the messagedialog window when its parent window is destroyed) or as *"modal"* (no interaction with other windows of the application).
- In the constructor of MessageDialog we could set type as any of *"info"*, *"warning"*, *"question"*, *"error"*, *"other"* depending on what type of message we want.
- In the constructor of MessageDialog we could set buttons as any of *"none"*, *"ok"*, *"close"*, *"cancel"*, *"yes_no"*, *"ok_cancel"*, or any button using *add_button()* as in *Gtk3::Dialog*.
- We could substitute the default image of the MessageDialog with another image using

  ```
  $image = Gtk3::Image->new();
  $image->set_from_stock("caps_lock_warning", "dialog");
  $image->show();
  $messagedialog->set_image($image);
  ```

  where "caps_lock_warning" is any image from Stock Items. We could also set any image as in the Image widget, as *$image->set_from_file('filename.png')*.
- *format_secondary_text('some secondary message')* sets a secondary message. The primary text becomes bold.

11. Menus, Toolbars and Tooltips

NOTE: To effectively program Application with Menus and Toolbars, you should create your Program as a Gtk3::Application not in the "main-loop"-style. Therefore you need to import **Glib::IO**, which gives you the possibilities to create GMenus, GActions, GApplications, Gtk3::Applications, Gtk3::ApplicationWindows etc.pp.. Basic instructions on setting up the Glib::IO bindings and creating a Gtk3::Application or a Gtk3::ApplicationWindows you can find in the chapter "1. 2 ApplicationWindows".

Although the implementation of Glib::IO is in a very early development status, creating Menus and Actions seem to work fine and without problems at the moment. If you nevertheless want to program your Menus etc. without Glib::IO, you can find examples here

But advanced techniques as creating your Menus from a xml-file or with Glade then isn't easily possible or at least doesn't work without using deprecated classes (such as Gtk3::UIManager).

11. 1. GMenu

A GtkApplication with a simple GMenu and SimpleActions.

Code used to generate this example

```
#! /usr/bin/perl

# Make a binding to the Gio API in the Perl program (just copy&paste ;-))
# This is necessary mainly for Gtk3::Application and some more stuff
# Alternatively you find an early implementation as a Perl module
# on https://git.gnome.org/browse/perl-Glib-IO (not yet published on CPAN!)
# Hopefully this module simplifies the use of the Gio API in the future
# (see also the notes above).
BEGIN {
  use Glib::Object::Introspection;
  Glib::Object::Introspection->setup(
    basename => 'Gio',
    version => '2.0',
    package => 'Glib::IO');
}
```

```
# The CLASS MyWindow, where we build the Gtk3::ApplicationWindow and its content
package MyWindow;
use strict;
use warnings;
use Gtk3;
use Glib qw/TRUE FALSE/;
# Our class must be a subclass of Gtk3::ApplicationWindow to inherit
# the methods, properties etc.pp. of Gtk3::ApplicationWindow
use base 'Gtk3::ApplicationWindow';

 sub new {
	my ($window, $app) = @_;
	$window = bless Gtk3::ApplicationWindow->new($app);
	$window->set_title ('GMenu Example');
	$window->signal_connect( 'delete_event' => sub {$app->quit()} );

	return $window;
}


# The MAIN FUNCTION should be as small as possible and do almost nothing except creating
# your Gtk3::Application and running it
# The "real work" should always be done in response to the signals fired by Gtk3::Application.
# see below
package main;
use strict;
use warnings;

use Gtk3;
use Glib ('TRUE', 'FALSE');

my $app = Gtk3::Application->new('app.id', 'flags-none');

$app->signal_connect('startup'  => \&_init     );
$app->signal_connect('activate' => \&_build_ui );
$app->signal_connect('shutdown' => \&_cleanup  );

$app->run(\@ARGV);

exit;


# The CALLBACK FUNCTIONS to the SIGNALS fired by the main function.
# Here we do the "real work" (see above)
sub _init {
	my ($app) = @_;

	# create a menu
	my $menu = Glib::IO::Menu->new();

	# append to the menu three options
```

```
	$menu->append('New', 'app.new');
	$menu->append('About','app.about');
	$menu->append('Quit','app.quit');
	# set the menu as menu of the application
	$app->set_app_menu($menu);

	# create an actionfor the option "new" of the menu
	my $new_action = Glib::IO::SimpleAction->new('new',undef);
	# connect it to the callback  function new_cb
	$new_action->signal_connect('activate'=>\&new_cb);
	# add the action to the application
	$app->add_action($new_action);

	# option "about"
	my $about_action = Glib::IO::SimpleAction->new('about',undef);
	$about_action->signal_connect('activate'=>\&about_cb);
	$app->add_action($about_action);

	# option "quit"
	my $quit_action = Glib::IO::SimpleAction->new('quit',undef);
	$quit_action->signal_connect('activate'=>\&quit_cb);
	$app->add_action($quit_action);
}

sub _build_ui {
	my ($app) = @_;
	my $window = MyWindow->new($app);
	$window->show();
}

sub _cleanup {
	my ($app) = @_;
}

# callback function for "new"
sub new_cb {
	print "This does nothing. It is only a demonstration. \n";
}

# callback function for "about"
sub about_cb {
	print "No AboutDialog for you. This is only a demonstration \n";
}

# callback function for "quit"
sub quit_cb {
	print "You have quit \n";
	$app->quit();
}
```

Useful methods for a Menu widget (To DO!)

In line 76 the signal *'activate'* from the action *new_action* (not the menu!) is connected to the callback function *new_cb()* using *$new_action->signal_connect('signal'=>\&callback function)*. See Signals and callbacks for a more detailed explanation.

Useful methods for a GSimpleAction:

- To create a new action that is *stateless*, that is, an action that do not retain or depend on a state given by the action itself, use

```
my $action = Glib::IO::SimpleAction->new('name', parameter_type)
```

  where *'name'* is the name of the action and *parameter_type* is the type of the parameters that the action receives when being activated. This can be *undef*, or *Glib::VariantType->new('s')* if the parameter is of type *str*, or instead of *'s'* a character as described here. To create a new *stateful* (i.e. not stateless) action, use

```
my $action = Glib::IO::SimpleAction->new_stateful('name', parameter_type, initial_state)
```

  where *initial_state* is defined as a GVariant - for instance Glib::Variant->new_string('start'); for a list of possibilities see here.

- *set_enabled(TRUE)* sets the action as enabled; an action must be enabled in order to be activated or in order to have its state changed from outside callers. This should only be called by the implementor of the action. Users of the action should not attempt to modify its enabled flag.
- *set_state(state)*, where *state* is a GVariant, sets the state of the action, updating the *'state'* property to the given value. This should only be called by the implementor of the action; users of the action should instead call *change_state(state)* (where *state* is as above) to request the change.

Useful methods for a GMenu:

- To insert an item in the menu in position *position*, use *insert(position, 'label', 'detailed_action')*, where *label* is the label that will appear in the menu and *detailed_action* is a string composed of the name of the action to which we prepend the prefix '*app.*'. A more detailed discussion of this can be found in the chapter about the menubar.

  To append or prepend an item in the menu use respectively *append('label', 'detailed_action')* and *prepend('label', 'detailed_action')*.

- Another way of adding items to the menu is to create them as *GMenuItems* and use *insert_item(position, $item)*, *append_item($item)*, or *prepend_item($item)*; so for instance we might have:

```
my $about = Glib::IO::MenuItem->new('About', 'app.about');
$menu->append_item($about);
```

- We can also add a whole subsection in a menu using *insert_section(position, 'label', $section)*, *append_section('label', $section)*, or *prepend_section('label', $section)*, where

label is the title of the subsection.

- To add a submenu that will expand and collapse, *use insert_submenu(position, 'label', $section)*, *append_submenu('label', $section)*, or *prepend_submenu('label', $section)*, where *label* is the title of the subsection.
- To remove an item from the menu, use *remove(position)*.
- To set a label for the menu, use *set_label('label')*.

11. 2. MenuButton

The GtkMenuButton widget is used to display a menu when clicked on. This menu can be provided either as a GtkMenu, or an abstract GmenuModel. The GtkMenuButton widget can hold any valid child widget. That is, it can hold almost any other standard GtkWidget. The most commonly used child is the provided GtkArrow.

Code used to generate this example

```
#! /usr/bin/perl

# Make a binding to the Gio API in the Perl program (just copy&paste ;-))
# This is necessary mainly for Gtk3::Application and some more stuff
# Alternatively you find an early implementation as a Perl module
# on https://git.gnome.org/browse/perl-Glib-IO (not yet published on CPAN!)
# Hopefully this module simplifies the use of the Gio API in the future
# (see also the notes above).
BEGIN {
  use Glib::Object::Introspection;
  Glib::Object::Introspection->setup(
    basename => 'Gio',
    version => '2.0',
    package => 'Glib::IO');
```

```
}


# The CLASS MyWindow, where we build the Gtk3::ApplicationWindow and its content
package MyWindow;
use strict;
use warnings;
use Gtk3;
use Glib ('TRUE', 'FALSE');
# Our class must be a subclass of Gtk3::ApplicationWindow to inherit
# the methods, properties etc.pp. of Gtk3::ApplicationWindow
use base 'Gtk3::ApplicationWindow';

sub new {
	my ($window, $app) = @_;
	$window = bless Gtk3::ApplicationWindow->new($app);
	$window->set_title ('MenuButton Example');
	$window->set_default_size(600,400);
	$window->signal_connect( 'delete_event' => sub {$app->quit()} );

	my $grid = Gtk3::Grid->new();

	# a menubutton
	my $menubutton = Gtk3::MenuButton->new();
	$menubutton->set_size_request(80,35);

	$grid->attach($menubutton, 0, 0, 1, 1);

	# a menu with two actions
	my $menumodel = Glib::IO::Menu->new();
	$menumodel->append('New', 'app.new');
	$menumodel->append('About','win.about');

	# a submenu with one action for the menu
	my $submenu = Glib::IO::Menu->new();
	$submenu->append('Quit','app.quit');
	$menumodel->append_submenu('Other',$submenu);

	# the menu is set as the menu of the menubutton
	$menubutton->set_menu_model($menumodel);

	# the action related to the window (about)
	my $about_action = Glib::IO::SimpleAction->new('about',undef);
	$about_action->signal_connect('activate'=>\&about_cb);
	$window->add_action($about_action);

	$window->add($grid);

	return $window;
}

# callback function for "about"
```

```
sub about_cb {
	print "No AboutDialog for you. This is only a demonstration \n";
}


# The MAIN FUNCTION should be as small as possible and do almost nothing except creating
# your Gtk3::Application and running it
# The "real work" should always be done in response to the signals fired by Gtk3::Application.
# see below
package main;
use strict;
use warnings;

use Gtk3;
use Glib ('TRUE', 'FALSE');

my $app = Gtk3::Application->new('app.test', 'flags-none');

$app->signal_connect('startup'  => \&_init     );
$app->signal_connect('activate' => \&_build_ui );
$app->signal_connect('shutdown' => \&_cleanup  );

$app->run(\@ARGV);

exit;


# The CALLBACK FUNCTIONS to the SIGNALS fired by the main function.
# Here we do the "real work" (see above)
sub _init {
	my ($app) = @_;

	# the actions related to the application
	my $new_action = Glib::IO::SimpleAction->new('new',undef);
	$new_action->signal_connect('activate'=>\&new_cb);
	$app->add_action($new_action);

	my $quit_action = Glib::IO::SimpleAction->new('quit',undef);
	$quit_action->signal_connect('activate'=>\&quit_cb);
	$app->add_action($quit_action);
}

sub _build_ui {
	my ($app) = @_;
	my $window = MyWindow->new($app);
	$window->show_all();
}

sub _cleanup {
	my ($app) = @_;
}
```

```
# callback function for "new"
sub new_cb {
	print "This does nothing. It is only a demonstration. \n";
}

# callback function for "quit"
sub quit_cb {
	print "You have quit \n";
	$app->quit();
}
```

Useful methods for a MenuButton widget

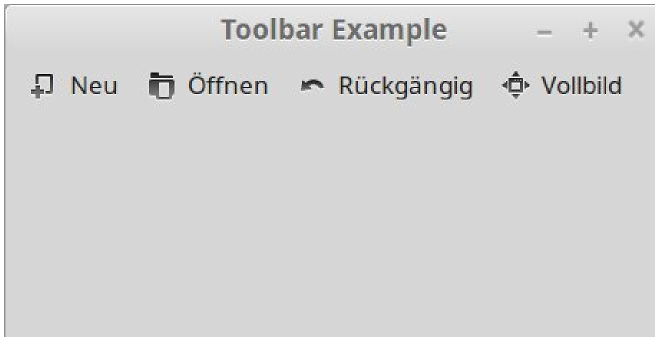
In line 57 the signal *'activate'* from the action *about_action* is connected to the callback function *about_callback()* using *$action->signal_connect('signal'=>\&callback function)*. See Signals and callbacks for a more detailed explanation.

The positioning of the menu is determined by the *'direction'* property of the menu button and the *'halign'* or *'valign'* properties of the menu. For example, when the direction is *"down"* (other option: *"up"*) and the horizontal alignment is *"start"* (other options: *"center"* and *"end"*), the menu will be positioned below the button, with the starting edge (depending on the text direction) of the menu aligned with the starting edge of the button. If there is not enough space below the button, the menu is popped up above the button instead. If the alignment would move part of the menu offscreen, it is 'pushed in'.

In the case of *vertical alignment*, the possible ArrowType *directions* are *"left"* and *"right"* and the *vertical alignment* is again *"start"*, *"center"* or *"end"*.

set_align_widget(alignment) and *set_direction(direction)* can be used to set these properties.

11. 3. Toolbar



An example of toolbar with buttons (from icon-name).

Code used to generate this example

```
#! /usr/bin/perl

# Make a binding to the Gio API in the Perl program (just copy&paste ;-))
# This is necessary mainly for Gtk3::Application and some more stuff
# Alternatively you find an early implementation as a Perl module
# on https://git.gnome.org/browse/perl-Glib-IO (not yet published on CPAN!)
# Hopefully this module simplifies the use of the Gio API in the future
# (see also the notes above).
BEGIN {
  use Glib::Object::Introspection;
  Glib::Object::Introspection->setup(
    basename => 'Gio',
    version => '2.0',
    package => 'Glib::IO');
}


# The CLASS MyWindow, where we build the Gtk3::ApplicationWindow and its content
package MyWindow;

use strict;
use warnings;

# set the pragma utf8 to prevent that umlauts of in the script created strings are displayed wrongly
use utf8;
# set the "Line Discipline" of the standard output into the UTF 8 Mode.  Thereby the terminal
# doesn't try to convert the string again to Latin-1
binmode STDOUT, ':utf8';

use Gtk3;
use Glib ('TRUE', 'FALSE');
# Our class must be a subclass of Gtk3::ApplicationWindow to inherit
# the methods, properties etc.pp. of Gtk3::ApplicationWindow
use base 'Gtk3::ApplicationWindow';

sub new {
	my ($window, $app) = @_;
```

```
	$window = bless Gtk3::ApplicationWindow->new($app);
	$window->set_title ('Toolbar Example');
	$window->set_default_size(400,200);
	$window->signal_connect( 'delete_event' => sub {$app->quit()} );

	# a grid to attach the toolbar
	my $grid = Gtk3::Grid->new();

	#CREATE THE TOOLBAR
	# a toolbar
	my $toolbar = Gtk3::Toolbar->new();

	# with extra horizontal space
	$toolbar->set_hexpand(TRUE);
	# which is the primary toolbar of the application - seems not to work in perl
	$toolbar->get_style_context()->add_class('Gtk3::STYLE_CLASS_PRIMARY_TOOLBAR'
);

	# create a button for the "new" action with a icon image
	my $new_icon = Gtk3::Image->new_from_icon_name('document-new', '16');
	my $new_button = Gtk3::ToolButton->new($new_icon,'Neu');
	# label is shown
	$new_button->set_is_important(TRUE);
	# insert the button at position in the toolbar
	$toolbar->insert($new_button,0);
	# show the button
	$new_button->show();
	# set the name of the action associated with the button.
	# The action controls the application ($app)
	$new_button->set_action_name('app.new');

	# create a button for the "open" action with a icon image
	my $open_icon = Gtk3::Image->new_from_icon_name('document-open', '16');
	my $open_button = Gtk3::ToolButton->new($open_icon,'Öffnen');
	# label is shown
	$open_button->set_is_important(TRUE);
	# insert the button at position 1 in the toolbar
	$toolbar->insert($open_button,1);
	# show the button
	$open_button->show();
	$open_button->set_action_name('app.open');

	# create a button for the "undo" action with a icon image
	my $undo_icon = Gtk3::Image->new_from_icon_name('edit-undo', '16');
	my $undo_button = Gtk3::ToolButton->new($undo_icon,'Rückgängig');
	# label is shown
	$undo_button->set_is_important(TRUE);
	# insert the button at position in the toolbar
	$toolbar->insert($undo_button,2);
	# show the button
	$undo_button->show();
	$undo_button->set_action_name('win.undo');
```

```
	# create a button for the 'fullscreen' action with a icon image
	my $fullscreen_icon = Gtk3::Image->new_from_icon_name('view-fullscreen', '16');
	my $fullscreen_button = Gtk3::ToolButton->new($fullscreen_icon,'Vollbild');
	# label is shown
	$fullscreen_button->set_is_important(TRUE);
	# insert the button at position in the toolbar
	$toolbar->insert($fullscreen_button,3);
	# show the button
	$fullscreen_button->show();
	$fullscreen_button->set_action_name('win.fullscreen');

	# show the toolbar
	$toolbar->show();

	# attach the toolbar to the grid
	$grid->attach($toolbar, 0, 0, 1, 1);

	# add the grid to the window
	$window->add($grid);

	# create the actions that control the window and connect their signal to a
	# callback method (see below)

	# undo
	my $undo_action = Glib::IO::SimpleAction->new('undo',undef);
	$undo_action->signal_connect('activate'=>\&undo_callback);
	$window->add_action($undo_action);

	# fullscreen
	my $fullscreen_action = Glib::IO::SimpleAction->new('fullscreen',undef);
	# important: We have to pass the toolbutton widget, where the signal occured, and the
	# window to the callback function as an anonym array ref!
	$fullscreen_action->signal_connect('activate'=>\&fullscreen_callback, [$fullscreen_button,
$window]);
	$window->add_action($fullscreen_action);

	return $window;
}

sub fullscreen_callback {
	# IMPORTANT: the second argument that is given is $paramter which here is undef!!!
	my ($action, $parameter, $ref) = @_;
	my $toolbutton = $$ref[0];
	my $window = $$ref[1];
	# receive the GDK Window of the MyWindow object
	my $gdk_win = $window->get_window();
	# Get the state flags of the GDK Window
	my $is_fullscreen = $gdk_win->get_state();
	# Check whether the fullscreen flag is set
	if ($is_fullscreen =~ m/fullscreen/) {
		$window->unfullscreen();
```

```
			my $fullscreen_icon = Gtk3::Image->new_from_icon_name('view-fullscreen', '16');
			$toolbutton->set_icon_widget($fullscreen_icon);
			$toolbutton->set_label('Vollbild');
			$fullscreen_icon->show();
		}
		else {
			$window->fullscreen();
			my $leave_fullscreen_icon = Gtk3::Image->new_from_icon_name('view-restore', 
'16');
			$toolbutton->set_icon_widget($leave_fullscreen_icon);
			$toolbutton->set_label('Vollbild verlassen');
			$leave_fullscreen_icon->show();
		}
}

sub undo_callback {
		my ($widget) = @_;
		print "You clicked \"Rückgängig\" \n";
		}


# The MAIN FUNCTION should be as small as possible and do almost nothing except creating
# your Gtk3::Application and running it
# The "real work" should always be done in response to the signals fired by Gtk3::Application.
# see below
package main;

use strict;
use warnings;

# set the pragma utf8 to prevent that umlauts of in the script created strings are displayed wrongly
use utf8;
# set the "Line Discipline" of the standard output into the UTF 8 Mode.  Thereby the terminal
# doesn't try to convert the string again to Latin-1
binmode STDOUT, ':utf8';

use Gtk3;
use Glib ('TRUE', 'FALSE');

my $app = Gtk3::Application->new('app.test', 'flags-none');

$app->signal_connect('startup'  => \&_init     );
$app->signal_connect('activate' => \&_build_ui );
$app->signal_connect('shutdown' => \&_cleanup  );

$app->run(\@ARGV);

exit;


# The CALLBACK FUNCTIONS to the SIGNALS fired by the main function.
# Here we do the "real work" (see above)
```

```
sub _init {
	my ($app) = @_;

	# create the actions that control the window and connect their signal
	# to a callback method (see below)

	# new
	my $new_action = Glib::IO::SimpleAction->new('new',undef);
	$new_action->signal_connect('activate'=>\&new_callback);
	$app->add_action($new_action);

	# open
	my $open_action = Glib::IO::SimpleAction->new('open',undef);
	$open_action->signal_connect('activate'=>\&open_callback);
	$app->add_action($open_action);
}

sub _build_ui {
	my ($app) = @_;
	my $window = MyWindow->new($app);
	$window->show_all();
}

sub _cleanup {
	my ($app) = @_;
}

# callback function for "new"
sub new_callback {
	print "You clicked \"Neu\". \n";
}

# callback function for "open"
sub open_callback {
	print "You clicked \"Öffnen\" \n";
}
```

Useful methods for a Toolbar widget

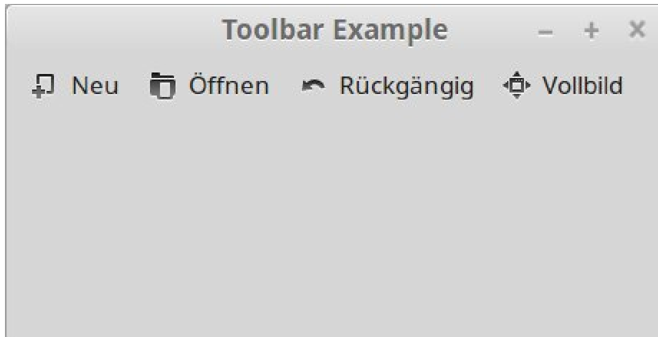
In line 114 the signal *'activate'* from the action *undo_action* is connected to the callback function *about_callback()* using *$action->signal_connect('signal'=>\&callback function)*. See Signals and callbacks for a more detailed explanation.

- Use *insert($tool_item, position)* to insert the *tool_item* at *position*. If *position* is negative, the item is appended at the end of the toolbar.
- *get_item_index($tool_item)* retrieves the position of tool_item on the toolbar.
- *get_n_items()* returns the number of items on the toolbar; *get_nth_item(position)* returns the item in position position.

- If the toolbar does not have room for all the menu items, and *set_show_arrow(TRUE)*, the items that do not have room are shown through an overflow menu.
- *set_icon_size(icon_size)* sets the size of icons in the toolbar; *icon_size* can be one of *"invalid"*, *"menu"*, *"small_toolbar"*, *"large_toolbar"*, *"button"*, *"dnd"*, *"dialog"*. This should be used only for special-purpose toolbars, normal application toolbars should respect user preferences for the size of icons. *unset_icon_size()* unsets the preferences set with *set_icon_size(icon_size)*, so that user preferences are used to determine the icon size.
- *set_style(style)*, where style is one of *"icons"*, *"text"*, *"both"*, *"both_horiz"*, sets if the toolbar shows only icons, only text, or both (vertically stacked or alongside each other). To let user preferences determine the toolbar style, and unset a toolbar style so set, use *unset_style()*.

11. 4. Tooltips (to do)

11. 5. Toolbar created using Glade

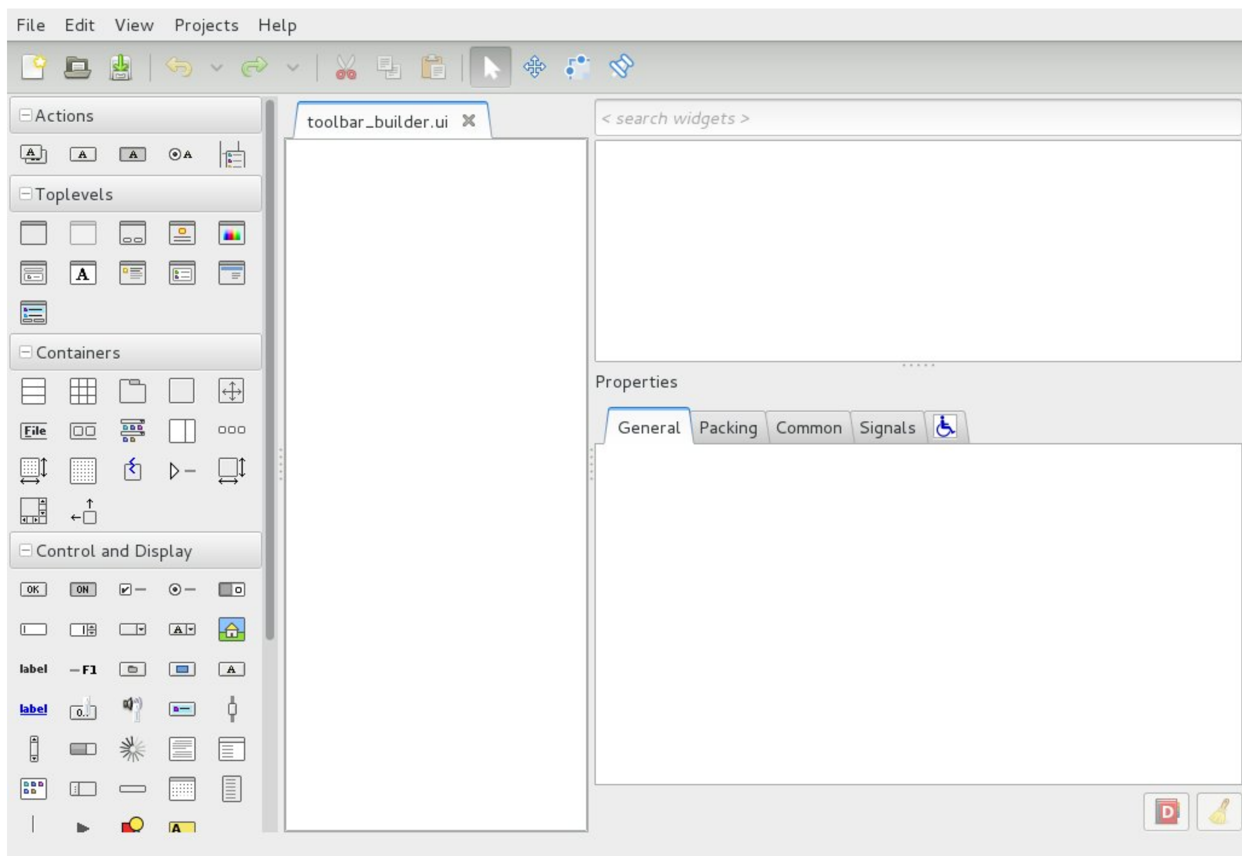


This example is similar to the Toolbar example above, except we use Glade to create the toolbar in an XML .ui file

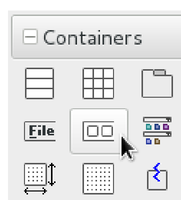
Creating the toolbar with Glade

To create the toolbar using Glade Interface Designer:

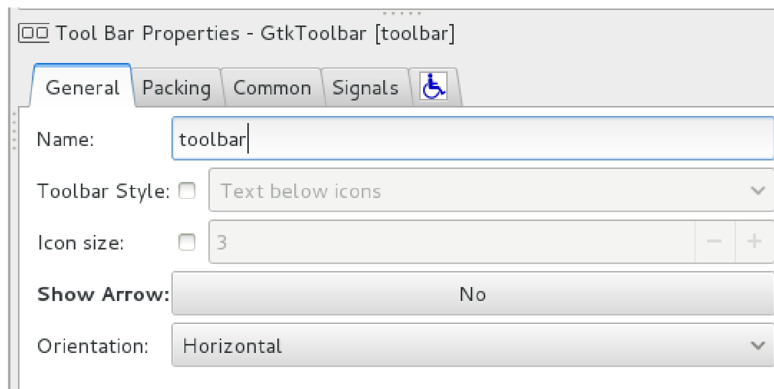
1. Open Glade and save the file as 11_5_*toolbar_builder.ui*



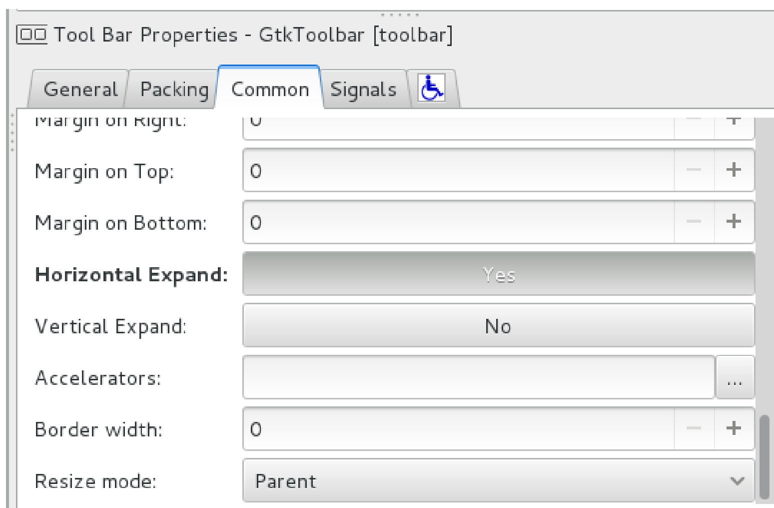
2. Under Containers on the left hand side, right click on the toolbar icon and select Add widget as toplevel.



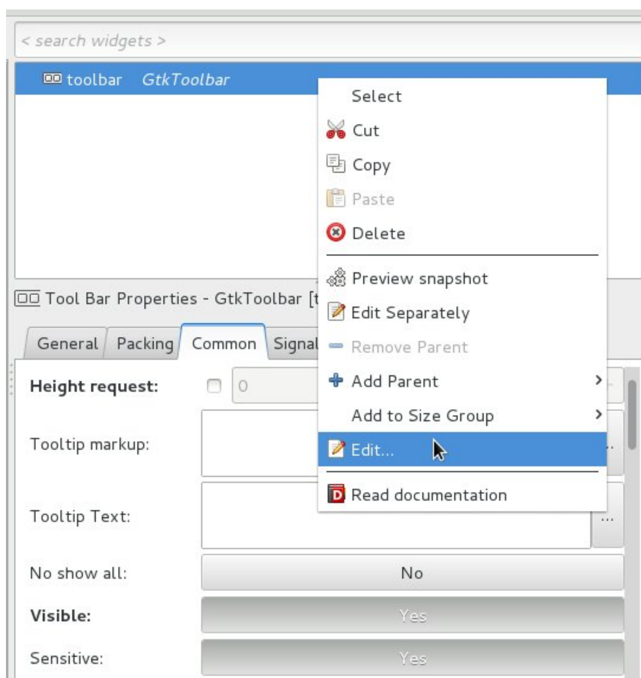
3. Under the General tab on the bottom right, change the Name to toolbar and Show Arrow to No.



4. Under the Common tab, set Horizontal Expand to Yes.



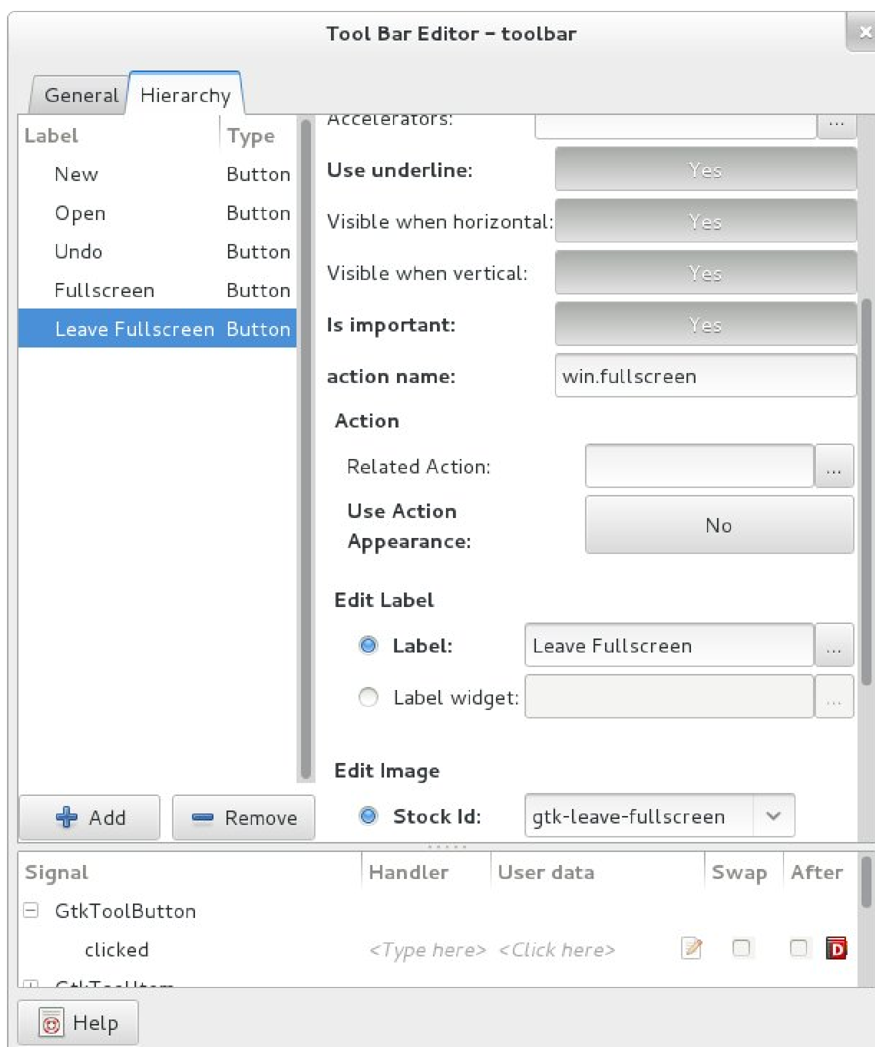
5. Right click on the toolbar in the top right and select Edit. The Tool Bar Editor window will appear.



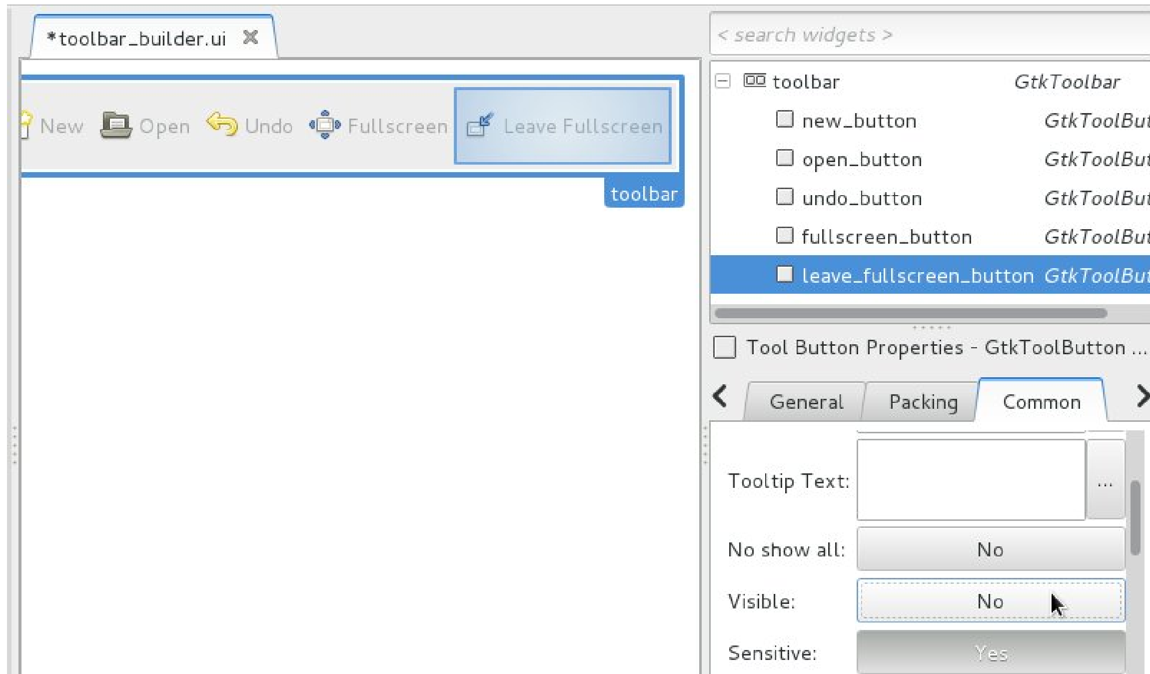
6. We want to add 5 ToolButtons: New, Open, Undo, Fullscreen and Leave Fullscreen. First, we will add the New ToolButton.
   1. Under Hierarchy tab, click Add.
   2. Change the name of the ToolItem to new_button.
   3. Scroll down and set Is important to Yes. This will cause the label of the ToolButton to be shown, when you view the toolbar.
   4. Enter the action name: app.new.
   5. Change the Label to New.
   6. Select the New Stock Id from the drop down menu, or type gtk-new.

   Repeat the above steps for the remaining ToolButtons, with the following properties:

Name	Is important	Action name	Label	Stock Id
open_button	Yes	app.open	Open	gtk-open
undo_button	Yes	win.undo	Undo	gtk-undo
fullscreen_button	Yes	win.fullscreen	Fullscreen	gtk-fullscreen
leave_fullscreen_button	Yes	win.fullscreen	Leave Fullscreen	Gtk-leave-fullscreen



7. Close the Tool Bar Editor.
8. When our program will first start, we do not want the Leave Fullscreen ToolButton to be visible, since the application will not be in fullscreen mode. You can set this in the Common tab, by clicking the Visible property to No. The ToolButton will still appear in the interface designer, but will behave correctly when the file is loaded into your program code. Note that the method show_all() would override this setting - so in the code we have to use show() separately on all the elements.



9. Save your work, and close Glade
10. The XML file created by Glade is shown below. This is the description of the toolbar. At the time of this writing, the option to add the class Gtk.STYLE_CLASS_PRIMARY_TOOLBAR in the Glade Interface did not exist. We can manually add this to the XML file. To do this, add the following XML code at line 9 of toolbar_builder.ui:

```
<style>
  <class name="primary-toolbar"/>
</style>
```

If you do not add this, the program will still work fine. The resulting toolbar will however look slightly different then the screenshot at the top of this page.

The created XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <!-- interface-requires gtk+ 3.0 -->
  <object class="GtkToolbar" id="toolbar">
    <property name="visible">True</property>
    <property name="can_focus">False</property>
    <property name="hexpand">True</property>
    <property name="show_arrow">False</property>
```

```
<child>
  <object class="GtkToolButton" id="new_button">
    <property name="use_action_appearance">False</property>
    <property name="visible">True</property>
    <property name="can_focus">False</property>
    <property name="use_action_appearance">False</property>
    <property name="is_important">True</property>
    <property name="action_name">app.new</property>
    <property name="label" translatable="yes">New</property>
    <property name="use_underline">True</property>
    <property name="stock_id">gtk-new</property>
  </object>
  <packing>
    <property name="expand">False</property>
    <property name="homogeneous">True</property>
  </packing>
</child>
<child>
  <object class="GtkToolButton" id="open_button">
    <property name="use_action_appearance">False</property>
    <property name="visible">True</property>
    <property name="can_focus">False</property>
    <property name="use_action_appearance">False</property>
    <property name="is_important">True</property>
    <property name="action_name">app.open</property>
    <property name="label" translatable="yes">Open</property>
    <property name="use_underline">True</property>
    <property name="stock_id">gtk-open</property>
  </object>
  <packing>
    <property name="expand">False</property>
    <property name="homogeneous">True</property>
  </packing>
</child>
<child>
  <object class="GtkToolButton" id="undo_button">
    <property name="use_action_appearance">False</property>
    <property name="visible">True</property>
    <property name="can_focus">False</property>
    <property name="use_action_appearance">False</property>
    <property name="is_important">True</property>
    <property name="action_name">win.undo</property>
    <property name="label" translatable="yes">Undo</property>
    <property name="use_underline">True</property>
    <property name="stock_id">gtk-undo</property>
  </object>
  <packing>
    <property name="expand">False</property>
    <property name="homogeneous">True</property>
  </packing>
</child>
<child>
```

```
      <object class="GtkToolButton" id="fullscreen_button">
        <property name="use_action_appearance">False</property>
        <property name="visible">True</property>
        <property name="can_focus">False</property>
        <property name="use_action_appearance">False</property>
        <property name="is_important">True</property>
        <property name="action_name">win.fullscreen</property>
        <property name="label" translatable="yes">Fullscreen</property>
        <property name="use_underline">True</property>
        <property name="stock_id">gtk-fullscreen</property>
      </object>
      <packing>
        <property name="expand">False</property>
        <property name="homogeneous">True</property>
      </packing>
    </child>
    <child>
      <object class="GtkToolButton" id="leave_fullscreen_button">
        <property name="use_action_appearance">False</property>
        <property name="can_focus">False</property>
        <property name="use_action_appearance">False</property>
        <property name="is_important">True</property>
        <property name="action_name">win.fullscreen</property>
        <property name="label" translatable="yes">Leave Fullscreen</property>
        <property name="use_underline">True</property>
        <property name="stock_id">gtk-leave-fullscreen</property>
      </object>
      <packing>
        <property name="expand">False</property>
        <property name="homogeneous">True</property>
      </packing>
    </child>
  </object>
</interface>
```

Code used to generate this example

We now create the code below, which adds the toolbar from the file we just created.

```
#! /usr/bin/perl

# Make a binding to the Gio API in the Perl program (just copy&paste ;-))
# This is necessary mainly for Gtk3::Application and some more stuff
# Alternatively you find an early implementation as a Perl module
# on https://git.gnome.org/browse/perl-Glib-IO (not yet published on CPAN!)
# Hopefully this module simplifies the use of the Gio API in the future
# (see also the notes above).
BEGIN {
  use Glib::Object::Introspection;
  Glib::Object::Introspection->setup(
    basename => 'Gio',
```

```
    version => '2.0',
    package => 'Glib::IO');
}


# The CLASS MyWindow, where we build the Gtk3::ApplicationWindow and its content
package MyWindow;

use strict;
use warnings;

# set the pragma utf8 to prevent that umlauts of in the script created strings are displayed wrongly
use utf8;
# set the "Line Discipline" of the standard output into the UTF 8 Mode.  Thereby the terminal
# doesn't try to convert the string again to Latin-1
binmode STDOUT, ':utf8';

use Gtk3;
use Glib ('TRUE', 'FALSE');
# Our class must be a subclass of Gtk3::ApplicationWindow to inherit
# the methods, properties etc.pp. of Gtk3::ApplicationWindow
use base 'Gtk3::ApplicationWindow';

# the following variables will be used later in a method
# (therefore we have declare outside of the new constructor
# Alternative solutions:
# 1) You can pass the widgets through an anonymous array ref as additional
# arguments to the callback function fullscreen_cb (see the pure toolbar example)
# 2) You can declare these variables with our
# 3) You can define (!) the callback function in the new constructor (not nice)
my ($fullscreen_button, $leave_fullscreen_button, $window);
sub new {
	$window = $_[0]; my $app = $_[1];
	$window = bless Gtk3::ApplicationWindow->new($app);
	$window->set_title ('Toolbar Example');
	$window->set_default_size(400,200);
	$window->signal_connect( 'delete_event' => sub {$app->quit()} );

	# a grid to attach the toolbar
	my $grid = Gtk3::Grid->new();
	$window->add($grid);
	# we have to show the grid (and therefore the toolbar) with show(),
	# as show_all() would show also the buttons in the toolbar that we want
	# to be hidden (such as the leave_fullscreen button)
	$grid->show();

	# a buidler to add the UI designed with Glade to the grid:
	my $builder = Gtk3::Builder->new();
	# get the file (if it is there)
	$builder->add_from_file('11_5_toolbar_builder.ui') or die 'file not found';
	# and attach it to the grid
	my $toolbar = $builder->get_object('toolbar');
```

```
	$grid->attach($toolbar, 0, 0, 1, 1);

	$fullscreen_button = $builder->get_object('fullscreen_button');
	$leave_fullscreen_button = $builder->get_object('leave_fullscreen_button');

	# create the actions that control the window and connect their signal to a
	# callback method (see below), add the action to the window:

	# undo
	my $undo_action = Glib::IO::SimpleAction->new('undo',undef);
	$undo_action->signal_connect('activate'=>\&undo_callback);
	$window->add_action($undo_action);

	# fullscreen
	my $fullscreen_action = Glib::IO::SimpleAction->new('fullscreen',undef);
	$fullscreen_action->signal_connect('activate'=>\&fullscreen_callback);
	$window->add_action($fullscreen_action);

	return $window;
}

# callback for undo
sub undo_callback {
	my ($action, $parameter) = @_;
	print "You clicked \"Undo\" \n";
}

# callback for fullscreen
sub fullscreen_callback {
	# receive the GDK Window of the MyWindow object
	my $gdk_win = $window->get_window();
	# Get the state flags of the GDK Window
	my $is_fullscreen = $gdk_win->get_state();
	# Check whether the fullscreen flag is set
	if ($is_fullscreen =~ m/fullscreen/) {
		$window->unfullscreen();
		$leave_fullscreen_button->hide();
		$fullscreen_button->show();
	}
	else {
		$window->fullscreen();
		$fullscreen_button->hide();
		$leave_fullscreen_button->show();
	}
}


# The MAIN FUNCTION should be as small as possible and do almost nothing except creating
# your Gtk3::Application and running it
# The "real work" should always be done in response to the signals fired by Gtk3::Application.
# see below
package main;
```

```
use strict;
use warnings;

# set the pragma utf8 to prevent that umlauts of in the script created strings are displayed wrongly
use utf8;
# set the "Line Discipline" of the standard output into the UTF 8 Mode.  Thereby the terminal
# doesn't try to convert the string again to Latin-1
binmode STDOUT, ':utf8';

use Gtk3;
use Glib ('TRUE', 'FALSE');

my $app = Gtk3::Application->new('app.test', 'flags-none');

$app->signal_connect('startup'  => \&_init     );
$app->signal_connect('activate' => \&_build_ui );
$app->signal_connect('shutdown' => \&_cleanup  );

$app->run(\@ARGV);

exit;


# The CALLBACK FUNCTIONS to the SIGNALS fired by the main function.
# Here we do the "real work" (see above)
sub _init {
	my ($app) = @_;

	# create the actions that control the application: create, connect their signal
	# to a callback method (see below), add the action to the application

	# new
	my $new_action = Glib::IO::SimpleAction->new('new',undef);
	$new_action->signal_connect('activate'=>\&new_callback);
	$app->add_action($new_action);

	# open
	my $open_action = Glib::IO::SimpleAction->new('open',undef);
	$open_action->signal_connect('activate'=>\&open_callback);
	$app->add_action($open_action);
}

sub _build_ui {
	my ($app) = @_;
	my $window = MyWindow->new($app);
	# show the window - with show() not show_all() because that would show also
	# the leave_fullscreen_button
	$window->show();
}

sub _cleanup {
	my ($app) = @_;
```

```
}

# callback function for "new"
sub new_callback {
	print "You clicked \"New\". \n";
}

# callback function for "open"
sub open_callback {
	print "You clicked \"Open\" \n";
}
```

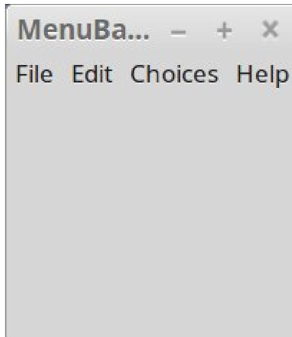
Useful methods for Gtk3::Builder

For the useful methods for a Toolbar widget, see the Toolbar Chapter.

Gtk3::Builder builds an interface from an XML UI definition.

- *add_from_file('filename')* loads and parses the given file and merges it with the current contents of the Gtk3::Builder.
- *add_from_string('string')* parses the given string and merges it with the current contents of the Gtk3::Builder.
- *add_objects_from_file('filename', 'object_ids')* is the same as *add_from_file()*, but it loads only the objects with the ids given in the *object_ids* list.
- *add_objects_from_string('string', 'object_ids')* is the same as *add_from_string()*, but it loads only the objects with the ids given in the *object_ids* list.
- *get_object('object_id')* retrieves the widget with the id *object_id* from the loaded objects in the builder.
- *get_objects()* returns all loaded objects.
- *connect_signals(handler_object)* connects the signals to the methods given in the *handler_object*. This can be any object which contains keys or attributes that are called like the signal handler names given in the interface description, e.g. a class or a hash. In line 74 the signal *'activate'* from the action *undo_action* is connected to the callback function *undo_callback()* using *$action->signal_connect(signal=>\&callback function)*. See Signals and callbacks for a more detailed explanation.

11. 6. MenuBar created using XML and GtkBuilder (to do)



A MenuBar created using XML and Gtk3::Builder

Create a MenuBar using XML

To create the menubar using XML:

1. Create *11_6_menubar.ui* using your favorite text editor.
2. Enter the following line at the top of the file:

```
<?xml version="1.0"? encoding="UTF-8"?>
```

3. We want to create the interface which will contain our menubar and its submenus. Our menubar will contain File, Edit, Choices and Help submenus. We add the following XML code to the file:

```
<?xml version="1.0"? encoding="UTF-8"?>
<interface>
	<menu id="menubar">
		<submenu>
			<attribute name="label">File</attribute>
		</submenu>
		<submenu>
			<attribute name="label">Edit</attribute>
		</submenu>
		<submenu>
			<attribute name="label">Choices</attribute>
		</submenu>
		<submenu>
			<attribute name="label">Help</attribute>
		</submenu>
	</menu>
</interface>
```

4. Now we will create the .pl file and use Gtk3::Builder to import the *11_6_menubar.ui* we just created

Add the MenuBar to the window using Gtk3::Builder

```
#! /usr/bin/perl

# Make a binding to the Gio API in the Perl program (just copy&paste ;-))
BEGIN {
  use Glib::Object::Introspection;
  Glib::Object::Introspection->setup(
    basename => 'Gio',
    version => '2.0',
    package => 'Glib::IO');
}

# The CLASS MyWindow, where we build the Gtk3::ApplicationWindow and its content
package MyWindow;
use strict;
use warnings;
use Gtk3;
use Glib ('TRUE', 'FALSE');
# Our class must be a subclass of Gtk3::ApplicationWindow to inherit
# the methods, properties etc.pp. of Gtk3::ApplicationWindow
use base 'Gtk3::ApplicationWindow';

sub new {
	my ($window, $app) = @_;
	$window = bless Gtk3::ApplicationWindow->new($app);
	$window->set_title ('MenuBar Example');
	$window->set_default_size (200, 200);
	$window->signal_connect( 'delete_event' => sub {$app->quit()} );

	return $window;
}

# The MAIN FUNCTION should be as small as possible and do almost nothing except creating
# your Gtk3::Application and running it
# The "real work" should always be done in response to the signals fired by Gtk3::Application.
# see below
package main;
use strict;
use warnings;

use Gtk3;
use Glib ('TRUE', 'FALSE');

my $app = Gtk3::Application->new('app.test', 'flags-none');

$app->signal_connect('startup'  => \&_init);
$app->signal_connect('activate' => \&_build_ui);
$app->signal_connect('shutdown' => sub {$app->quit();});

$app->run(\@ARGV);

exit;
```

```
# The CALLBACK FUNCTIONS to the SIGNALS fired by the main function.
# Here we do the "real work" (see above)
sub _init {
	my ($app) = @_;

	# A builder to add the UI designed with glade to the grid
	my $builder = Gtk3::Builder->new();
	$builder->add_from_file('11_6_menubar-firststep.ui') or die 'file not found';

	# we use the method Gtk3::Application->set_menubar('menubar')
	# to add the menubar to the application (Note: NOT the window!)
	my $menubar=$builder->get_object('menubar');
	$app->set_menubar($menubar);

 }

sub _build_ui {
	my ($app) = @_;
	# Building the Gtk3::ApplicationWindow and its content is here done
	# by a seperate class (see above)
	my $window = MyWindow->new($app);
	$window->show();
}
```

Now run the perl application. It should look like the picture at the top of this chapter.

Add items to the menus

We start off by adding 2 menuitems to the File menu: New and Quit. We do this by adding a *section* to the *File* submenu with these items: The 11_6_*menubar.ui* should look like this (lines 6 to 13 inclusive comprise the newly added section)

menubar.ui

```
<?xml version="1.0"? encoding="UTF-8"?>
<interface>
	<menu id="menubar">
		<submenu>
			<attribute name="label">File</attribute>
			<section>
				<item>
					<attribute name="label">New</attribute>
				</item>
				<item>
					<attribute name="label">Quit</attribute>
				</item>
			</section>
		</submenu>
		<submenu>
			<attribute name="label">Edit</attribute>
		</submenu>
		<submenu>
```

```
                    <attribute name="label">Choices</attribute>
                </submenu>
                <submenu>
                    <attribute name="label">Help</attribute>
                </submenu>
        </menu>
</interface>
```

Following this pattern, you can now add a Copy and a Paste item to the Edit submenu, and an About item to the Help submenu.

Setup actions

We now create the actions for "New" and "Quit" connected to a callback function in the Python file; for instance we create "new" as:

```
my $new_action = Glib::IO::SimpleAction->new('new',undef);
$new_action->signal_connect('activate'=>\&new_callback);
```

And we create the callback function of 'new' as:

```
sub new_callback {
        print "You clicked \"New \" \n";
}
```

Now, in the XML file, we connect the menu items to the actions in the XML file by adding the "action" attribute:

```
<item>
        <attribute name="label">New</attribute>
        <attribute name="action">app.new</attribute>
</item>
```

Note that for an action that is relative to the application, we use the prefix *"app."*; for actions that are relative to the window we use the prefix *"win."*.

Finally, in the Python file, we add the action to the application or to the window - so for instance *app.new* will be added to the application in the method *_init()* as

```
$app->add_action($new_action);
```

See the chapter Signals and callbacks for a more detailed explanation of signals and callbacks.

Actions: Application or Window?

Above, we created the "new" and "open" actions as part of the MyApplication class. Actions which control the application itself, such as "quit" should be created similarly.

Some actions, such as "copy" and "paste" deal with the window, not the application. Window actions should be created as part of the window class.

The complete example files contain both application actions and window actions. The window actions are the ones usually included in the *application menu* also. It is not good practice to include window actions in the application menu. For demonstration purposes, the complete example files

which follow include XML in the UI file which creates the application menu which includes a "New" and "Open" item, and these are hooked up to the same actions as the menubar items of the same name.

Choices submenu and items with state

Lines 30 to 80 inclusive of the Complete XML UI file for this example demonstrate the XML code used to create the UI for Choices menu.

The actions created so far are *stateless*, that is they do not retain or depend on a state given by the action itself. The actions we need to create for the Choices submenu, on the other hand, are *stateful*. An example of creation of a stateful action is:

```
my $shape_action = Glib::IO::SimpleAction->new_stateful('shape', Glib::VariantType->new('s'),
Glib::Variant->new_string('line'));
```

where the variables of the method are: *name, parameter type* (in this case, a string - see here for a complete list of character meanings), *initial state* (in this case, 'line' - in case of a True boolean value it should be *Glib::Variant->new_boolean(TRUE)*, and so on, see here for a complete list)

After creating the stateful SimpleAction we connect it to the callback function and we add it to the window (or the application, if it is the case), as before:

```
$shape_action->signal_connect('activate'=>\&shape_callback);
$window->add_action($shape_action);
```

Complete XML UI file for this example

```
<?xml version="1.0"? encoding="UTF-8"?>
<interface>
	<menu id="menubar">
		<submenu>
			<attribute name="label">File</attribute>
			<section>
				<item>
					<attribute name="label">New</attribute>
					<attribute name="action">app.new</attribute>
				</item>
				<item>
					<attribute name="label">Quit</attribute>
					<attribute name="action">app.quit</attribute>
				</item>
			</section>
		</submenu>
		<submenu>
			<attribute name="label">Edit</attribute>
			<section>
				<item>
					<attribute name="label">Copy</attribute>
					<attribute name="action">win.copy</attribute>
				</item>
				<item>
					<attribute name="label">Paste</attribute>
					<attribute name="action">win.paste</attribute>
```

```
            </item>
        </section>
    </submenu>
    <submenu>
        <attribute name="label">Choices</attribute>
        <submenu>
        <attribute name="label">Shapes</attribute>
          <section>
            <item>
              <attribute name="label">Line</attribute>
              <attribute name="action">win.shape</attribute>
              <attribute name="target">line</attribute>
            </item>
            <item>
              <attribute name="label">Triangle</attribute>
              <attribute name="action">win.shape</attribute>
              <attribute name="target">triangle</attribute>
            </item>
            <item>
              <attribute name="label">Square</attribute>
              <attribute name="action">win.shape</attribute>
              <attribute name="target">square</attribute>
            </item>
            <item>
              <attribute name="label">Polygon</attribute>
              <attribute name="action">win.shape</attribute>
              <attribute name="target">polygon</attribute>
            </item>
            <item>
              <attribute name="label">Circle</attribute>
              <attribute name="action">win.shape</attribute>
              <attribute name="target">circle</attribute>
            </item>
          </section>
      </submenu>
      <section>
        <item>
          <attribute name="label">On</attribute>
          <attribute name="action">app.state</attribute>
          <attribute name="target">on</attribute>
        </item>
        <item>
          <attribute name="label">Off</attribute>
          <attribute name="action">app.state</attribute>
          <attribute name="target">off</attribute>
        </item>
      </section>
      <section>
        <item>
          <attribute name="label">Awesome</attribute>
          <attribute name="action">app.awesome</attribute>
        </item>
```

```
                    </section>
                </submenu>
                <submenu>
                        <attribute name="label">Help</attribute>
                        <section>
                                <item>
                                        <attribute name="label">About</attribute>
                                        <attribute name="action">win.about</attribute>
                                </item>
                        </section>
                </submenu>
        </menu>
        <menu id="appmenu">
                <section>
                        <item>
                                <attribute name="label">New</attribute>
                                <attribute name="action">app.new</attribute>
                    </item>
                    <item>
                                <attribute name="label">Quit</attribute>
                                <attribute name="action">app.quit</attribute>
                    </item>
                </section>
        </menu>
</interface>
```

Complete Perl file for this example

```
#! /usr/bin/perl

# Make a binding to the Gio API in the Perl program (just copy&paste ;-))
# This is necessary mainly for Gtk3::Application and some more stuff
# Alternatively you find an early implementation as a Perl module
# on https://git.gnome.org/browse/perl-Glib-IO (not yet published on CPAN!)
# Hopefully this module simplifies the use of the Gio API in the future
# (see also the notes above).
BEGIN {
  use Glib::Object::Introspection;
  Glib::Object::Introspection->setup(
    basename => 'Gio',
    version => '2.0',
    package => 'Glib::IO');
}

# The CLASS MyWindow, where we build the Gtk3::ApplicationWindow and its content
package MyWindow;
use strict;
use warnings;
use Gtk3;
use Glib ('TRUE', 'FALSE');
# Our class must be a subclass of Gtk3::ApplicationWindow to inherit
```

```
# the methods, properties etc.pp. of Gtk3::ApplicationWindow
use base 'Gtk3::ApplicationWindow';

sub new {
	my ($window, $app) = @_;
	$window = bless Gtk3::ApplicationWindow->new($app);
	$window->set_title ('MenuBar Example');
	$window->set_default_size (200, 200);
	$window->signal_connect( 'delete_event' => sub {$app->quit()} );

	# action without a state created (name, parameter type)
	my $copy_action = Glib::IO::SimpleAction->new('copy',undef);
	# connected with the callback function
	$copy_action->signal_connect('activate'=>\&copy_callback);
	# added to the window
	$window->add_action($copy_action);

	# action without a state created (name, parameter type)
	my $paste_action = Glib::IO::SimpleAction->new('paste',undef);
	# connected with the callback function
	$paste_action->signal_connect('activate'=>\&paste_callback);
	# added to the window
	$window->add_action($paste_action);

	# action with a state created
	# options: (name, parameter type, initial state)
	my $shape_action = Glib::IO::SimpleAction->new_stateful('shape',
Glib::VariantType->new('s'), Glib::Variant->new_string('line'));
	# connected to the callback function
	$shape_action->signal_connect('activate'=>\&shape_callback);
	# added to the windows
	$window->add_action($shape_action);

	# action with a state created
	my $about_action = Glib::IO::SimpleAction->new('about',undef);
	# action connected to the callback function
	$about_action->signal_connect('activate'=>\&about_callback);
	# action added to the application
	$window->add_action($about_action);

	return $window;
}

 # callback function for the copy action
sub copy_callback {
	print "\"Copy\" activated \n";
}
# callback function for the paste action
sub paste_callback {
	print "\"Paste\" activated \n";
}
# callback function for the shape action
```

```
sub shape_callback {
	my ($action, $parameter) = @_;
	my $string = $parameter->get_string();
	print "Shape is set to $string \n";

	# Note that we set the state of the action!
	$action->set_state($parameter);
}

# callback function for about (see AboutDialog example)
sub about_callback {
	my ($action, $parameter) = @_;
	print "About activated \n";
}

# The MAIN FUNCTION should be as small as possible and do almost nothing except creating
# your Gtk3::Application and running it
# The "real work" should always be done in response to the signals fired by Gtk3::Application.
# see below
package main;

use strict;
use warnings;

use Gtk3;
use Glib ('TRUE', 'FALSE');

my $app = Gtk3::Application->new('app.test', 'flags-none');

$app->signal_connect('startup'  => \&_init     );
$app->signal_connect('activate' => \&_build_ui );
$app->signal_connect('shutdown' => \&_cleanup  );

$app->run(\@ARGV);

exit;

# The CALLBACK FUNCTIONS to the SIGNALS fired by the main function.
# Here we do the "real work" (see above)
sub _init {
	my ($app) = @_;

	# action without a state created
	my $new_action = Glib::IO::SimpleAction->new('new',undef);
	# action connected to the callback function
	$new_action->signal_connect('activate'=>\&new_callback);
	# action added to the application
	$app->add_action($new_action);
	# action without a state created
	my $quit_action = Glib::IO::SimpleAction->new('quit',undef);
	# action connected to the callback function
	$quit_action->signal_connect('activate'=>\&quit_callback);
```

```
	# action added to the application
	$app->add_action($quit_action);

	# action with a state created
	my $state_action = Glib::IO::SimpleAction->new_stateful('state',
Glib::VariantType->new('s'), Glib::Variant->new_string('off'));
	# action connected to the callback function
	$state_action->signal_connect('activate'=>\&state_callback);
	# action added to the application
	$app->add_action($state_action);

	# action with a state created
	my $awesome_action = Glib::IO::SimpleAction->new_stateful('awesome', undef,
Glib::Variant->new_boolean(TRUE));
	# action connected to the callback function
	$awesome_action->signal_connect('activate'=>\&awesome_callback);
	# action added to the application
	$app->add_action($awesome_action);

	# A builder to add the UI designed with glade to the grid
	my $builder = Gtk3::Builder->new();
	$builder->add_from_file('11_6_menubar.ui') or die 'file not found';

	# we use the method Gtk3::Application->set_menubar('menubar')
	# to add the menubar to the application (Note: NOT the window!)
	my $menubar=$builder->get_object('menubar');
	$app->set_menubar($menubar);

}

sub _build_ui {
	my ($app) = @_;
	# Building the Gtk3::ApplicationWindow and its content is here done
	# by a seperate class (see above)
	my $window = MyWindow->new($app);
	$window->show();
}

sub _cleanup {
	my ($app) = @_;
}

# callback function for new
sub new_callback {
	print "You clicked \"New \" \n";
}

# callback function for quit
sub quit_callback {
	print "You clicked \"Quit\" \n";
	$app->quit();
}
```

```
# callback function for state
sub state_callback {
	my ($action, $parameter) = @_;
	my $string = $parameter->get_string();
	print "State is set to $string \n";

	$action->set_state($parameter);
}

# callback function for awesome
sub awesome_callback {
	my ($action, $parameter) = @_;
	my $state = $action->get_state();
	my $boolean = $state->get_boolean();
	if ($boolean) {
		print "You unchecked \"Awesome\" \n";
		$action->set_state(Glib::Variant->new_boolean(FALSE));
	}
	else {
		print "You checked \"Awesome\" \n";
		$action->set_state(Glib::Variant->new_boolean(TRUE));
	}
}
```

Mnemonics and Accelerators

Labels may contain mnemonics. Mnemonics are underlined characters in the label, used for keyboard navigation. Mnemonics are created by placing an underscore before the mnemonic character. For example "_File" instead of just "File" in the 11_6_menubar.ui label attribute.

The mnemonics are visible when you press the Alt key. Pressing Alt+F will open the File menu.

Accelerators can be explicitly added in the UI definitions. For example, it is common to be able to quit an application by pressing Ctrl+Q or to save a file by pressing Ctrl+S. To add an accelerator to the UI definition, you simply need add an "accel" attribute to the item.

<attribute name="accel"><Primary>q</attribute> will create the Ctrl+Q sequence when added to the Quit label item. Here, "Primary" refers to the Ctrl key on a PC or the ⌘ key on a Mac.

```
<item>
	<attribute name="label">_Quit</attribute>
	<attribute name="action">app.quit</attribute>
	<attribute name="accel">&lt;Primary&gt;q</attribute>
</item>
```

Translatable strings

Since GNOME applications are being translated into [many languages](), it is important that the strings in your application are translatable. To make a label translatable, simple set *translatable="yes"*:

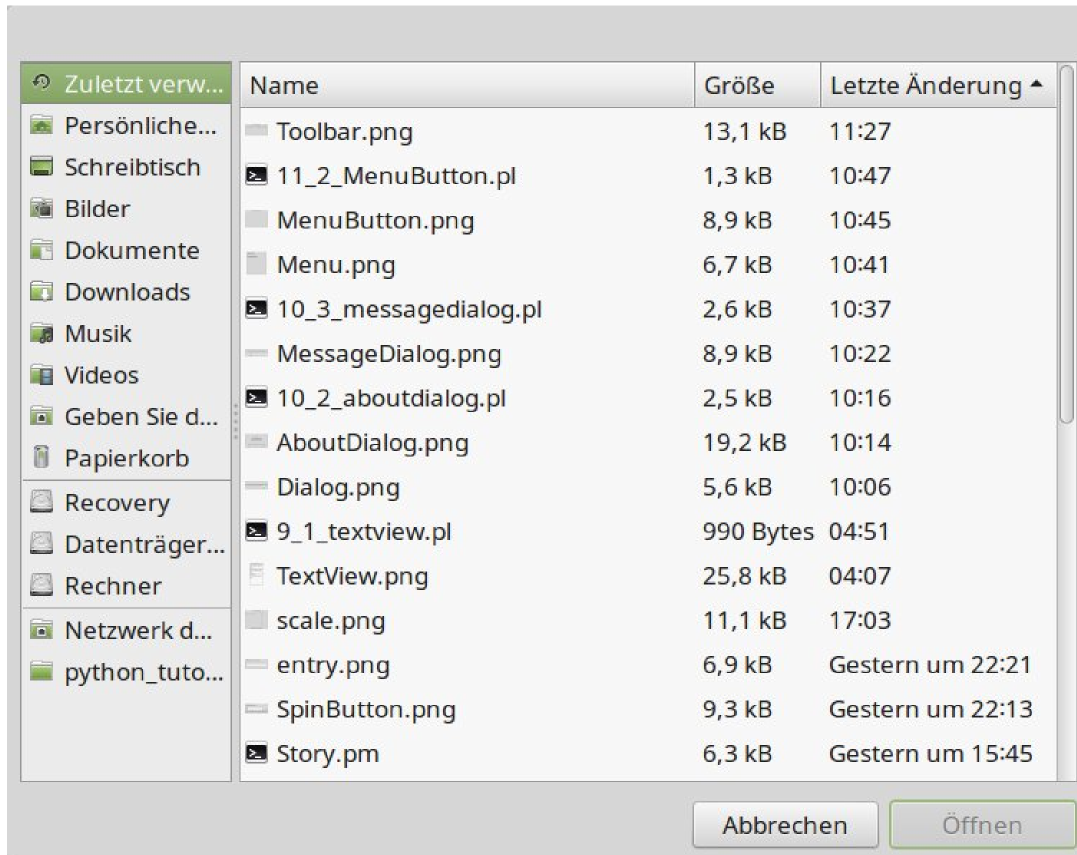
```
<attribute name="label" translatable="yes">Quit</attribute>
```

12. Selectors

12. 1. ColorButton (to do)

12. 2. FontChooserWidget (to do

12. 3. FileChooserDialog



This FileChooserDialog saves a text document, which can be opened or written from scratch in a TextView.

It is also possible to call a FileChooserDialog to open a new document.

Steps to recreate the example

1. Create a file *12_3_filechooser.ui* to describe an app-menu with items "New", "Open", "Save", "Save as", and "Quit". This can be done with Glade or in a text editor. See XML file which creates the app-menu below.

2. Create a Perl program for a Gtk3::TextView with a Gtk3::Buffer $buffer, and a $file which will be a Glib::IO::File and we set initially as *undef*.

3. In this program, create also the actions corresponding to the items in the app-menu, connect them to callback functions, and import the menu in the *_init()* method with a *Gtk3::Builder*.

4. "New" and "Quit" actions and callback functions are quite straightforward, see **Code used to generate this example**. See Signals and callbacks for a more detailed explanation of signals and callback functions.

5. "Open" callback should create and open a Gtk3::FileChooserDialog for "Open", connected with another callback function for each of the two "Open" and "Cancel" buttons of the FileChooserDialog.

6. "Save as" works basically as "Open", but the callback function of the "Save" button depends on a more complex method *save_to_file()*.

7. "Save" can be reduced to the case where the file is None, that is the case where $file is a new file, which in turn is the case "Save as"; and to the case where the $file is not None, which in turn is reduced to *save_to_file()*.

8. Finally, the method *save_to_file()*: see Code used to generate this example, lines 230 - 267.

XML file which creates the app-menu

```
<?xml version="1.0"?>
<interface>
  <menu id="appmenu">
    <section>
      <item>
        <attribute name="label">New</attribute>
        <attribute name="action">win.new</attribute>
      </item>
      <item>
        <attribute name="label">Open</attribute>
        <attribute name="action">win.open</attribute>
      </item>
    </section>
    <section>
      <item>
        <attribute name="label">Save</attribute>
        <attribute name="action">win.save</attribute>
      </item>
      <item>
        <attribute name="label">Save As...</attribute>
        <attribute name="action">win.save-as</attribute>
      </item>
    </section>
    <section>
      <item>
        <attribute name="label">Quit</attribute>
        <attribute name="action">app.quit</attribute>
      </item>
    </section>
  </menu>
</interface>
```

Code used to generate this example

```
#! /usr/bin/perl

# Make a binding to the Gio API in the Perl program (just copy&paste ;-))
# This is necessary mainly for Gtk3::Application and some more stuff
```

```
# Alternatively you find an early implementation as a Perl module
# on https://git.gnome.org/browse/perl-Glib-IO (not yet published on CPAN!)
# Hopefully this module simplifies the use of the Gio API in the future, so that for example the still
# necessary converting of the bytes in a bytestring would be redundant (see also the notes above).
BEGIN {
  use Glib::Object::Introspection;
  Glib::Object::Introspection->setup(
    basename => 'Gio',
    version => '2.0',
    package => 'Glib::IO');
}


# The CLASS MyWindow, where we build the Gtk3::ApplicationWindow and its content
package MyWindow;
use strict;
use warnings;
use utf8;
use Gtk3;
use Glib ('TRUE', 'FALSE');
use Encode;
# Our class must be a subclass of Gtk3::ApplicationWindow to inherit
# the methods, properties etc.pp. of Gtk3::ApplicationWindow
use base 'Gtk3::ApplicationWindow';

# MyWindow instance variables (we need them in the callback functions):
my $window;
my $buffer;
my $file;

 sub new {
	$window = $_[0]; my $app = $_[1];
	$window = bless Gtk3::ApplicationWindow->new($app);
	$window->set_title ('FileChooserDialog Example');
	$window->set_default_size(400,400);
	$window->signal_connect( 'delete_event' => sub {$app->quit()} );

	# the file
	$file = undef;

	# the textview with the buffer
	$buffer = Gtk3::TextBuffer->new();
	my $textview = Gtk3::TextView->new();
	$textview->set_buffer($buffer);
	$textview->set_wrap_mode('word');

	# a scrolled window for the textview
	my $scrolled_window = Gtk3::ScrolledWindow->new();
	$scrolled_window->set_policy('automatic', 'automatic');
	$scrolled_window->add($textview);
	$scrolled_window->set_border_width(5);
```

```
	# add the scrolled window to the window
	$window->add($scrolled_window);

	# the actions for the window menu, connected to the callback functions
	my $new_action = Glib::IO::SimpleAction->new('new',undef);
	$new_action->signal_connect('activate'=>\&new_callback);
	$window->add_action($new_action);

	my $open_action = Glib::IO::SimpleAction->new('open',undef);
	$open_action->signal_connect('activate'=>\&open_callback);
	$window->add_action($open_action);

	my $save_action = Glib::IO::SimpleAction->new('save',undef);
	$save_action->signal_connect('activate'=>\&save_callback);
	$window->add_action($save_action);

	my $save_as_action = Glib::IO::SimpleAction->new('save-as',undef);
	$save_as_action->signal_connect('activate'=>\&save_as_callback);
	$window->add_action($save_as_action);

	return $window;
}

# callback for new
sub new_callback {
	my ($action, $parameter) = @_;
	$buffer->set_text('');
	print "New file created \n";
}

# callback for open
sub open_callback {
	my ($action, $parameter) = @_;
	# create a filechooserdialog to open:
	# the arguments are: title of the window, parent_window, action
	# (buttons, response)
	my $open_dialog = Gtk3::FileChooserDialog->new('Pick a file',
							$window,
							'open',
							('gtk-cancel', 'cancel',
							'gtk-open', 'accept'));

	# not only local files can be selected in the file selector
	$open_dialog->set_local_only(FALSE);

	# dialog always on top of the textview window
	$open_dialog->set_modal(TRUE);

	# connect the dialog with the callback function open_response_cb()
	$open_dialog->signal_connect('response' => \&open_response_cb);
```

```
	# show the dialog
	$open_dialog->show();

}

# callback function for the dialog open_dialog
sub open_response_cb {
	my ($dialog, $response_id) = @_;
	my $open_dialog = $dialog;
	# if response id is 'ACCEPTED' (the button 'Open' has been clicked)
	if ($response_id eq 'accept') {
		print "accept was clicked \n";

		# $file is the file that we get from the FileChooserDialog
		$file = $open_dialog->get_file();

		# load the content of the file into memory:
		# success is a boolean depending on the success of the operation
		# content is self-explanatory
		# etags is an entity tag (can be used to quickly determine if the
		# file has been modified from the version on the file system)
		my ($success, $content, $etags) = Glib::IO::File::load_contents($file) or print
"Error: Open failed \n";

		# NOTE: GIO reads and writes files in raw bytes format,
		# which means everything is passed on without any encoding/decoding.
		# We have to convert these data so that Perl can understand them.
		# First we convert the bytes (= pure digits) to a bytestring without encoding
		$content = pack 'C*', @{$content};
		# Then we decode this bytestrng in the utf8 encoding format
		my $content_utf8 = decode('utf-8', $content);

		# important: we need the length in byte!! Usually the perl function
		# length deals in logical characters, not in physical bytes! For how
		# many bytes a string encoded as UTF-8 would take up, we have to use
		# length(Encode::encode_utf8($content) (for that we have to 'use Encode'
		# first [see more http://perldoc.perl.org/functions/length.html]
		# Alternative solutions:
		# 1) For insert all the text set length in the $buffer->set_text method
		#	to -1 (!text must be nul-terminated)
		# 2) In Perl Gtk3 the length argument is optional! Without length
		#	all text is inserted
		my $length = length(Encode::encode_utf8($content_utf8));
		$buffer->set_text($content_utf8, $length);

		my $filename = $open_dialog->get_filename();
		print "opened: $filename \n";

		$dialog->destroy();
		}
	# if response id is 'CANCEL' (the button 'Cancel' has been clicked)
	elsif ($response_id eq 'cancel') {
```

```
			print "cancelled: Gtk3::FileChooserAction::OPEN \n";
			$dialog->destroy();
			}
}

# callback function for save_as
sub save_as_callback {
	# create a filechooserdialog to save:
	# the arguments are: title of the window, parent_window, action,
	# (buttons, response)
	my $save_dialog = Gtk3::FileChooserDialog->new('Pick a file',
							$window,
							'save',
							('gtk-cancel', 'cancel',
							'gtk-save', 'accept'));
	# the dialog will present a confirmation dialog if the user types a file name
	# that already exists
	$save_dialog->set_do_overwrite_confirmation(TRUE);
	# dialog always on top of the textview window
	$save_dialog->set_modal(TRUE);

	if ($file) {
	$save_dialog->select_file($file) or print "Error Selecting file failed\n";
	}

	# connect the dialog to the callback function save_response_cb
	$save_dialog->signal_connect('response' => \&save_response_cb);

	# show the dialog
	$save_dialog->show();
}

# callback function for the dialog save_dialog
sub save_response_cb {
	my ($dialog, $response_id) = @_;
	my $save_dialog = $dialog;

	# if $response_id is 'ACCEPT' (= the button 'Save' has been clicked)
	if ($response_id eq 'accept') {
		# get the currently selected file
		# more modern is to use the GFile Method $file->get_file (see above)
		$file = $save_dialog->get_file();

		# save to file (see below)
		save_to_file();

		# destroy the FileChooserDialog
		$dialog->destroy;
	}
	elsif ($response_id eq 'cancel') {
		print "cancelled: FileChooserAction.SAVE \n";
		# destroy the FileChooserDialog
```

```
			$dialog->destroy;
		}
}

# callback function for save
sub save_callback {
		my ($action, $parameter) = @_;
		# if $file is not already there
		if ($file) {
			save_to_file();
		}
		# $file is a new file
		else {
			# use save_as
			save_as_callback($action, $parameter);
		}
}

# save_to_file
sub save_to_file {
		# get the content of the buffer, without hidden characters
		my ($start, $end) = $buffer->get_bounds();
		my $current_contents = $buffer->get_text($start, $end, FALSE);
		if ($current_contents) {
			# NOTE AGAIN: Gio reads and writes files in raw bytes format (see above)
			# Therefore the method replace_readwrite expects an array reference containing
			# raw bytes!!! So we have to convert the perlish content into an array ref
			# containing the raw bytes as follows:

			# First we have to reconvert the textstring with the utf8 encoding format to
			# to a bytestring
			my $content_utf8 = encode('utf-8', $current_contents);
			# Then we convert the bytestring in bytes and give a array reference to the function
			my @contents = unpack 'C*', $content_utf8;

			# set the content as content of $file
			# arguments: contents, etags, make_backup, flags, GError
			$file->replace_contents(\@contents,
						undef,
						FALSE,
						'none',
						undef) or print "Error: Saving failed\n";
			my $path = $file->get_path();
			print "saved $path \n";
		}
		# if the contents are empty
		else {
			# create (if the file does not exist) or overwrite the file in readwrite mode.
			# arguments: etags, make_backup, flags, GError
			$file->replace_readwrite(undef,
						FALSE,
						'none',
```

```
                                        undef);
            my $path = $file->get_path();
            print "saved $path \n";
      }
}


# The MAIN FUNCTION should be as small as possible and do almost nothing except creating
# your Gtk3::Application and running it
# The "real work" should always be done in response to the signals fired by Gtk3::Application.
# see below
package main;

use strict;
use warnings;
use utf8;

use Gtk3;
use Glib ('TRUE', 'FALSE');

my $app = Gtk3::Application->new('app.test', 'flags-none');

$app->signal_connect('startup'  => \&_init     );
$app->signal_connect('activate' => \&_build_ui );
$app->signal_connect('shutdown' => \&_cleanup  );

$app->run(\@ARGV);

exit;


# The CALLBACK FUNCTIONS to the SIGNALS fired by the main function.
# Here we do the "real work" (see above)
sub _init {
      my ($app) = @_;

      # app action "quit", connected to the callback function
      my $quit_action = Glib::IO::SimpleAction->new('quit',undef);
      $quit_action->signal_connect('activate'=>\&quit_callback);
      $app->add_action($quit_action);

      # get the menu from the ui file with a builder
      my $builder = Gtk3::Builder->new();
      $builder->add_from_file('12_3_filechooserdialog.ui') or die "file not found \n";
      my $menu = $builder->get_object('appmenu');
      $app->set_app_menu($menu);
}

sub _build_ui {
      my ($app) = @_;
      my $window = MyWindow->new($app);
      $window->show_all();
```

```
}

sub _cleanup {
	my ($app) = @_;
}

# callback function for "quit"
sub quit_callback {
	print "You have quit \n";
	$app->quit();
}
```

Useful methods for a FileChooserDialog

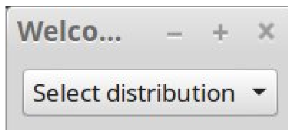
Note that the *action* of the FileChooserDialog can be one of the following: *"open"* (the file chooser will only let the user pick an existing file), *"save"* (the file chooser will let the user pick an existing file, or type in a new filename), *"select_folder"* (the file chooser will let the user pick an existing folder), *"create_folder"* (the file chooser will let the user name an existing or new folder).

Besides the methods used in the Code used to generate this example, we have:

- *set_show_hidden(TRUE)* is used to display hidden files and folders.
- *set_select_multiple(TRUE)* sets that multiple files can be selected. This is only relevant if the mode/action is *"open"* or *"select_folder"*.
- In a 'Save as' dialog, *set_current_name(current_name)* sets current_name in the file selector, as if entered by the user; current_name can be something like Untitled.txt. This method should not be used except in a "Save as" dialog.
- The default current folder is "recent items". To set another folder use *set_current_folder_uri(uri)*; but note you should use this method and cause the file chooser to show a specific folder only when you are doing a "Save as" command and you already have a file saved somewhere.

13. TreeViews and ComboBoxes

13. 1. ComboBox (one column)



This ComboBox prints to the terminal your selection when you change it.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;

my @distros = ('Select distribution', 'Fedora', 'Mint', 'Suse');

my $window=Gtk3::Window->new('toplevel');
$window->set_title('Welcome to GNOME');
$window->set_default_size(200,-1);
$window->set_border_width(10);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit});

# the data in the model, of type string
my $liststore = Gtk3::ListStore->new('Glib::String');

# append the data in the model
foreach my $data (@distros) {

	# You add data in two steps
	# 1) Add a new empty row to the Treestore; to this row
	# generate a reference or a pointer ($iter)
	# 2) In order to fill this row with content, you apply the Gtk3::Treestore
	# set-methode to this
	# !!! The list of pairs must contain as many elements as
	#  the number of columns in the  Tree- or ListStore!!!
	# example: $liststore ->set ($iter, 0 => 'content of the row in column 1',
	# 1 => ('content of the row in column 2 etc.)

	my $iter = $liststore->append();
	$liststore->set($iter, 0 => "$data");

	}

# a combobox to see the data stored in the model
my $combobox = Gtk3::ComboBox->new_with_model($liststore);

# a cellrenderer to render the text
```

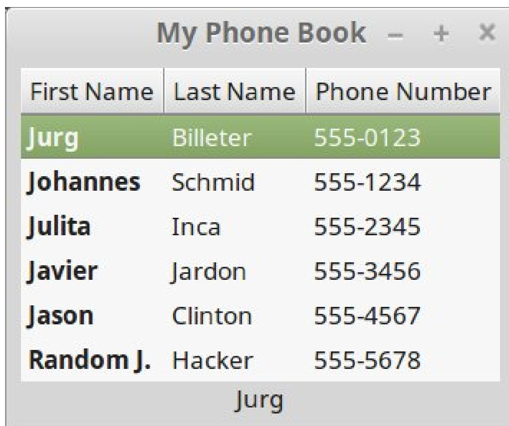
```
my $cell = Gtk3::CellRendererText->new();

# pack the cell into the beginning of the combobox, allocating
# no more space than needed
$combobox->pack_start($cell, FALSE);
# associate/verknüpfe a property ('text') of the cellrenderer (cell)
# to a column (column 0) in the model used by the combobox
$combobox->add_attribute($cell, 'text', 0);

# the first row is the active one by default at the beginning
$combobox->set_active(0);

# connect the signal emitted when a row is selected to the callback
# function
$combobox->signal_connect('changed'=>\&on_changed);

# add the combobox to the window
$window->add($combobox);

# show the window and run the Application
$window->show_all;
Gtk3->main();

sub on_changed {
	# if the row selected is not the first one, write its value on the
	# terminal
	my ($combo) = @_;
	my $active = $combo->get_active();
	if ($active != 0) {
		print "You choose $distros[$active]. \n";
		}
	return TRUE;
	}
```

Useful methods for a ComboBox widget

The ComboBox widget is designed around a Model/View/Controller design: the Model stores the data; the View gets change notifications and displays the content of the model; the Controller, finally, changes the state of the model and notifies the view of these changes. For more information and for a list of useful methods for ComboBox see The Model/View/Controller design.

In line 54 the *'changed'* signal is connected to the callback function *on_changed()* using *$widget->signal_connect("signal" => \&callback function)*. See Signale und Callbacks for a more detailed explanation.

13. 2. Simple TreeView with ListStore



This TreeView displays a simple ListStore with the selection 'changed' signal connected.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;

my @columns = ('First Name','Last Name','Phone Number');
my @phonebook =(['Jurg', 'Billeter', '555-0123'],
                ['Johannes', 'Schmid', '555-1234'],
                ['Julita', 'Inca', '555-2345'],
                ['Javier', 'Jardon', '555-3456'],
                ['Jason', 'Clinton', '555-4567'],
                ['Random J.', 'Hacker', '555-5678']);

my $window=Gtk3::Window->new('toplevel');
$window->set_title('My Phone Book');
$window->set_default_size(200,100);
$window->set_border_width(10);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit});

# the data in the model
# (three strings for each row, one for each column)
my $listmodel = Gtk3::ListStore->new('Glib::String','Glib::String','Glib::String');

# append the values in the model
for (my $i=0; $i <= $#phonebook; $i++) {
        # You add data in two steps
        # 1) Add a new empty row to the Treestore; to this row
        # generate a reference or a pointer ($iter)
        # 2) In order to fill this row with content, you apply the Gtk3::Treestore
        # set-methode to this
        # !!! The list of pairs must contain as many elements as
        #  the number of columns in the  Tree- or ListStore!!!
        # example: $liststore ->set ($iter, 0 => 'content of the row in column 1',
        # 1 => ('content of the row in column 2 etc.)
```

```
        my $iter = $listmodel->append();
        $listmodel->set($iter, 0 => "$phonebook[$i][0]",
                               1 => "$phonebook[$i][1]",
                               2 => "$phonebook[$i][2]");
        }

# a treeview to see the data stored in the model
my $view = Gtk3::TreeView->new($listmodel);

# create a cellrenderer to render the text for each of the 3 columns
for (my $i=0; $i <= $#columns; $i++) {
        my $cell = Gtk3::CellRendererText->new();

        # the text in the first column should be in boldface
        if ($i == 0) {
                $cell->set_property('weight_set',TRUE);
                # !!! Pango.Weight.BOLD is not recognized in perl I don't know
                # why. I took instead 800 for bold (default value (ie normal) is 400)
                $cell->set_property('weight',800);
                }

        # the column is created
        # Usage:
        # Gtk3::TreeViewColumn->new_with_attributes (title, cell_renderer,
        # attr1 => col1, ...)
        my $col = Gtk3::TreeViewColumn->new_with_attributes($columns[$i],$cell,'text' => $i);

        # alternatively you can do the same as following(see the first example):
        # first: create a TreeViewColumn
        #my $col = Gtk3::TreeViewColumn->new();
        # second: pack the renderer into the beginning of the column,
        # allocating no more space than needed
        #$col->pack_start($cell, FALSE);
        # third: set the title
        #$col -> set_title($columns[$i]);
        # fourth: add the attributes
        #       USAGE: add_attribute(cell_renderer, attribute=>column);
        #$col->add_attribute($cell, text=>$i);

        # and it is appended to the treeview
        $view->append_column($col)
        }

# create a TreeSelection Objekt
my $treeselection=$view->get_selection();
# when a row is selected, it emits a signal
$treeselection->signal_connect('changed' => \&on_changed);

# the label we use to show the selection
my $label = Gtk3::Label->new();
$label->set_text('');
```

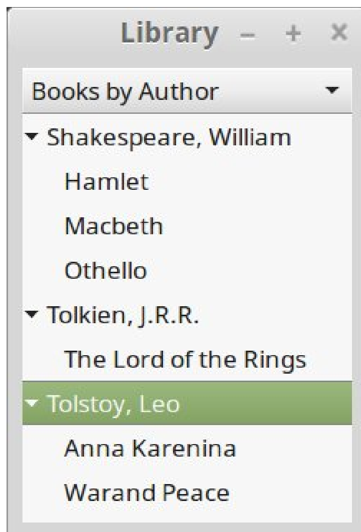
```
# a grid to attach the widgets
my $grid = Gtk3::Grid->new();
$grid->attach($view, 0,0,1,1);
$grid->attach($label, 0,1,1,1);

# attach the grid to the window
$window->add($grid);

# show the window and run the Application
$window->show_all;
Gtk3->main();

sub on_changed {
	my ($sel) = @_;

	# get the model and the iterator that points at the data in the model
	my ($model, $iter) = $sel->get_selected();

	# set the label to a new value depending on the selection, if there is
      # one
	if ($iter != '') {
		# we want the data at the model's column 0
		# where the iter is pointing
		my $value = $model->get_value($iter,0);
		# set the label to a new value depending on the selection
		$label->set_text("$value");
		}
	else {
		$label->set_text('');
		}
	return TRUE;
	}
```

Useful methods for a TreeView widget

The TreeView widget is designed around a Model/View/Controller design: the Model stores the data; the View gets change notifications and displays the content of the model; the Controller, finally, changes the state of the model and notifies the view of these changes. For more information, and for a list of useful methods for TreeModel, see The Model/View/Controller design.

In line 83 the *'changed'* signal is connected to the callback function *on_changed()* using *$widget->signal_connect("signal" => \&callback function)*. See Signale und Callbacks for a more detailed explanation.

13. 3. Simpler TreeView with TreeStore



This TreeView displays a TreeStore.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;

my @books =(['Tolstoy, Leo', 'Warand Peace', 'Anna Karenina'],
            ['Shakespeare, William', 'Hamlet', 'Macbeth', 'Othello'],
            ['Tolkien, J.R.R.', 'The Lord of the Rings']);

my $window=Gtk3::Window->new('toplevel');
$window->set_title('Library');
$window->set_default_size(250,100);
$window->set_border_width(10);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit});

# the data are stored in the moel
# create a treestore with one column
my $store = Gtk3::TreeStore->new('Glib::String');

# append the values in the model
for (my $i=0; $i <= $#books; $i++) {
	# You add data in two steps
	# 1) Add a new empty row to the Treestore; to this row
	# generate a reference or a pointer ($iter)
	# 2) In order to fill this row with content, you apply the Gtk3::Treestore
	# set-methode to this
	# !!! The list of pairs must contain as many elements as
	#  the number of columns in the  Tree- or ListStore!!!
	# example: $liststore ->set ($iter, 0 => 'content of the row in column 1',
	# 1 => ('content of the row in column 2 etc.)
```

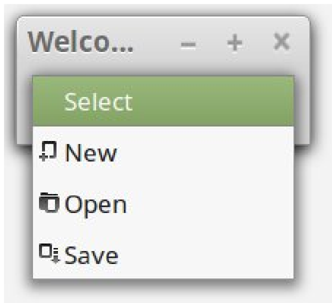
```
	# First of all you generate the parent Iter and fill in the parent cells.
	# these are always at in the first place (i.e. $books[$i][0]).
	# only one column is needed
	my $iter = $store->append();
	$store->set($iter, 0 => "$books[$i][0]");

	for (my $j=1; $j <= $#{$books[$i]}; $j++) {
		# in dieser Spalter fügen wir Kind Iters zu den Eltern Iters hinzu
		# und fügen diesen Kind Iters Daten hinzu / erneut nur 1 Spalte
		my $iter_child = $store->append($iter);
		$store->set($iter_child, 0 => "$books[$i][$j]");
		}
	}

# the treeview shows the model
# create a treeview on the model $store
my $view = Gtk3::TreeView->new();
$view->set_model($store);

# the cellrenderer for the column - text
my $renderer_books = Gtk3::CellRendererText->new();
# the column is created
my $column_books = Gtk3::TreeViewColumn->new_with_attributes('Books by Author',
$renderer_books, 'text'=>0);
# and it is appended to the treeview
$view->append_column($column_books);

# the books are sortable by authors
$column_books->set_sort_column_id(0);

# add the treeview to the window
$window->add($view);

# show the window and run the Application
$window->show_all;
Gtk3->main();
```

Useful methods for a TreeView widget

The TreeView widget is designed around a Model/View/Controller design: the Model stores the data; the View gets change notifications and displays the content of the model; the Controller, finally, changes the state of the model and notifies the view of these changes. For more information and for a list of useful methods for TreeModel see The Model/View/Controller design.

13. 4. The Model/View/Controller design (to do)

13. 5. ComboBox (two columns)



This ComboBox prints to the terminal your selection when you change it.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;

my @actions = (        ['Select', ''],
                       ['New', 'document-new'], # same as 'gtk-new'
                       ['Open', 'document-open'], # same as 'gtk-open'
                       ['Save', 'document-save']); # same as 'gtk-save'

my $window=Gtk3::Window->new('toplevel');
$window->set_title('Welcome to GNOME');
$window->set_default_size(200,-1);
$window->set_border_width(10);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit});

# the data in the model, of type string on two columns
my $listmodel = Gtk3::ListStore->new('Glib::String','Glib::String');

# append the data
for (my $i; $i<=$#actions; $i++) {

        # You add data in two steps
        # 1) Add a new empty row to the Treestore; to this row
        # generate a reference or a pointer ($iter)
        # 2) In order to fill this row with content, you apply the Gtk3::Treestore
        # set-methode to this
        # !!! The list of pairs must contain as many elements as
        #  the number of columns in the  Tree- or ListStore!!!
        # example: $liststore ->set ($iter, 0 => 'content of the row in column 1',
        # 1 => ('content of the row in column 2 etc.)

        my $iter = $listmodel->append();
        $listmodel->set($iter, 0 => "$actions[$i][0]", 1 => "$actions[$i][1]");

        }
```

```
# a combobox to see the data stored in the model
my $combobox = Gtk3::ComboBox->new_with_model($listmodel);

# cellrenderers to render the data
my $renderer_pixbuf = Gtk3::CellRendererPixbuf->new();
my $renderer_text = Gtk3::CellRendererText->new();

# we pack the cell into the beginning of the combobox, allocating
# no more space than needed;
# first the image, then the text;
# note that it does not matter in which order they are in the model,
# the visualization is decided by the order of the cellrenderers
$combobox->pack_start($renderer_pixbuf, FALSE);
$combobox->pack_start($renderer_text, FALSE);

# associate a property of the cellrenderer to a column in the model
# used by the combobox
$combobox->add_attribute($renderer_text, 'text' => 0);
$combobox->add_attribute($renderer_pixbuf, 'icon-name' => 1);

# the first row is the active one at the beginning
$combobox->set_active(0);

# connect the signal emitted when a row is selected to the callback
# function
$combobox->signal_connect('changed', \&on_changed);

# add the combobox to the window
$window->add($combobox);

# show the window and run the Application
$window->show_all;
Gtk3->main();

sub on_changed {
	# if the row selected is not the first one, write its value on the
	# terminal
	my ($combo) = @_;
	my $active = $combo->get_active();
	if ($active != 0) {
		print "You choose $actions[$active][0]. \n";
		}
	return TRUE;
	}
```

Useful methods for a ComboBox widget

The ComboBox widget is designed around a Model/View/Controller design: the Model stores the data; the View gets change notifications and displays the content of the model; the Controller, finally, changes the state of the model and notifies the view of these changes. For more information and for a list of useful methods for ComboBox see The Model/View/Controller design.

In line the 'changed' signal is connected to the callback function on_changed() using widget.connect(signal, callback function). See Signale und Callbacks for a more detailed explanation.

13. 6. More Complex Treeview with ListStore

This TreeView displays a simple ListStore with the selection 'changed' signal connected.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;

my @list_of_dvds = ('The Usual Suspects','Gilda','The Godfather', 'Pulp Fiction', 'Once Upon a
Time in the West', 'Rear Window');

my $window=Gtk3::Window->new('toplevel');
$window->set_title('My DVDs');
$window->set_default_size(250,100);
$window->set_border_width(10);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit});

# the data are stored in the model
# create a liststore with one column
my $listmodel = Gtk3::ListStore->new('Glib::String');

# save the number of lines in a seperate variable
my $row_count = 0;

# append the values in the model
for (my $i=0; $i <= $#list_of_dvds; $i++) {
	# You add data in two steps
	# 1) Add a new empty row to the Treestore; to this row
	# generate a reference or a pointer ($iter)
```

```
        # 2) In order to fill this row with content, you apply the Gtk3::Treestore
        # set-methode to this
        # !!! The list of pairs must contain as many elements as
        #  the number of columns in the  Tree- or ListStore!!!
        # example: $liststore ->set ($iter, 0 => 'content of the row in column 1',
        # 1 => ('content of the row in column 2 etc.)

        my $iter = $listmodel->append();
        $listmodel->set($iter, 0 => "$list_of_dvds[$i]");

        # Wenn eine Zeile hinzugefügt wurde, erhöhe die Variable $row_count
        $row_count++;
        }

# a treeview to see the data stored in the model
my $view = Gtk3::TreeView->new($listmodel);

# cellrenderer for the first column
my $cell = Gtk3::CellRendererText->new();
# the first column is created
my $col = Gtk3::TreeViewColumn->new_with_attributes('Title',$cell,'text' => 0);
# and it is appended to the treeview
$view->append_column($col);

# create a TreeSelection Objekt
my $selection=$view->get_selection();
# when a row is selected, it emits a signal
$selection->signal_connect('changed' => \&on_changed);

# the label we use to show the selection
my $label = Gtk3::Label->new();
$label->set_text('');

# a button to ad new titles, connected to a callback function
my $button_add = Gtk3::Button->new('Add');
$button_add->signal_connect('clicked' => \&add_cb);

# an entry to enter titles
my $entry = Gtk3::Entry->new();

# a button to remove titles, connected to a callback function
my $button_remove = Gtk3::Button->new('Remove');
$button_remove->signal_connect('clicked'=>\&remove_cb);

# a button to remove all titles, connected to a callback function
my $button_remove_all = Gtk3::Button->new('Remove All');
$button_remove_all->signal_connect('clicked' => \&remove_all_cb);

# a grid to attach the widgets
my $grid = Gtk3::Grid->new();
$grid->attach($view, 0, 0, 4, 1);
$grid->attach($label, 0, 1, 4, 1);
```

```
$grid->attach($button_add, 0, 2, 1, 1);
$grid->attach_next_to($entry, $button_add, 'right', 1, 1);
$grid->attach_next_to($button_remove, $entry, 'right', 1, 1);
$grid->attach_next_to($button_remove_all, $button_remove, 'right', 1, 1);

# ad the grid to the window
$window->add($grid);

# show the window and run the Application
$window->show_all;
Gtk3->main();

sub on_changed {
	my ($sel) = @_;

	# get the model and the iterator that points at the data in the model
	my ($model, $iter) = $sel->get_selected();

	# set the label to a new value depending on the selection, if there is
       # one
	if ($iter != '') {
		# we want the data at the model's column 0
		# where the iter is pointing
		my $value = $model->get_value($iter,0);
		# set the label to a new value depending on the selection
		$label->set_text("$value");
		}
	else {
		$label->set_text('');
		}
	return TRUE;
	}

sub add_cb {
	# append to the model the title that is in the entry
	my $title = $entry->get_text();
	my $add_iter = $listmodel->append();
	$listmodel->set($add_iter, 0 => "$title");
	# and print a message in the terminal
	print "$title has been added \n";

	# Wenn eine Zeile hinzugefügt wurde, erhöhe die Variable $row_count
	$row_count++;
	}

sub remove_cb {
	# check if there is still an entry in the model
	# the methode iter_n_children($iter) returns the number of children
	# that iter has or here with $iter=NULL the number of toplevel nodes
	# which are in a liststore (no childs!!) the number of lines
	my $len = $listmodel->iter_n_children();
	#if ($len != 0) {
```

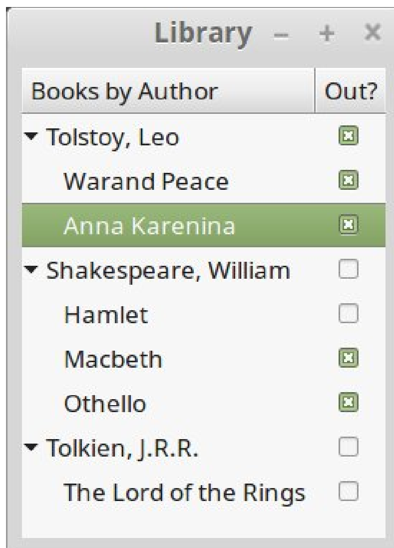
```
	# another way is the methode $listmodel->get_iter_first which returns
	# FALSE if the tree is empty
	#if ($listmodel->get_iter_first()) {

	# last but not least you can save the lenght of rows in an own
	# variable (here: $rows (see below)
	if ($row_count != 0) {
		# get the selection
		my ($model, $iter) = $selection->get_selected();
		# if there is a selection, print a message in the terminal
		# and remove it from the model (TO DO)

		# we want the data at the model's column 0
		# where the iter is pointing
		my $value = $model->get_value($iter,0);
		print "$value has been removed \n";
		$listmodel->remove($iter);

		# wenn eine Zeile gelöscht wurde, erniedrige die Variable $row_count
		$row_count--;
		}
	else {
		print "Empty list \n";
		}
	}
sub remove_all_cb {
	# check if there is still an entry in the model
	if ($row_count != 0) {
		# remove all the entries in the model
		for (my $i=1; $i <= $row_count; $i++) {
			my $iter = $listmodel->get_iter_first;
			$listmodel->remove($iter);
			}

		# Setze die Anzahl der Zeilen auf 0
		$row_count = 0;
		}
	else {
		print "Empty list \n";
		}
	}
```

Useful methods for a TreeView widget

The TreeView widget is designed around a Model/View/Controller design: the Model stores the data; the View gets change notifications and displays the content of the model; the Controller, finally, changes the state of the model and notifies the view of these changes. For more information and for a list of useful methods for TreeModel see The Model/View/Controller design.

In line XX the 'changed' signal is connected to the callback function on_changed() using widget.connect(signal, callback function). See Signale und Callbacks for a more detailed explanation.

13. 7. More Complex TreeView with TreeStore



This TreeView displays a TreeStore with two columns, one of which is rendered as a toggle.

Code used to generate this example

```
#!/usr/bin/perl

use strict;
use Glib ('TRUE','FALSE');
use Gtk3 -init;

my @books = (['Tolstoy, Leo', ['Warand Peace', TRUE], ['Anna Karenina', FALSE]],
             ['Shakespeare, William', ['Hamlet', FALSE], ['Macbeth', TRUE], ['Othello', FALSE]],
             ['Tolkien, J.R.R.', ['The Lord of the Rings', FALSE]],);

my $window=Gtk3::Window->new('toplevel');
$window->set_title('Library');
$window->set_default_size(250,100);
$window->set_border_width(10);
$window->signal_connect('delete_event' => sub {Gtk3->main_quit});

# the data are stored in the moel
# create a treestore with two column
my $store = Gtk3::TreeStore->new('Glib::String', 'Glib::Boolean');

# fill in the model
for (my $i=0; $i <= $#books; $i++) {
	# You add data in two steps
	# 1) Add a new empty row to the Treestore; to this row
	# generate a reference or a pointer ($iter)
	# 2) In order to fill this row with content, you apply the Gtk3::Treestore
	# set-methode to this
	# !!! The list of pairs must contain as many elements as
	#  the number of columns in the  Tree- or ListStore!!!
	# example: $liststore ->set ($iter, 0 => 'content of the row in column 1',
	# 1 => ('content of the row in column 2 etc.)
```

```
	# First of all you generate the parent Iter and fill in the parent cells.
	# these are always at in the first place (i.e. $books[$i][0]).
	# only one column is needed
	my $iter = $store->append();
	$store->set($iter, 0 => "$books[$i][0]", 1 => FALSE);

	for (my $j=1; $j <= $#{$books[$i]}; $j++) {
		# in dieser Spalter fügen wir Kind Iters zu den Eltern Iters hinzu
		# und fügen diesen Kind Iters Daten hinzu / erneut nur 1 Spalte
		my $iter_child = $store->append($iter);
		$store->set($iter_child, 0 => "$books[$i][$j][0]", 1 =>"$books[$i][$j][1]");
		}
	}

# the treeview shows the model
# create a treeview on the model $store
my $view = Gtk3::TreeView->new();
$view->set_model($store);

# the cellrenderer for the first column - text
my $renderer_books = Gtk3::CellRendererText->new();
# the first column is created
my $column_books = Gtk3::TreeViewColumn->new_with_attributes('Books by Author',
$renderer_books, 'text'=>0);
# and it is appended to the treeview
$view->append_column($column_books);

# the books are sortable by authors
$column_books->set_sort_column_id(0);

# the cellrenderer for the second column - boolean renderer as a toggle
my $renderer_in_out = Gtk3::CellRendererToggle->new();
# the second column is created
my $column_in_out = Gtk3::TreeViewColumn->new_with_attributes('Out?', $renderer_in_out,
'active'=>1);
# and it is appended to the treeview
$view->append_column($column_in_out);
# connect the cellrenderertoggle with a callback function
$renderer_in_out->signal_connect('toggled' => \&on_toggled);

# add the treeview to the window
$window->add($view);

# show the window and run the Application
$window->show_all;
Gtk3->main();

# callback function for the signa emitted by the cellrenderertoggle
sub on_toggled {
	my ($widget, $path_string) = @_;
```

```
	# Get the boolean value (1=TRUE, 0=FALSE) of the selected row
	# first generate a Gtk3::TreePath, by using $path_string as a argument
	# to the new_from_string method of Gtk3::TreePath.
	# This will give us a 'geographical' indication which row was edited.
	my $path = Gtk3::TreePath->new_from_string($path_string);
	# the get the Gt3::Treeiter of the TreePath $path, which will refer
	# to a row
	my $iter = $store->get_iter($path);
	# last get the value with the function get_value on the model
	my $current_value = $store->get_value($iter,1);

	# change the value of the toggled item
	# instead of the if/elsif construction you can simple write this line
	# "$current_value ^= 1;" [but I don't understand why this works :-)]
	if ($current_value == 0) {$current_value = 1;}
	elsif ($current_value==1){$current_value = 0; }
	$store->set( $iter, 1, $current_value);

	# check if length if the path is 1
	# (that is, if we are selecting an author (= parent cell)
	if (length($path_string) == 1) {

		# get the number of the childrens that the parent $iter has
		my $n = $store->iter_n_children($iter);

		# get the iter associated with its first child
		my $citer = $store->iter_children($iter);

		foreach (my $i = 0; $i <= $n-1; $i++) {
			$store->set($citer,1 => $current_value);
			$store->iter_next($citer);
			}
		}

	# if the length of the path is not 1
	# (that is if we are selecting a book)
	else {
		# get the parent and the first child of the parent
		# (that is the first book of the author)
		my $piter = $store->iter_parent($iter);
		my $citer = $store->iter_children($piter);

		# get the number of the childrens that the parent $iter has
		my $n = $store->iter_n_children($piter);

		# Erzeuge eine Variable, mit der mittels einer Schleife
		#überprüft wird, ob alle Kinder items selected sind
		my $all_selected;

		# check if all children are selected
		foreach (my $i = 0; $i <= $n-1; $i++) {
			my $value = $store->get_value($citer,1);
```

```
				if ($value == 1) {
					$all_selected = 1;
					}
				if ($value == 0) {
					$all_selected = 0;
					last;
					}
				$store->iter_next($citer);
				}

			# wenn all_selected = 1 (=TRUE) soll auch das Eltern item
			# ausgewählt werden
			if ($all_selected == 1) {
				$store->set($piter, 1, 1);
				}
			# wenn ich alle Kind Elemente selektiert sind
			# soll auch das Eltern Element nicht selektiert sein
			elsif ($all_selected == 0) {
				$store->set($piter, 1, 0);
				}
			}


		}
```

Useful methods for a TreeView widget

The TreeView widget is designed around a Model/View/Controller design: the Model stores the data; the View gets change notifications and displays the content of the model; the Controller, finally, changes the state of the model and notifies the view of these changes. For more information and for a list of useful methods for TreeModel see The Model/View/Controller design.

In line XX the 'toggled' signal is connected to the callback function on_toggled() using widget.connect(signal, callback function). See Signale und Callbacks for a more detailed explanation.

Questions?

If you found any programming mistakes, misspellings or other mistakes, please let me know (Maximilian-Lika@gmx.de)

If you have further questions regarding using the perl-Gtk3 module perhaps the GTK-Perl list (gtk-perl-list@gnome.org) is a better place to ask.