

# Tutorial for beginners (version 2016/06/02)

In folgendem Tutorial werden Sie die Grundlagen der GUI Programmierung mit Perl/Gtk3 lernen.

Wenn Sie noch nie vorher programmiert haben oder nicht mit den Konzepten der objektorientierten Programmierung vertraut sind, sollten Sie vielleicht zunächst ein paar Basics diesbezüglich lernen. Das Buch "Perl in 21 Tagen", "Einführung in Perl" oder ein gutes Perl Tutorial sind vielleicht dann ein besserer Ort zu starten für Sie. Wenn Sie diese Grundlagen beherrschen, können Sie gerne hierher zurückkommen und dieses Tutorial austesten.

## Code Beispiele starten

Zum Starten der Code Beispiele in diesem Tutorial gehen Sie folgendermaßen vor:

1. Schreiben Sie oder fügen Sie den Code in eine Datei (ein), und speichern Sie diese Datei mit einem Namen wie filename.pl
2. Zum Ausführen des Codes, schreiben Sie in einem Terminal den folgenden Befehl:

```
perl ./filename.pl
```

Nach Ausführung des Codes, werden Sie entweder die Steuerelemente (Widgets) auf Ihrem Bildschirm sehen oder (wenn Sie etwas von dem Code falsch eingetippt haben), werden Sie eine Fehlermeldung sehen, welche Ihnen bei der Identifizierung des Problems helfen wird.

## Ein Rundlauf durch die einzelnen Widgets

Das Tutorial wird Sie durch zunehmend komplexe Beispiele und Theorie führen, aber Sie können sich auch frei fühlen, direkt zu dem Teil des Tutorials zu gehen, welches für Sie am meisten hilfreich ist.

### 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

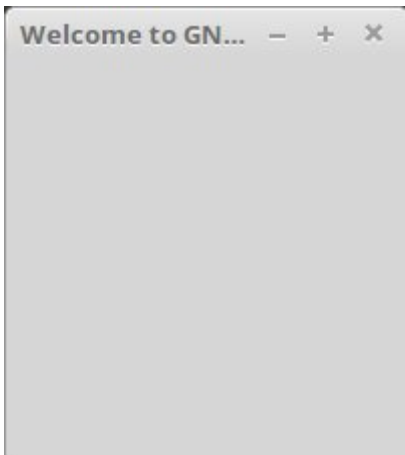
13.7. More Complex TreeView with TreeStore

## **14. Custom widget**

14.1. Widget (**to do**)

# 1. Ein einfaches Fenster

## 1. 1. Fenster



Diese Gtk3 Anwendung zeigt ein einfaches Fenster mit Titel an.

### *Code used to generate this example*

```
#!/usr/bin/perl

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

# Erstelle ein Gtk3 Fenster
my $window = Gtk3::Window->new('toplevel');

# Setze den Titel
$window->set_title('Welcome to GNOME')

# Beende das Programm, wenn der Benutzer die Anwendung schließt
$window->signal_connect('delete_event'=>sub{Gtk3->main_quit()});

# Zeige das Fenster
$window->show_all;

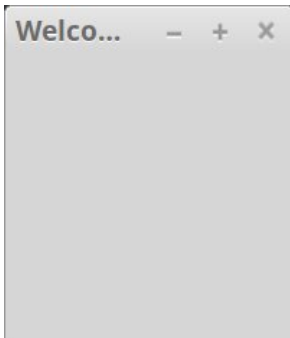
# Erstelle und führe die Anwendung aus
Gtk3->main();
```

### *Useful methods for a Window widget*

- `set_default_size(200, 100)` legt die voreingestellte Größe des Fensters auf eine Weite von 200 und eine Höhe von 100 fest; Wenn wir anstatt einer positiven Zahl -1 übergeben, erhalten wir die Standard Größe
- `set_position("center")` zentriert das Fenster. Andere Optionen sind `"none"`, `"mouse"`,

"center-always", "center-on-parent".

## 1. 2. Application Window<sup>1</sup>



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 shouldn'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 the `GApplication` 'run' and 'quit' methods.

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

1) We can make a binding 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
```

---

<sup>1</sup> 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!

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 decision 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 separate 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. Bilder und Beschriftungen

### 2. 1. Bild



Diese Gtk Anwendung zeigt ein Bild aus dem aktuellen Verzeichnis an.



Wenn das Bild nicht erfolgreich geladen wird, wird ein "kaputtes Bild"-Icon angezeigt. "Filename.png" muss im aktuellen Verzeichnis sein, damit dieser Code funktioniert.

#### *Code used to generate this example*

```
#!/usr/bin/perl

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

# Erstelle ein Fenster und lege den Titel und die Größe fest
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()});

# Erstelle ein Bild
my $image = Gtk3::Image->new();

# Lege den Inhalt des Bildes als die Datei filename.png fest
$image->set_from_file('gnome-image.png');

# Füge das Bild dem Fenster hinzu
```

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

# Zeige das Fenster und starte die Anwendung
$window->show_all;
Gtk3->main();
```

### *Useful methods for an Image widget*

- Zum Laden eines Bildes über das Netzwerk, benutzen Sie *set\_from\_pixbuf(\$pixbuf)*, wobei *pixbuf* ein *Gdk::Pixbuf* ist:

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

# Erstelle ein Fenster
[... siehe oben!]

# Erstelle ein Pixbuf von der Datei="gnome-image.png", mit der Weite=32
# und der Höhe=64 und dem booleschen Wer preserve_aspect_ratio=FALSE.
my $pixbuf = Gtk3::Gdk::Pixbuf->new_from_file_at_scale('gnome-image.png', 64, 128,
FALSE);

# Erstelle das Bild
my $image = Gtk3::Image->new();

# Lege den Inhalt des Bildes als das eben erstellte Objekt $pixbuf fest.
$image->set_from_pixbuf($pixbuf);

# Füge das Bild dem Fenster hinzu
$window->add($image);

[... see oben!]
```

Wenn *preserve\_aspect\_ratio*=*TRUE* ist, können wir *new\_from\_file\_at\_size(filename, width, height)* nutzen. Wenn die *Weite* oder *Höhe* -1 ist, wird es nicht zusammengepresst.

Zum Laden aus einem Input Stream, sehen Sie *new\_from\_stream()* and *new\_from\_stream\_at\_scale()* in der Dokumentation.

## **2. 2. Strings [check the english!!!]**

### **[python specific – to do!]**

GTK+ benutzt UTF-8 kodierte Zeichenketten für alle Texte. Wenn Umlaute (ä, ö, ü, ß etc.) falsch dargestellt werden, müssen Sie das Pragma "use utf8;" am Anfang des Skriptes setzen. Wenn Sie auch etwas im Terminal anzeigen wollen (bspw. Mit dem "print"-Befehl), müssen Sie die "Line discipline" der Standardausgabe in den UTF-8 Modus setzen. Dies verhindert, dass das Terminal versucht, die Zeichenketten wieder nach Latin-1 umzuwandeln. Sehen Sie ein Beispiel für dieses Problem in dem Kapitel über die Toolbar.

```
# Setze das Pragma utf8 damit Umlaute im Gtk GUI korrekt angezeigt werden
use utf8;
# Damit das Terminal nicht versucht, die Strings nach Latin-1 umzuwandeln
# mit der Folge, dass die Umlaute im Terminal falsch angezeigt werden
# muss die 'Line Discipline' der Standardausgabe in den UTF 8 Modus gesetzt werden
binmode STDOUT, ':utf8';
```

## 2. 4. Beschriftung



Eine einfache Beschriftung.

### *Code used to generate this example*

```
#!/usr/bin/perl

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

# Erstelle ein Fenster
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()});

# Erstelle ein Label
my $label = Gtk3::Label->new();

# Lege den Text des Labels fest
$label -> set_text('Hello GNOME!');

# Füge die Beschriftung dem Fenster hinzu
$window->add($label);

# Zeige das Fenster und starte die Anwendung
$window->show_all;
Gtk3->main();
```

### *Useful methods for a Label widget*



Eine Erklärung, wie man mit Zeichenketten in Perl/Gtk3 umgeht, finden Sie im Kapitel "Strings" (siehe oben)

- `set_line_wrap(TRUE)` bricht die Zeilen um, wenn der Text des Labels die Größe des Widgets übersteigt
- `set_justify("left")` (oder `"right"`, `"center"`, `"fill"`) legt die Ausrichtung der Zeilen im Text des Labels relative zueinander fest. Diese Methode hat keine Auswirkung bei einer einzeiligen Beschriftung.
- Für ausgezeichneten Text können wir `set_markup('text')` benutzen, wobei `"text"` einen Text in der Pango Auszeichnungssprache ist. Ein Beispiel:<sup>2</sup>

```
$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>");
```

- Um Zeilen umzubrechen, wenn der Text die Größe des Widgets übersteigt, müssen Sie `set_line_wrap(TRUE)` schreiben

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

### 3. Properties

#### Overview

*Eigenschaften* beschreiben die Konfiguration und den Status des Elements. Jedes Element hat seinen eigenen speziellen Satz von Eigenschaften. Zum Beispiel hat ein Steuerelement wie ein Button die Eigenschaft `"label"`, welche den Text des Elements enthält. Man kann den Namen und den Wert einer beliebigen Eigenschaft ändern mit der Methode, die mit dieser Eigenschaft verknüpft ist. Um zum Beispiel ein Label mit dem Text "Hello World", einem Winkel von 25 Grad, und der Ausrichtung rechtsbündig zu erzeugen, können Sie folgendes verwenden:

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

Wenn Sie einmal ein solches Label erstellt haben, können Sie den Text des Labels mit `$label->get_label()` erhalten, und analog auch den Inhalt für die anderen Eigenschaften.

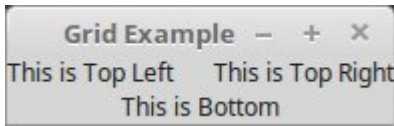
Anstatt diese *"getters"* und *"setters"* zu verwenden, können Sie die Eigenschaften jeweils auch mit `get_property('propname')` and `set_property('prop-name', value)` erhalten und setzen.

---

<sup>2</sup> Beachte: An Zeilenumbrüchen innerhalb der Anführungszeichen werden auch im Label die Zeilen umgebrochen. Um dies zu verändern und dennoch auch im Code die Lesbarkeit zu erhöhen werden hier die Anführungszeichen jeweils geschlossen und mit dem Verbindungsoperator `"."` verbunden

## 4. Grid, separator and scrolling

### 4. 1. Grid



Einige Beschriftungen (*labels*) in einem Raster (*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});

# drei Labels/Beschriftungen
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");

# Ein Raster/Grid
my $grid = Gtk3::Grid->new();

# Etwas Platz zwischen den Spalten des Rasters
$grid->set_column_spacing(20);

# Im Raster:
# Bringe das erste Label in der oberen linken Ecke an
$grid->attach($label_top_left,0,0,1,1);

# Bringe das zweite Label an
$grid->attach($label_top_right,1,0,1,1);

# Hefte das dritte Label unter dem ersten Label an
$grid->attach_next_to($label_bottom, $label_top_left, 'bottom', 2, 1);

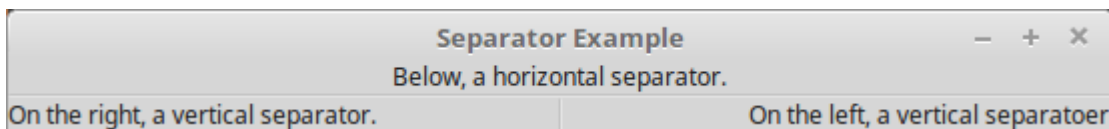
# Füge das Raster dem Fenster hinzu
$window->add($grid);

# Zeige das Fenster und starte die Anwendung
$window->show_all;
Gtk3->main();
```

## Useful methods for a Grid

- Um ein *Kind-Element* in der Position *links*, *oben* in einem Slot mit der übergebenen *Weite*, *Höhe* anzubringen benutzen Sie `attach(Kind, links, oben, Breite, Höhe)`. Wenn ein *Geschwister-Element* bereits vorhanden ist, können wir auch `attach_next_to(Kind, Geschwister, Seite, Breite, Höhe)` benutzen, wobei *Seite* "left", "right", "top" oder "bottom" sein kann.
- `insert_row(position)` und `insert_column(position)` machen genau was sie sagen; Kinder, welche an oder unter dieser Position angebracht sind werden eine Reihe nach unten verschoben, und Kinder, welche diese Position überspannen, wachsen um auch die neue Reihe zu überspannen; `insert_next_to(Geschwister, Seite)` fügt eine Reihe oder Spalte an der angegebenen Position. Die neue Zeile oder Spalte wird neben dem *Geschwister* und an der *Seite*, die durch die Eigenschaft *Seite* festgelegt ist, platziert. Wenn *Seite* "top" oder "bottom" ist, wird eine Reihe eingefügt, wenn *Seite* "left" oder "right" ist, wird eine Spalte eingefügt.
- `set_row_homogeneous(TRUE)` und `set_column_homogeneous(TRUE)` stellen sicher, dass jede Reihe oder Spalte jeweils dieselbe Breite oder Höhe haben.
- `set_row_spacing(spacing)` und `set_column_spacing(spacing)` erzwingen Abstand zwischen den Reihen oder Spalten. Der Wert des Abstands kann zwischen 0 (was der Standard Wert ist) und 32767 liegen.

## 4. 2. Separator



Ein horizontaler und ein vertikaler Trenner unterteilt einige Beschriftungen.

### 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()});

# Drei Beschriftungen
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');

# Ein horizontaler Trenner
my $hseparator = Gtk3::Separator->new('horizontal');

# Ein verticaler Trenner
my $vseparator = Gtk3::Separator->new('vertical');

# Ein Raster, um die Beschriftungen und Trenner anzuheften
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);

# Füge das Raster dem Fenster hinzu
$window -> add($grid);

# Zeige das Fenster und starte die Anwendung
$window -> show_all();
Gtk3->main();

```

## 4. 3 ScrolledWindow



Ein Bild in einem scrollbaren Fenster.

### *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()});

# Das scrollbare Fenster
my $scrolled_window=Gtk3::ScrolledWindow->new();
$scrolled_window->set_border_width(10);

# Die Scrollbar soll immer angezeigt werden (andere Optionen: automatisch ("automatic")
# oder nie ("never")
$scrolled_window->set_policy('always', 'always');

# Ein Bild, das etwas größer ist als das Fenster
my $image = Gtk3::Image->new();
$image->set_from_file('gnome-image.png');

# Füge das Bild dem scrollbaren Fenster hinzu
$scrolled_window->add_with_viewport($image);

# Füge das scrollbare Fenster dem Hauptfenster hinzu
$window->add($scrolled_window);

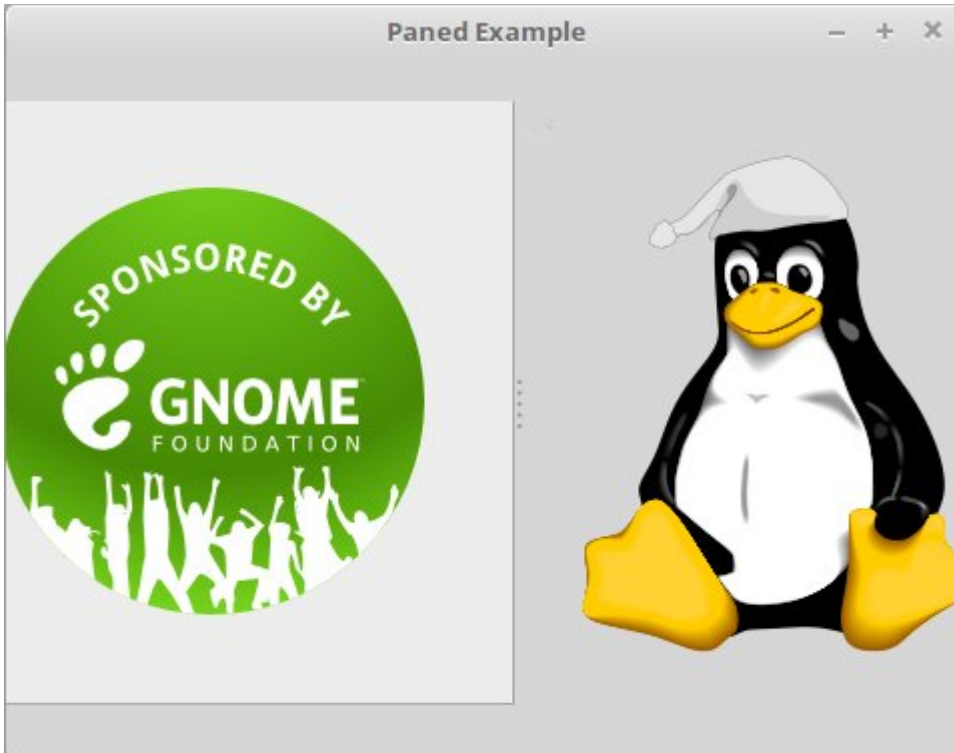
# Zeige das Fenster und starte die Anwendung
$window->show_all();
Gtk3->main;

```

### ***Useful methods for a ScrolledWindow widget***

- *set\_policy(hscrollbar\_policy, vscrollbar\_policy)*, bei dem mögliche Optionen *"automatic"*, *"always"* oder *"never"* sind, bestimmen, ob die horizontale oder vertikale Scrollbar erscheinen soll: Mit der Option *"automatic"* erscheint sie nur, wenn es notwendig ist, die Optionen *"always"* (immer) und *"never"* (nie) sind selbsterklärend.
- *add\_with\_viewport(\$widget)* wird verwendet, um ein *Gtk3::Widget* *\$widget* ohne native Scrollmöglichkeiten in das scrollbare Fenster einzufügen.
- *set\_placement(window\_placement)* legt die Platzierung des Inhalts in Bezug auf die Scrollbalken für das scrollbare Fenster fest. Die möglichen Optionen sind *"top-left"* (Standardoption: Die Scrollbalken sind unten und rechts von dem Fenster), *"top-right"*, *"bottom-left"*, *"bottom-right"*.
- *set\_hadjustment(\$adjustment)* und *set\_vadjustment(\$adjustment)* legen das *Gtk3::Adjustment* *\$adjustment* fest. Dieses repräsentiert den Anfangswert, die untere und obere Grenze, zusammen mit der Einzelschritt- und Seiten-Erhöhung und der Seitengröße, und wird erstellt mit dem Befehl *"my \$adjustment=Gtk3::Adjustment->new(initial value, minimum value, maximum value, step\_increment, page\_increment, page\_size)"*, wobei die einzelnen Werte Fließzahlen sind. (Beachte, dass *step\_increment* in diesem Fall nicht verwendet wird und daher auf 0 gesetzt werden kann.

## 4. 4 Paned



Zwei Bilder in zwei anpassbaren Bereichen, horizontal ausgerichtet.

### *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()});

# Ein neues Element mit zwei anpassbaren Bereichen
# einer links und einer rechts
my $paned = Gtk3::Paned->new('horizontal');

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

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

# Füge das erste Bild dem linken Bereich hinzu
$paned -> add1($image1);
```

```
# Füge das zweite Bild dem rechtem Bereich hinzu
$paned -> add2($image2);

# Füge die Bereiche dem Fenster hinzu
$window -> add($paned);

# Zeige das Fenster und starte die Anwendung
$window -> show_all();
Gtk3->main();
```

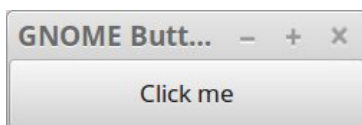
### *Useful methods for a Paned widget*

Für zwei vertikal ausgerichtete Bereiche, benutze "*vertical*" anstatt von "*horizontal*". Die Methode *add1(\$widget1)* fügt das *\$widget1* dem oberen Bereich hinzu, und *add2(\$widget2)* fügt das *\$widget2* dem unteren Bereich hinzu.

## 5. Signals, callbacks and buttons

### 5. 1. Signals and callbacks

### 5. 2. Button



Eine Schaltfläche, die mit einer einfachen Funktion verknüpft ist.

### *Code used to generate this example*

```
#!/usr/bin/perl

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

# Ein Fenster
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()});

# Ein Button
my $button = Gtk3::Button->new();
# Mit einer Beschriftung
$button->set_label('Click me');

# Verbinde das Signal "clicked", welches von dem Button ausgesendet wird,
# mit der Funktion do_clicked
$button->signal_connect('clicked' => \&do_clicked);
```

```
# Füge den Button dem Fenster hinzu
$window -> add ($button);

# Zeige das Fenster und starte die Funktion
$window -> show_all();
Gtk3->main();

# Die Funktion, die mit dem "clicked"-Signal des Buttons verknüpft ist
sub do_clicked {
    print("You clicked me! \n");
}
```

### Useful methods for a Button widget

In Zeile 20 wird das "clicked" Signal des Buttons mit der Funktion `do_clicked()` verbunden, mit dem Befehl `$widget->signal_connect(signal, callback function)`. Siehe Signale und Callbacks für eine detailliertere Erklärung.

- `set_relief("none")` setzt den Relief-Stil der Ecken des `Gtk3::Buttons` als "ohne Stil" fest – im Gegensatz zu "normal"
- Wenn die Beschriftung der Schaltfläche ein Standardsymbol ist (*stock icon*), legt `set_use_stock(TRUE)` die Bezeichnung als den Namen des korrespondierenden Standardsymbols fest.
- Folgendermaßen wird ein Bild (z.B. ein stock image) für den Button festgelegt:

```
my $image = Gtk3::Image->new();
# USAGE: set_from_stock(stock-id, Gtk3::IconSize3);
$image -> set_from_stock("gtk-about", "4");
$button-> set_image($image);
```

- Wenn wir `set_focus_on_click(FALSE)` benutzen, wird der Button nicht fokussiert, wenn er mit der Maus angeklickt wird. Dies könnte hilfreich sein in Fällen von Werkzeugleisten, damit der Tastaturfokus nicht von dem Hauptbereich der Anwendung entfernt wird.

## 5. 3. LinkButton



Eine Schaltfläche, die einen Link zu einer Webseite enthält.

### Code used to generate this example

```
#!/usr/bin/perl
```

- 
- 3 Unfortunately the nicknames 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' |                        |                        |                 |

```

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

# Ein Fenster
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()});

# Ein Button
my $button = Gtk3::LinkButton->new('http://live.gnome.org');
# mit einem Label
$button->set_label('Link to GNOME live');

# Füge den Button dem Fenster hinzu
$window -> add ($button);

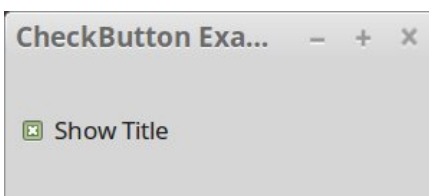
# Zeige das Fenster und starte die Anwendung
$window -> show_all();
Gtk3->main();

```

## Useful methods for a LinkButton widget

- *get\_visited()* gibt den *"besucht"* Status der URI, auf die der LinkButton verweist, zurück (*TRUE* für besucht und *FALSE* für nicht besucht). Der Button erhält den *"besucht"*-Status, wenn er angeklickt wurde.
- *set\_visited(TRUE)* legt den *"besucht"* Status der URI, auf die der LinkButton verweist, als *TRUE* für wahr (bzw. analog *FALSE* für nicht besucht) fest
- Jedes Mal, wenn der Button gedrückt wird, wird das Signal *'activate-link'* ausgesendet. Für eine detaillierte Erklärung der Signale und callback Funktionen, siehe das Kapitel Signals and callbacks

## 5. 4. CheckButton



Dieser CheckButton schaltet den Titel an und aus.

### Code used to generate this example

```

#!/usr/bin/perl

use strict;

```

```

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

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

# Eine Schaltfläche
my $button = Gtk3::CheckBox->new();
# mit einer Beschriftung
$button->set_label('Show Title');

# Folgende Zeile verbindet das Signal "toggled", dass von dem CheckBox ausgesendet wird
# mit der Callback Funktion toggled_cb
$button->signal_connect('toggled' => \&toggled_cb);

# standardmäßig soll der CheckBox ausgewählt sein
$button->set_active(TRUE);

# Füge den Button dem Fenster hinzu
$window -> add($button);

# Zeige das Fenster und starte die Anwendung
$window -> show_all();
Gtk3->main();

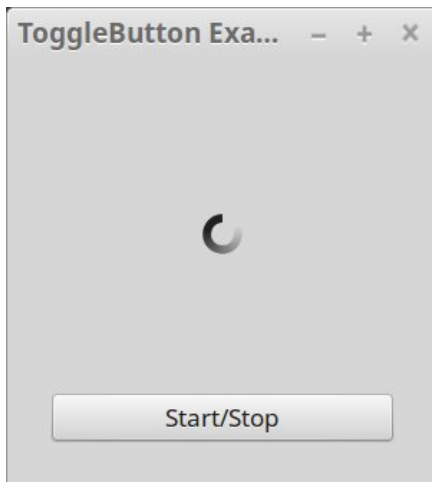
sub toggled_cb {
    # Setze den Titel des Fenster ("CheckBox Example"),
    # wenn der CheckBox ausgewählt ist
    if ($button->get_active()) {
        $window->set_title ('CheckBox Example');
    }
    else {
        $window->set_title ('');
    }
}

```

### ***Useful methods for a CheckBox widget***

In Zeile 18 wird das *"toggled"* Signal mit der Callback Funktion *toggled\_cb()* mit der Methode *\$widget->signal\_connect("signal", \&callback\_function)* verbunden. Für eine detailliertere Erklärung siehe das Kapitel Signals and callbacks.

## 5. 5. ToggleButton



Wenn dieser ToggleButton im aktivierten Status ist, dreht sich der Spinner/Kreis.

### *Code used to generate this example*

```
#!/usr/bin/perl

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

# Ein Fenster
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()});

# Eine Spinner-Animation (d.h. ein Kreis, der sich dreht)
my $spinner = Gtk3::Spinner->new();
# Die Animation soll sich horizontal und vertikal ausdehnen
$spinner->set_hexpand(TRUE);
$spinner->set_vexpand(TRUE);

# Eine ToggleSchaltfläche
my $button = Gtk3::ToggleButton->new_with_label('Start/Stop');
# Mit folgender Zeile wird das Signal "toggled", das vom ToggleButton ausgesendet wird, wenn
# sich der Status des Buttons ändert, mit der Callback Funktion toggled_cb verbunden
$button->signal_connect('toggled' => \&toggled_cb);

# Ein Raster, um die Elemente anzuordnen
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);
```

```

# Füge das Raster dem Fenster hinzu
$window->add($grid);

# Zeige das Fenster und starte die Anwendung
$window -> show_all();
Gtk3->main();

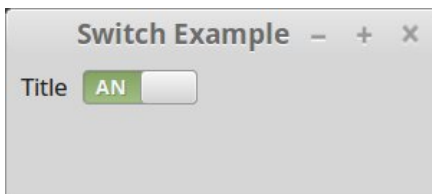
# Die Callback Funktion für das 'toggled' Signal
sub toggled_cb {
    # Wenn der ToggleButton in aktiviertem Zustand ist, starte die Animation
    if ($button->get_active()) {
        $spinner->start();
    }
    # im andern Fall, stoppe sie
    else {
        $spinner->stop();
    }
}

```

### *Useful methods for a ToggleButton widget*

In Zeile 25 wird das *"toggled"* Signal mit der Callback Funktion *toggled\_cb()* mit der Methode *\$widget->signal\_connect("signal", \&callback\_function)* verbunden. Für eine detailliertere Erklärung siehe das Kapitel Signals and callbacks.

## 5. 6. Switch



Der On-/Off-Schalter lässt den Titel erscheinen und verschwinden.

### *Code used to generate this example*

```

#!/usr/bin/perl

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

# Ein Fenster
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()});

```



```

# Ein Schalter
my $switch = Gtk3::Switch->new();
# Der Schalter soll standardmäßig an sein
$switch->set_active(TRUE);
# Verbinde das Signal "notify::active", welches von dem Schalter ausgesendet wird,
# mit der Callback Funktion activate_cb
$switch->signal_connect('notify::active' => \&activate_cb);

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

# Ein Raster, um die Elemente anzuordnen
my $grid = Gtk3::Grid->new();
$grid->set_column_spacing(10);
$grid->attach($label, 0, 0, 1, 1);
$grid->attach($switch, 1, 0, 1, 1);

# Füge das Raster dem Fenster hinzu
$window->add($grid);

# Zeige das Fenster und starte die Anwendung
$window->show_all();
Gtk3->main();

# Callback Funktion. Da das Signal notify::active ist
# benötigen wir das 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 Zeile 20 wird das *"notify::active"* Signal mit der Callback Funktion *activate\_cb()* mit der Methode *\$widget->signal\_connect("signal", \&callback\_function)* verbunden. Für eine detailliertere Erklärung siehe das Kapitel Signals and callbacks.

## 5. 7. RadioButton



Drei RadioButtons. Man kann im Terminal sehen, ob sie an- oder ausgeschaltet sind.

### *Code used to generate this example*

```
#!/usr/bin/perl

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

# Ein Fenster
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()});

# Ein neuer RadioButton
my $button1 = Gtk3::RadioButton->new('Button1');
# mit der Beschriftung "Button 1"
$button1->set_label("Button 1");
# Verbinde das Signal "toggled", das von dem RadioButton ausgesendet wird,
# mit der Callback Funktion toggled_cb
$button1->signal_connect('toggled' => \&toggled_cb);

# Ein weiterer RadioButton, in der selben Gruppe wie $button1
my $button2 = Gtk3::RadioButton->new_from_widget($button1);
# mit der Beschriftung "Button 2"
$button2->set_label('Button 2');
# Verbinde das Signal "toggled", das von dem RadioButton ausgesendet wird,
# mit der Callback Funktion toggled_cb
$button2->signal_connect('toggled' => \&toggled_cb);
# Standardmäßig soll $button2 nicht ausgewählt sein
$button2->set_active(FALSE);

# Ein weiterer RadioButton, in derselben Gruppe wie $button1
my $button3 = Gtk3::RadioButton->new_from_widget($button1);
# mit der Beschriftung "Button 3"
$button3->set_label('Button 3');
# Verbinde das Signal "toggled", das von dem RadioButton ausgesendet wird,
# mit der Callback Funktion toggled_cb
$button3->signal_connect('toggled' => \&toggled_cb);
# Standardmäßig soll $button2 nicht ausgewählt sein
$button3->set_active(FALSE);
```

```

# Ein Raster, um die Buttons anzuordnen
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);
# Füge das Raster dem Fenster hinzu
$window->add($grid);

# Zeige das Fenster und starte die Anwendung
$window->show_all();
Gtk3->main();

# Die Callback Funktion
sub toggled_cb {
    # Beachte: Der erste Wert, der von der Funktion übergebenen wurde, beinhaltet
    # immer eine Referenz auf das Widget, bei dem das Signal auftrat!
    my ($button) = @_;
    # Eine weitere Variable, um den Zustand des Buttons zu beschreiben
    my $state = 'unknown';
    # Wann immer der RadioButton ausgewählt wird, wird $state auf "on" gestellt
    if ($button->get_active()) {
        $state = 'on';
    }
    # Ansonsten ist $state "off"
    else {
        $state = 'off';
    }
    # Wann immer die Funktion aufgerufen wird (also ein RadioButton an- oder
    # abgewählt wird), schreibe im Terminal, welcher RadioButton an- oder ausgeschaltet
    # wurde
    my $button_label = $button->get_label();
    print ("$button_label was turned $state \n");
}

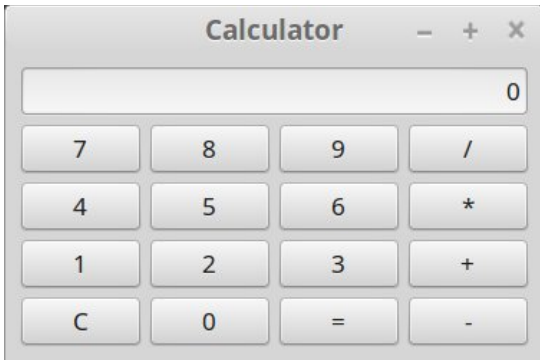
```

### ***Useful methods for a RadioButton widget***

In Zeile 20 wird das *"toggled"* Signal mit der Callback Funktion *toggled\_cb()* mit der Methode *\$widget->signal\_connect("signal", \&callback\_function)* verbunden. Für eine detailliertere Erklärung siehe das Kapitel Signals and callbacks.

Ein anderer Weg, um einen neuen RadioButton mit einer Beschriftung zu erzeugen, ist *"my \$button1 = Gtk3::RadioButton->new\_with\_label("", "Button1");"*. (Das erste Argument beschreibt die Gruppe der RadioButtons, welche wir mit der Funktion *get\_group()* erhalten können, das zweite ist das Label)

## 6. ButtonBox



Ein Rechner – Die Buttons sind in einer horizontalen ButtonBox eingeschlossen.

### Code used to generate this example<sup>4</sup>

```
#!/usr/bin/perl

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

# Ein Fenster
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()});

# Ein Eingabefeld
my $entry = Gtk3::Entry->new();
# mit einem voreingestellten Text
$entry->set_text('0');
# Der Text soll rechts ausgerichtet sein
$entry->set_alignment(1);
# Der Text im Eingabefeld soll nicht durch das Hineinschreiben modifiziert werden können
$entry->set_can_focus(FALSE);

# Ein Raster
my $grid = Gtk3::Grid->new();
$grid->set_row_spacing(5);

# um das Eingabefeld einzufügen
$grid->attach($entry, 0,0,1,1);

# Die Beschriftungen der Knöpfe werden in einem Array gespeichert
my @buttons = (7, 8, 9, '/',
               4, 5, 6, '*',
               1, 2, 3, '+',
               'C', 0, '=', '-');
```

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

```

# Jede der 4 Reihen ist eine ButtonBox, die in das Raster eingefügt wird
foreach my $i (0..3) {
    my $hbox = Gtk3::ButtonBox->new('horizontal');
    $hbox->set_spacing(5);
    $grid->attach($hbox, 0, $i+1, 1, 1);
    # Jede ButtonBox, d.h. jede Zeile, hat 4 Knöpfe, die mit der Callback Funktion verknüpft
    # wird
    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);
    }
}

# Einige Variablen für die Berechnungen
my $first_number = 0;
my $second_number = 0;
my $counter = 0;
my $operation = "";

# Füge das Raster dem Fenster hinzu
$window->add($grid);

# Zeige das Fenster und starte die Anwendung
$window->show_all();
Gtk3->main();

# Die Callback Funktion für alle 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();

    # für die Rechengvorgänge
    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');
    $operation = 'minus';
    }
elseif ($label eq '*') {
    $counter += 1;
    if ($counter > 1) {
        do_operation();
    }
    $entry->set_text('0');
    $operation = 'multiplication';
    }
elseif ($label eq '/') {
    $counter += 1;
    if ($counter > 1) {
        do_operation();
    }
    $entry->set_text('0');
    $operation = 'division';
    }
# für den '=' Button
elseif ($label eq '=') {
    do_operation();
    $entry->set_text("$first_number");
    $counter = 1;
    }
# für den "C" (=Cancel) Button
elseif ($label eq 'C') {
    $first_number = 0;
    $second_number = 0;
    $counter = 0;
    $entry->set_text('0');
    $operation = "";
    }
# für Zahl-Buttons
else {
    my $new_digit = $button->get_label();
    # ZERO DIVISION 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 += $second_number;
}
elseif ($operation eq 'minus') {
    $first_number -= $second_number;
}
elseif ($operation eq 'multiplication') {
    $first_number *= $second_number;
}
elseif ($operation eq 'division') {
    $first_number = $first_number / $second_number;
    # ZERO DIVISI ERROR -> TO DO!!!
}
}

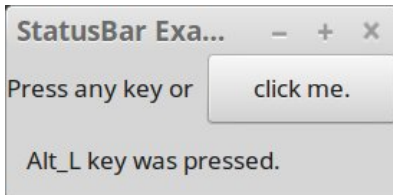
```

## Useful methods for a ButtonBox widget

- Das Layout der ButtonBox wird mit dem Befehl *set\_layout(layout)* festgelegt, wobei *layout* "spread" (die Buttons werden gleichmäßig über die Box verteilt), "edge" (die Buttons werden an den Ecken der Box platziert), "start" (die Buttons werden zum Start hin gruppiert), "end" (die Buttons werden zum Ende der Box hin gruppiert), "expand".
- *set\_child\_secondary(\$button, TRUE)* bestimmt, ob der Button in einer sekundären Gruppe von Kindern erscheinen soll. Ein typischer Gebrauch eines sekundären Kind-Elements ist der Hilfe-Button in einem Dialog. Diese sekundäre Gruppe erscheint nach den anderen Kindern, wenn das Layout "start", "spread" oder "edge" ist, bzw. vor den anderen Kindern, wenn das Layout "end" ist. Wenn der Stil "start" oder "end" ist, werden die sekundären Kind-Elemente von den primären Kind-Elementen gesehen auf der anderen Seite der Button Box platziert. Bei den anderen Stilen, erscheinen sie unmittelbar neben den Haupt-Kind-Elementen
- *set\_child\_non\_homogeneous(\$button, TRUE)* bestimmt, ob das Kind-Element von der vereinheitlichenden Größeneinteilung ausgenommen ist. Standard ist FALSE.
- *set\_spacing(spacing)* legt den Abstand zwischen den Buttons in der Box in pixeln fest.

## 7. Other display widgets

### 7. 1. Statusbar



Die StatusBar berichtet, wenn man den Button klickt oder eine Taste drückt (und welche Taste!)

#### *Code used to generate this example*

```
#!/usr/bin/perl

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

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

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

# Eine Schaltfläche
my $button = Gtk3::Button->new('click me. ');
# die mit einer Callback Funktion verbunden ist
$button->signal_connect('clicked', \&button_clicked_cb);

# Die StatusBar
my $statusbar = Gtk3::Statusbar->new();
# Die StatusBar enthält eine $context_id, die zwar nicht im UI erscheint, aber für die eindeutige
# Identifizierung der Quelle der Nachricht notwendig ist
my $context_id = $statusbar->get_context_id('example');
# Wir setzen eine Nachricht auf den Stapelspeicher der StatusBar
$statusbar->push($context_id, 'Waiting for you to do something...');

# Ein Raster, um die Elemente anzuordnen
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);
```




```

# Füge das Raster dem Fenster hinzu
$window->add($grid);

# Zeige das Fenster und starte die Anwendung
$window -> show_all();
Gtk3->main();

# Callback Funktion, wenn der Button angeklickt wird:
# Wenn man den Button anklickt, soll das Ereignis der StatusBar gemeldet werden,
# auf welcher wir einen neuen Status ausgeben
sub button_clicked_cb {
    $statusbar->push($context_id, 'You clicked the button.');
```

 Gtk3::Gdk::keyval\_name(\$keyval)<sup>5</sup> wandelt den Tastenwert von der Funktion \$event->keyval in den symbolischen Namen um. Die Namen und korrespondierenden Tastenwerte können [hier](#) gefunden werden, aber bspw. Wird GDK\_KEY\_BackSpace zu "BackSpace".

### Useful methods for a StatusBar widget

In Zeile 21 wird das "clicked" Signal mit der Callback Funktion clicked\_cb() mit der Methode \$widget->signal\_connect("signal", \&callback\_function) verbunden. Für eine detailliertere Erklärung siehe das Kapitel Signals and callbacks.

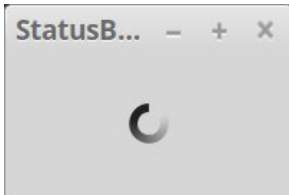
- `pop($context_id)` entfernt die erste Nachricht im Stapelspeicher der StatusBar mit der übergebenen `context_id`.
- `remove_all($context_id)` entfernt alle Nachrichten im Stapelspeicher der StatusBar mit der übergebenen `context_id`.
- `remove($context_id, $message_id)` entfernt die Nachricht mit der angegebenen `message_id`

---

5 The more perlsh Schreibweise "`Gtk3::Gdk->keyval_name`" seems not to work correctly

in dem Stapelspeicher der Statusbar mit der jeweiligen *context\_id*. Die *message\_id* wird zurückgegeben, wenn man mit `my message_id = push($context_id, "the message")` die Nachricht auf den Stapelspeicher der StatusBar schiebt.

## 7. 2 Spinner



Der Kreis wird durch das Drücken der Leertaste gestoppt und gestartet.

### *Code used to generate this example*

```
#!/usr/bin/perl

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

# ein Fenster
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()});

# eine Spinner-Animation
my $spinner = Gtk3::Spinner->new();
# die sich standardmäßig dreht
$spinner -> start();
# Füge die Spinner-Animation dem Fenster hinzu
$window->add($spinner);

# Zeige das Fenster und starte die Anwendung
$window -> show_all();
Gtk3->main();

# Event Handler
# Ein Signal von der Tastatur (die Leertaste) steuert, ob die Spinner-Animation stoppt oder startet
sub do_key_press_event {
    # $keyname ist der symbolische Name des Tastenwertes, der von dem Ereignis übergeben
    # wird
    my ($widget, $event) = @_;
    my $keyval = $event->keyval;
    # !!! WICHTIG: Die perlische Schreibweise Gtk3::Gdk->keyval_name($event);
    # scheint buggy zu sein !!!
    my $keyname = Gtk3::Gdk::keyval_name($keyval);

    # Prüfe, ob die Leertaste gedrückt wurde! Wenn dies der Fall ist...
```

```

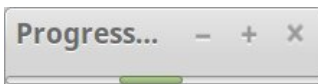
    if ($keyname == 'space') {
        # und die Spinner-Animation aktiviert ist...
        if ($spinner->get_property('active')) {
            # stoppe die Animation
            $spinner->stop();
        }
        # Wenn die Spinner-Animation aber nicht aktiviert ist
        else {
            # starte sie erneut
            $spinner->start();
        }
    }
    # Beende die Signalausendung
    return TRUE;
}

```



Gtk3::Gdk::keyval\_name(\$keyval)<sup>6</sup> wandelt den Tastenwert von der Funktion \$event->keyval in den symbolischen Namen um. Die Namen und korrespondierenden Tastenwerte können [hier](#) gefunden werden, aber bspw. Wird GDK\_KEY\_BackSpace zu "BackSpace".

### 7. 3. ProgressBar



Dieser Fortschrittsbalken wird durch das Drücken irgendeiner Taste auf der Tastatur gestoppt und gestartet.

#### *Code used to generate this example*

```

#!/usr/bin/perl

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

# ein Fenster
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()});

# ein Fortschrittsbalken
my $progressbar = Gtk3::ProgressBar->new();
# Füge den Fortschrittsbalken dem Fenster hinzu
$window->add($progressbar);

# Die Methode pulse wird alle 100 Millisekunden aufgerufen

```

<sup>6</sup> The more perlsh Schreibweise "*Gtk3::Gdk->keyval\_name*" seems not to work correctly

```

# und $source_id wird als ID der Ereignisquelle festgelegt,
# (d.h. der Fortschrittsbalken ändert seine Position alle 100 Millisekunden!)
my $source_id = Glib::Timeout->add(100, \&pulse);

# Zeige das Fenster und starte die Anwendung
$window -> show_all();
Gtk3->main();

# Event Handler
# irgendein Signal von der Tastatur steuert, ob der Ladebalken startet/stoppt
sub do_key_press_event {
    # Wenn der Ladebalken gestoppt ist (also $source_id == 0 ist – siehe den else-Block unten)
    # schalte ihn ein
    if ($source_id == 0) {
        $source_id = Glib::Timeout->add(100, \&pulse);
    }
    # Wenn der Ladebalken sich bewegt, entferne die Ereignisquelle mit der ID $source_id
    # von dem Hauptkontext (d.h. stoppe den Ladebalken) und lege $source_id als 0 fest
    else {
        Glib::Source->remove($source_id);
        $source_id = 0;
    }
    # Beende die Signalausendung
    return TRUE;
}

# Funktion der Glib Ereignisquelle
# Der Ladebalken ist im "aktivierten Zustand", wenn diese Methode aufgerufen wird!
sub pulse {
    $progressbar->pulse();
    # Rufe die Funktion erneut auf
    return TRUE;
}

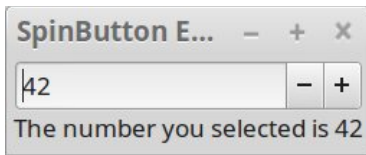
```

### ***Useful methods for a ProgressBar widget***

- Anstatt *pulse()*, bei welchem sich der Ladebalken nach vor und zurück bewegt, können wir *set\_fraction(fraction)* verwenden, wenn wir wollen, dass der Ladebalken sich sukzessive einen Bruchteil (= *fraction*; als Option ist eine Fließkommazahl zwischen 0.0 (entspricht 0 %) inklusive 1.0 (entspricht 100 %) möglich) füllt.
- Um einen Text festzulegen und ihn zu zeigen (über den Ladebalken gelagert) benutze *set\_text("text")* und *set\_show\_text(TRUE)*. Wenn der Text nicht gesetzt ist, aber dennoch *set\_show\_text(TRUE)* ist der Text die Prozentzahl der bereits geleisteten Arbeit.

## 8. Entry Widgets

### 8. 1. SpinButton



Wähle eine Nummer durch Eintragen in das Eingabefeld oder durch Klicken auf die +/- Buttons!

#### *Code used to generate this example*

```
#!/usr/bin/perl

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

# ein Fenster
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()});

# ein Gtk3::Adjustment (d.h. ein Einstellungselement) mit folgenden Parametern:
# Anfangswert, Minimalwert, Maximalwert,
# Einzelschritterhöhung (= Erhöhung, wenn die Pfeiltasten oder die +/- Buttons gedrückt werden),
# Seitenerhöhung (hier nicht verwendet),
# Seitengröße (hier nicht verwendet)
my $ad = Gtk3::Adjustment->new(0, 0, 100, 1, 0, 0);

# ein Spin-Button für Ganzzahlen (Nachkommastellen=0)
# Optionen: ($adjustment, climb_rate, digits)
my $spin = Gtk3::SpinButton->new($ad, 1, 0);
# so weit wie möglich
$spin->set_hexand(TRUE);

# wir verbinden das Signal "value-changed" das von dem Spin-Button ausgesendet wird
# mit der Callback Funktion spin_selected
$spin->signal_connect('value-changed' => \&spin_selected);

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

# Ein Raster, um die Elemente anzuordnen
my $grid = Gtk3::Grid->new();
$grid->attach($spin, 0, 0, 1, 1);
$grid->attach($label, 0, 1, 2, 1);
```

```
# Füge das Raster dem Fenster hinzu
$window->add($grid);

# Zeige das Fenster und starte die Anwendung
$window->show_all();
Gtk3->main();

# Callback Funktion: Das Signal des Spin-Buttons wird dazu verwendet, um den Text des Labels
# zu verändern
sub spin_selected {
    my $number = $spin->get_value_as_int();
    $label->set_text("The number you selected is $number");
}
```

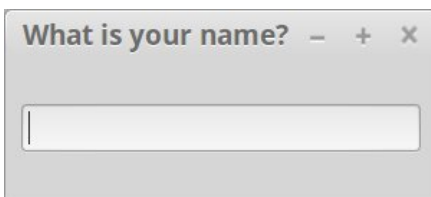
### Useful methods for a SpinButton widget

Um einen Spin-Button zu konstruieren wird ein `Gtk3::Adjustment` benötigt. Dieses repräsentiert den Anfangswert, die untere und obere Grenze, zusammen mit der Einzelschritt- und Seiten-Erhöhung und der Seitengröße, und wird erstellt mit dem Befehl `"my $adjustment=Gtk3::Adjustment->new(initial value, minimum value, maximum value, step_increment, page_increment, page_size)"`, wobei die einzelnen Werte Fließzahlen sind. *step\_increment* ist die Erhöhung/Verringerung, die durch das Drücken der Pfeiltasten oder der Schaltflächen des SpinButtons bewirkt wird. (Beachte, dass *page\_increment* und *page\_size* in diesem Fall nicht verwendet werden und daher auf 0 gesetzt werden sollten.

In Zeile 27 wird das `"value-changed"` Signal mit der Callback Funktion `spin_selected()` mit der Methode `$widget->signal_connect("signal", \&callback_function)` verbunden. Für eine detailliertere Erklärung siehe das Kapitel Signals and callbacks.

- Wenn der Wert des Spin-Buttons (auf den Minimal- bzw. Maximalwert) springen soll, wenn er das Maximum bzw. Minimum übersteigt, muss `set_wrap(TRUE)` eingestellt werden. Wenn dies der Fall ist, wird das `"wrapped"` Signal ausgesendet.
- `set_digits(digits)` legt die Genauigkeit der Nachkommastellen, die von dem SpinButton angezeigt werden soll (bis zu 20 Nachkommastellen) fest.
- Um den Wert des SpinButtons als Ganzzahl zu erhalten, nutze `get_value_as_int()`.

## 8. 2. Entry



Diese Anwendung begrüßt Sie/Dich im Terminal mit dem eingegebenen Namen.

## Code used to generate this example

```
#!/usr/bin/perl

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

# ein Fenster
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()});

# Ein einzeliges Eingabefeld
my $name_box = Gtk3::Entry->new();
# Das Eingabefeld sendet ein Signal, wenn die Eingabetaste gedrückt wird
# Dieses wird mit der Callback Funktion cb_activate verknüpft
$name_box->signal_connect('activate', \&cb_activate);

# Füge das Eingabefeld dem Fenster hinzu
$window->add($name_box);

# Zeige das Fenster und starte die Anwendung
$window->show_all();
Gtk3->main();

# Der Inhalt des Eingabefeldes wird im Terminal mit einer Begrüßung ausgegeben
sub cb_activate {
    # Erhalte den Inhalt des Eingabeelements
    my $entry = $_[0];
    my $name = $entry->get_text();
    # Gebe den Inhalt in einer schönen Form im Terminal aus
    #(!ohne "\n" kommt keine Ausgabe auf dem Terminal, bis das Programm beendet wird!)
    print "Hello $name! \n";
}
```

## Useful methods for Entry widget

In Zeile 12 wird das "activate" Signal mit der Callback Funktion `cb_activate()` mit der Methode `$widget->signal_connect("signal", \&callback_function)` verbunden. Für eine detailliertere Erklärung siehe das Kapitel Signals and callbacks. Einige der Signale, die ein `Gtk3::Entry` Eingabefeld aussenden kann sind 'activate' (wird ausgesendet, wenn der Benutzer die Eingabetaste betätigt); 'backspace' (wird ausgelöst, wenn der Benutzer die Backspace (Rücktaste) oder Shift-Backspace Tasten betätigt); 'copy-clipboard' (Str-c und Str-Insert); 'paste-clipboard' (Str-v und Shift-Insert); 'delete-from-cursor' (Delete-Taste für das Löschen eines Zeichens bzw. Str-Delete, für das Löschen eines Wortes); 'icon-press' (ausgesendet, wenn der Benutzer ein aktivierbares Symbol anklickt); 'icon-release' (ausgesendet bei der Freigabe des Buttons bei einem Klick über ein aktivierbares Symbol); 'insert-at-cursor' (ausgesendet, wenn der Benutzer die Einfügung einer

festgelegten Zeichenkette am Cursor beginnt); *'move-cursor'* (ausgesendet, wenn der Benutzer eine Cursor-Bewegung beginnt); *'populate-popup'* (ausgesendet, bevor das Kontext-Menü des Eingabefeldes gezeigt wird; kann benutzt werden, um Elemente zu dem Eingabefeld hinzuzufügen).

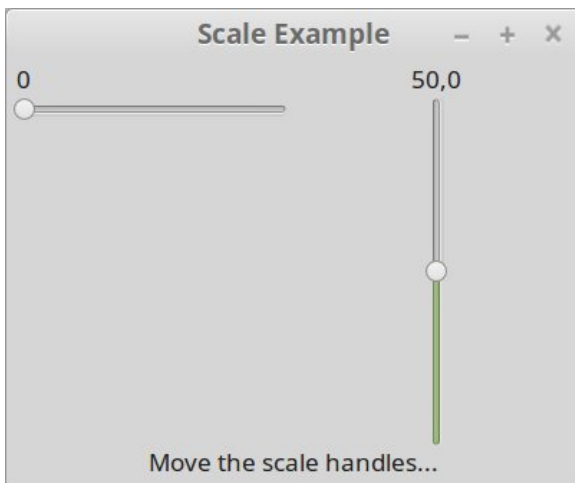
- *get\_buffer()* und *set\_buffer(\$buffer)*, wobei *\$buffer* ein *Gtk3::EntryBuffer* Objekt ist, können benutzt werden, um einen Buffer (=Zwischenspeicher) für das Eingabefeld festzulegen bzw. zu erhalten.
- *get\_text()* und *set\_text('some text')* können dazu verwendet werden, um den Inhalt für das Eingabefeld festzusetzen bzw. zu erhalten..
- *get\_text\_length()* ist selbsterklärend.
- *get\_text\_area()* liefert den Bereich, in dem der Text des Eingabefeldes gezeichnet ist.
- Mit *set\_visibility(FALSE)* werden die Buchstaben im Eingabefeld als unsichtbare Zeichen dargestellt. Dazu werden die besten vorhandenen unsichtbaren Zeichen in der aktuellen Schriftart verwendet. Die zu verwendenden unsichtbaren Zeichen können verändert werden mit *set\_invisible\_char(ch)*, wobei *ch* ein Unicode Zeichen sein muss. Die zuletzt genannte Methode kann mit *unset\_invisible\_char()* rückgängig gemacht werden.
- *set\_max\_length(int)* schneidet jede Einabe länger als *int* ab, um die gewünschte maximale Länge zu erhalten. *int* muss eine Ganzzahl sein.
- Standardmäßig wird das Signal *"activate"* ausgesendet, wenn man die Eingabetaste drückt. 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)*.
- Um einen Rahmen um das Eingabefeld setzen: *set\_has\_frame(TRUE)*.
- *set\_placeholder\_text('some text')* bestimmt den Text, der im Eingabefeld angezeigt werden soll, wenn er leer und unfokussiert ist.
- *set\_overwrite\_mode(TRUE)* und *set\_overwrite\_mode(FALSE)* sind selbsterklärend.
- Mit *set\_editable(FALSE)* kann der Benutzer den Text im Eingabeelement nicht bearbeiten.
- *set\_completion(\$completion)*, wobei *\$completion* ein *Gtk3::EntryCompletion* Element ist, setzt die Vervollständigung fest oder deaktiviert sie, wenn *\$completion* 'None' ist.
- Ein Eingabefeld kann Fortschritts- oder Aktivitätsinformationen hinter dem Text anzeigen. Mit *set\_progress\_fraction(fraction)*, wobei *fraction* eine Fließkommazahl zwischen 0.0 und inklusiv 1.0 ist, kann das Eingabefeld zu dem angegebenen Bruchteil gefüllt werden. 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()*).
- Ein Eingabefeld kann auch Icons anzeigen. Diese Symbole können durch Klick aktivierbar



sein, können als Drag-Quelle eingestellt sein and können Tooltips haben.

Um ein Icon hinzuzufügen, kann man `set_icon_from_stock(icon_position, stock_id)`, `set_icon_from_pixbuf(icon_position, pixbuf)` oder `set_icon_from_icon_name(icon_position, icon_name)` verwenden, wobei `icon_position` entweder `primary`<sup>7</sup> (Das Icon wird am Beginn des Eingabefeldes dargestellt) oder `secondary`<sup>8</sup> (Das Icon wird am Ende des Eingabefeldes dargestellt) sein kann. Um de mIcon einen Tooltip hinzuzufügen, benutzen wir `set_icon_tooltip_text('tooltip text')` oder `set_icon_tooltip_markup('tooltip text in Pango markup language')`.

### 8. 3. Scale



Schiebe die Skalen!

#### Code used to generate this example

```
#!/usr/bin/perl

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

# ein Fenster
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()});

# zwei Gtk3::Adjustments (d.h. zwei Einstellungselemente) mit folgenden Parametern:
# Anfangswert, Minimalwert, Maximalwert,
# Einzelschritterhöhung (= Erhöhung, wenn die Pfeiltasten oder die +/- Buttons gedrückt werden),
# Seitenerhöhung (Klicke auf die Griffe, um es zu sehen!),
# Seitengröße (hier nicht verwendet)
```

<sup>7</sup> Or `Gtk3::EntryIconPosition::PRIMARY`

<sup>8</sup> Or `Gtk3::EntryIconPosition::SECONDARY`

```

my $ad1 = Gtk3::Adjustment->new(0,0,100,5,10,0);
my $ad2 = Gtk3::Adjustment->new(50,0,100,5,10,0);

# eine horizontale Skala
my $h_scale = Gtk3::Scale->new('horizontal',$ad1);
# von Ganzzahlen (d.h. ohne Nachkommastellen)
$h_scale->set_digits(0);
# welche sich horizontal ausdehnen kann, wenn im Raster Platz ist (siehe unten)
$h_scale->set_hexpand(TRUE);
# und die am Beginn des im Raster freien Platzes ausgerichtet ist (siehe unten)
$h_scale->set_valign('start');

# Das Signal "value-changed", das von der Skala ausgesendet wird, wird mit
# der Callback Funktion scale_moved verbunden
$h_scale->signal_connect('value-changed' => \&scale_moved);

# eine vertikale Skala
my $v_scale = Gtk3::Scale->new('vertical',$ad2);
# welche sich vertikal ausdehnen kann, wenn im Raster Platz ist (siehe unten)
$v_scale->set_vexpand(TRUE);

# Das Signal "value-changed", das von der Skala ausgesendet wird, wird mit
# der Callback Funktion scale_moved verbunden
$v_scale->signal_connect('value-changed' => \&scale_moved);

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

# ein Raster, um die Elemente anzuordnen
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);

# Zeige das Fenster und starte die Anwendung
$window->show_all();
Gtk3->main();

# Jedes Signal von den Skalen wird an das Label gesendet, dessen Text verändert wird
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

Um einen Skala zu konstruieren wird ein `Gtk3::Adjustment` benötigt. Dieses repräsentiert den Anfangswert, die untere und obere Grenze, zusammen mit der Einzelschritt- und Seiten-Erhöhung und der Seitengröße, und wird erstellt mit dem Befehl `"my $adjustment=Gtk3::Adjustment->new(initial value, minimum value, maximum value, step_increment, page_increment, page_size)"`, wobei die einzelnen Werte Fließzahlen sind. `step_increment` ist die Erhöhung/Verringerung, die durch das Drücken der Pfeiltasten bewirkt wird, `page_increment` ist die Erhöhung/Verringerung, die durch das Klicken auf die Skala selbst bewirkt wird. Beachte, dass `page_size` in diesem Fall nicht verwendet wird und daher auf 0 gesetzt werden sollte.

In Zeile 34 wird das `"value-changed"` Signal mit der Callback Funktion `scale_moved()` mit der Methode `$widget->signal_connect("signal", \&callback_function)` verbunden. Für eine detailliertere Erklärung siehe das Kapitel Signals and callbacks.

- `get_value()` liefert den aktuellen Wert der Skala; `set_value(value)` setzt ihn (wenn der Wert `value`, vom Typ eine Fließkommazahl, außerhalb des minimalen oder maximalen Bereichs ist, wird er abgeschnitten, um in diesen Bereich zu passen). Diese Methoden sind Methoden der Klasse `Gtk3::Range`.
- Um zu verhindern, dass der aktuelle Wert als eine Zeichenkette neben dem Schieber angezeigt wird, benutzt man `set_draw_value(FALSE)`
- Um den Teil der Skala zwischen dem ursprgl. und aktuellen Wert hervorzuheben (noch nicht getestet!), benutzt man

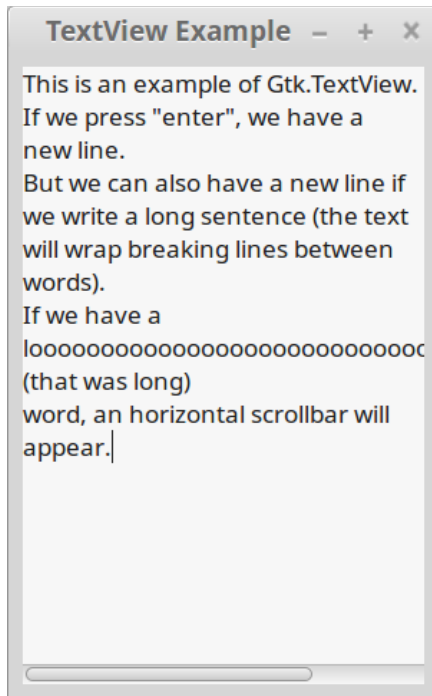
```
$h_scale->set_restrict_to_fill_level(FALSE);  
$h_scale->set_fill_level($h_scale->get_value());  
$h_scale->set_show_fill_level(TRUE);
```

in der Callback Funktion des `"value-changed"` Signals, um eine neue Füllung zu haben jedes Mal, wenn der Wert sich ändert. Diese Methoden sind Methoden der Klasse `Gtk3::Range`.

- `add_mark(value, position, markup)` fügt eine Markierung am Wert `value` (Fließkommazahl oder Ganzzahl, je nach Genauigkeit der Skala) ein, an der Position `"left"`, `"right"`, `"top"` oder `"bottom"` mit dem Text `Null` oder `markup` in der Pango Markup Sprache. Um die Markierungen zu löschen, benutzt man `clear_marks()`.
- `set_digits(digits)` legt die Genauigkeit der Skala auf `digits` Nachkommastellen fest

## 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
```

```

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

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

$window->add($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*<sup>9</sup> - 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 *set\_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

---

<sup>9</sup> 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 ('striketrough' 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

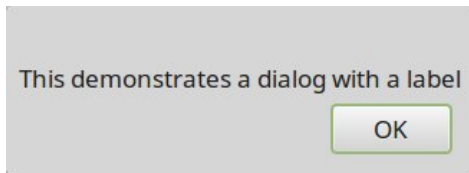
- *forward\_search(needle, flags, limit)* searches forward for needle. The search will not continue past the *Gtk.TextIter* limit. The flags can be set to one of the following, or any combination of it by concatenating them with the bitwise-OR operator |: 0 (the match must be exact); *Gtk.TextSearchFlags.VISIBLE\_ONLY* (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 program 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();
}

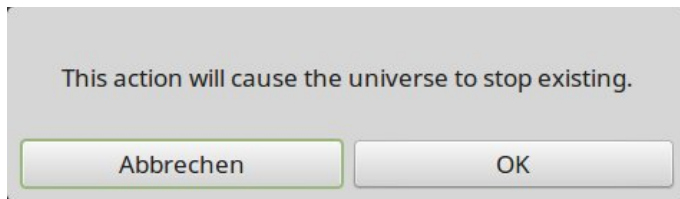
# destroy the aboutdialog
sub on_close {
    my ($aboutdialog) = @_ ;
    $aboutdialog->destroy();
}

```

### ***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 action for 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*

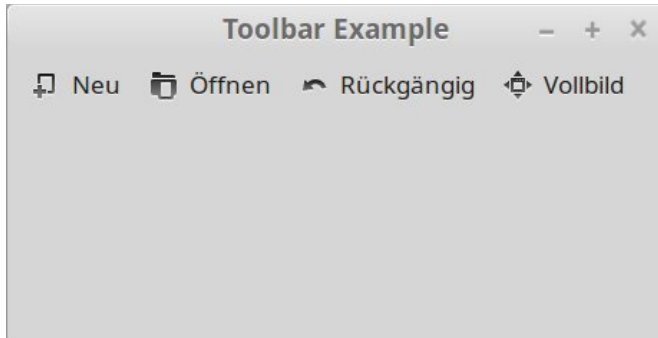
In line 57 the signal *'activate'* from the action *about\_action* is connected to the callback function *about\_callback()* using *\$action->signal\_connect('signal'=>\&callback function)*. See Signals and callbacks for a more detailed explanation.

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_hexexpand(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 occurred, 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 $parameter which here is undef!!!
    my ($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');
        $toolbar->set_icon_widget($fullscreen_icon);
        $toolbar->set_label('Vollbild');
        $fullscreen_icon->show();
    }
    else {
        $window->fullscreen();
        my $leave_fullscreen_icon = Gtk3::Image->new_from_icon_name('view-restore',
'16');
        $toolbar->set_icon_widget($leave_fullscreen_icon);
        $toolbar->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***

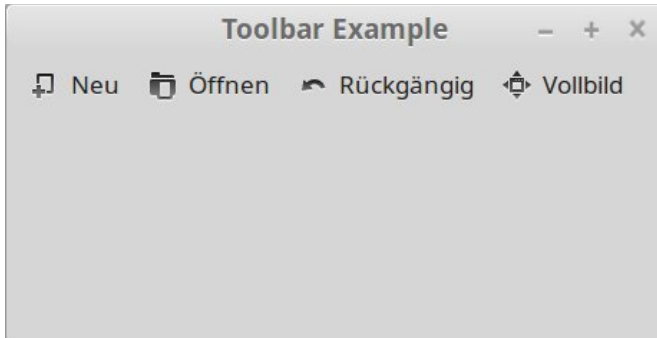
In line 114 the signal 'activate' from the action *undo\_action* is connected to the callback function *about\_callback()* using *\$action->signal\_connect('signal'=>\&callback function)*. See Signals and callbacks for a more detailed explanation.

- 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)

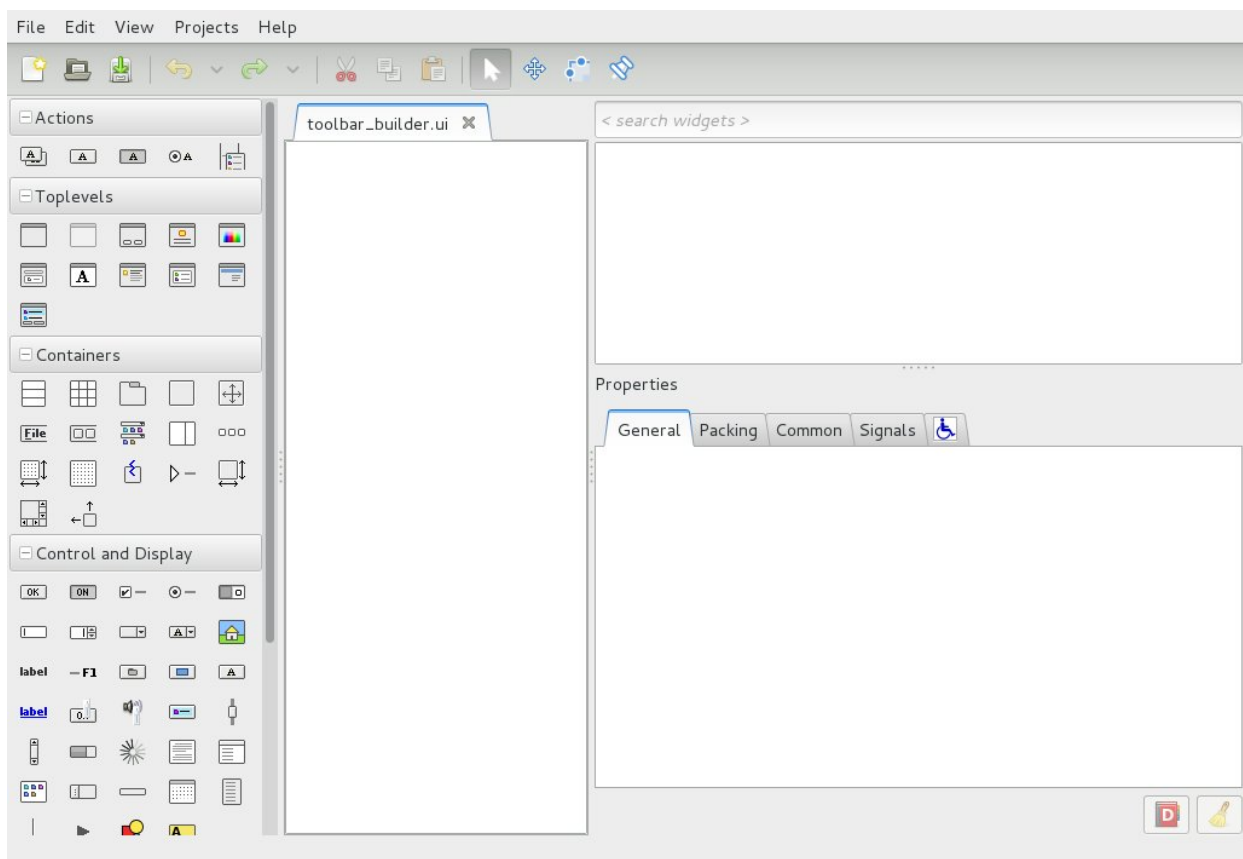
## 11. 5. Toolbar created using Glade



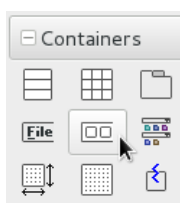
This example is similar to the Toolbar example above, except we use Glade to create the toolbar in an XML .ui file

### *Creating the toolbar with Glade*

To create the toolbar using [Glade Interface Designer](#):

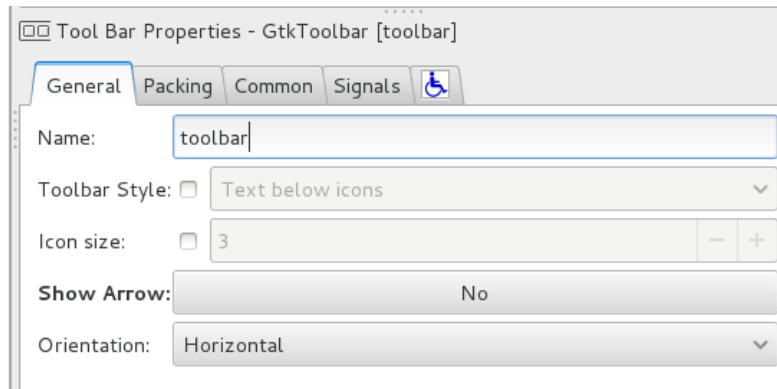


1. Open Glade and save the file as `11_5_toolbar_builder.ui` Under Containers on the left hand side, right click on the toolbar icon and select Add widget as toplevel.

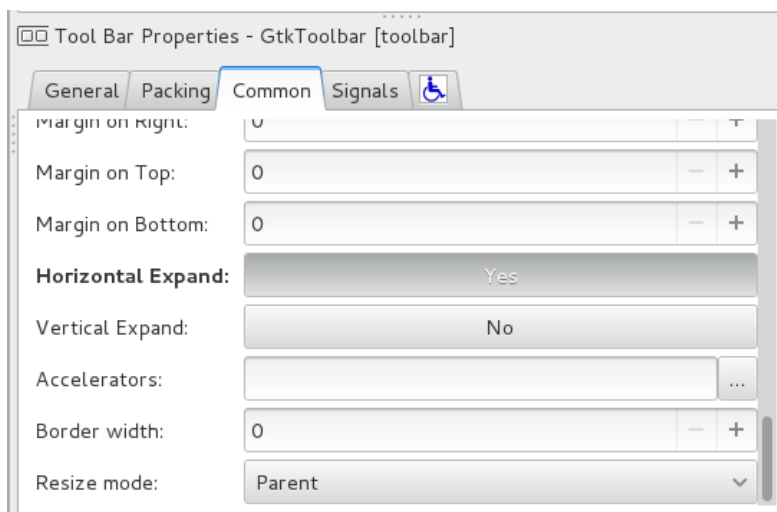


2. Under the General tab on the bottom right, change the Name to toolbar and Show Arrow to

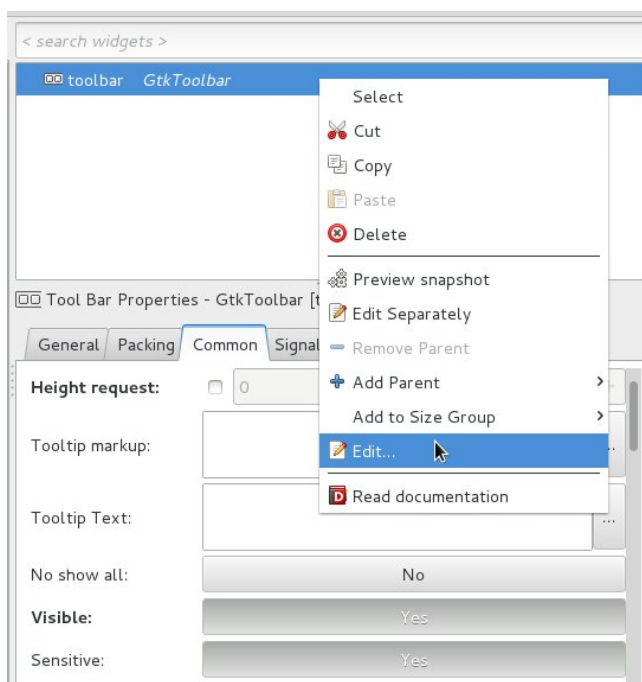
No.



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



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

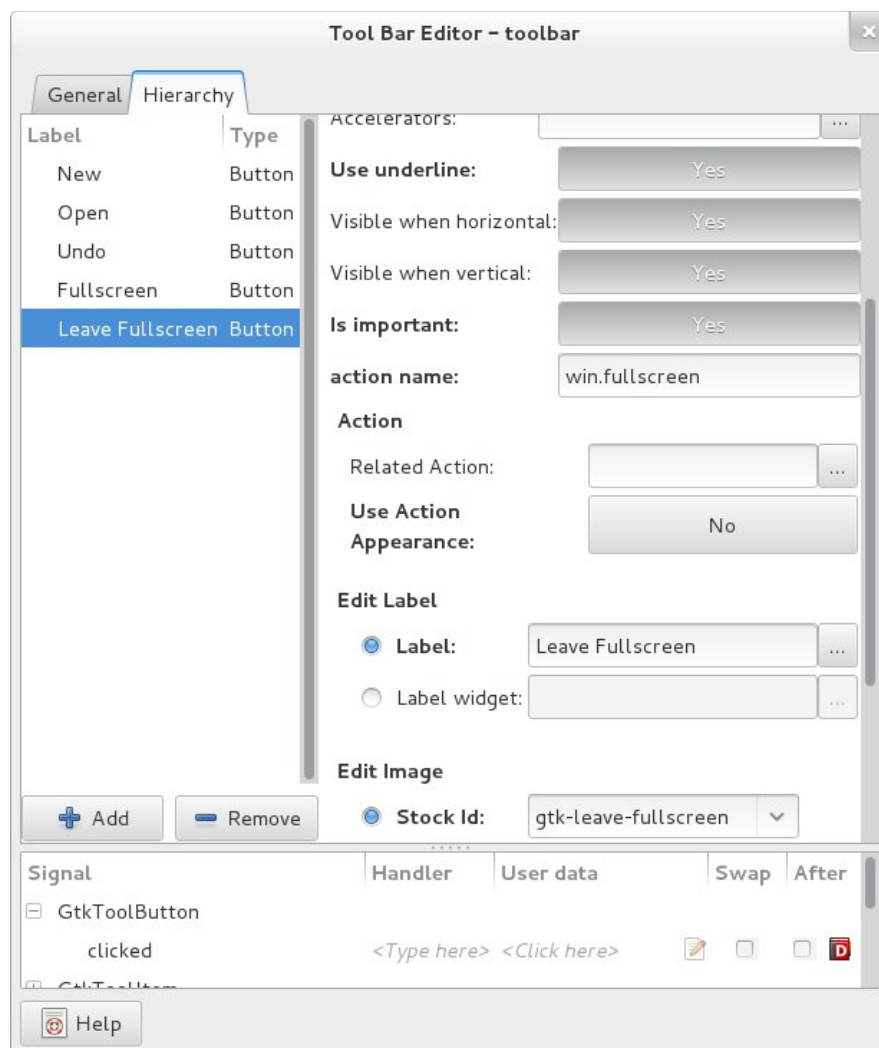


5. We want to add 5 ToolButtons: New, Open, Undo, Fullscreen and Leave Fullscreen. First, we will add the New ToolButton.

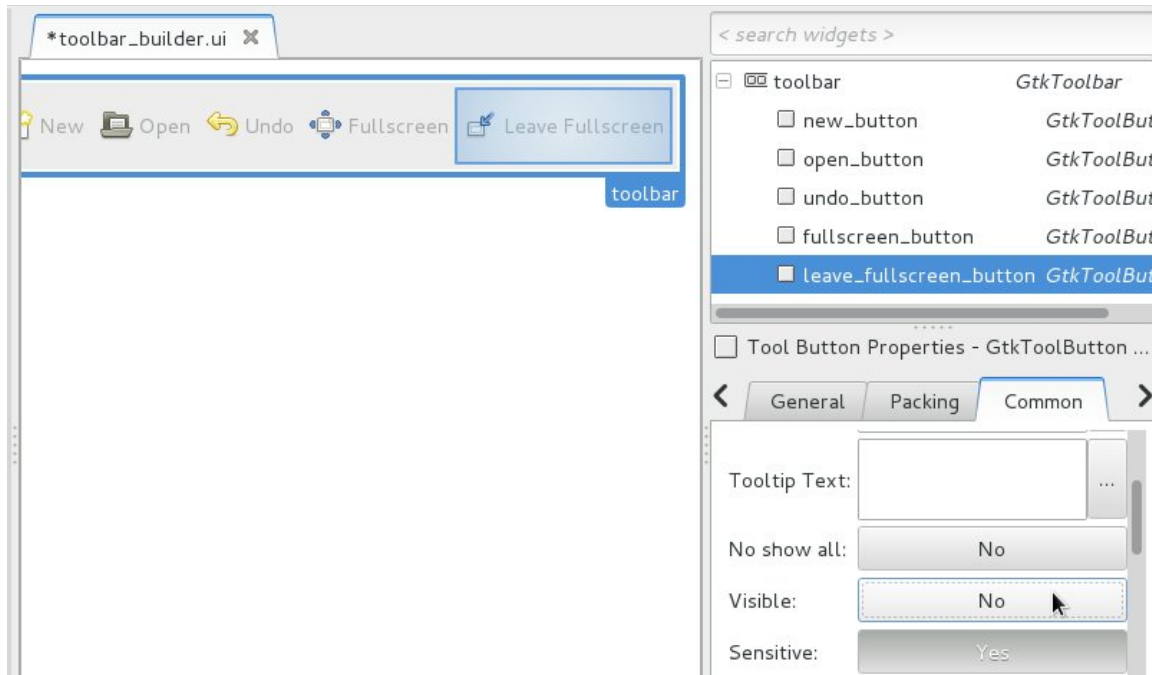
1. Under Hierarchy tab, click Add.
2. Change the name of the ToolItem to new\_button.
3. Scroll down and set Is important to Yes. This will cause the label of the ToolButton to be shown, when you view the toolbar.
4. Enter the action name: app.new.
5. Change the Label to New.
6. Select the New Stock 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



6. Close the Tool Bar Editor.
7. 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.



8. Save your work, and close Glade
9. 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";
}

```

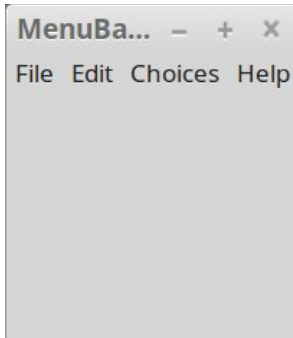
### ***Useful methods for Gtk3::Builder***

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



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 separte 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">&lt;Primary&gt;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, simply set *translatable="yes"*:

```

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

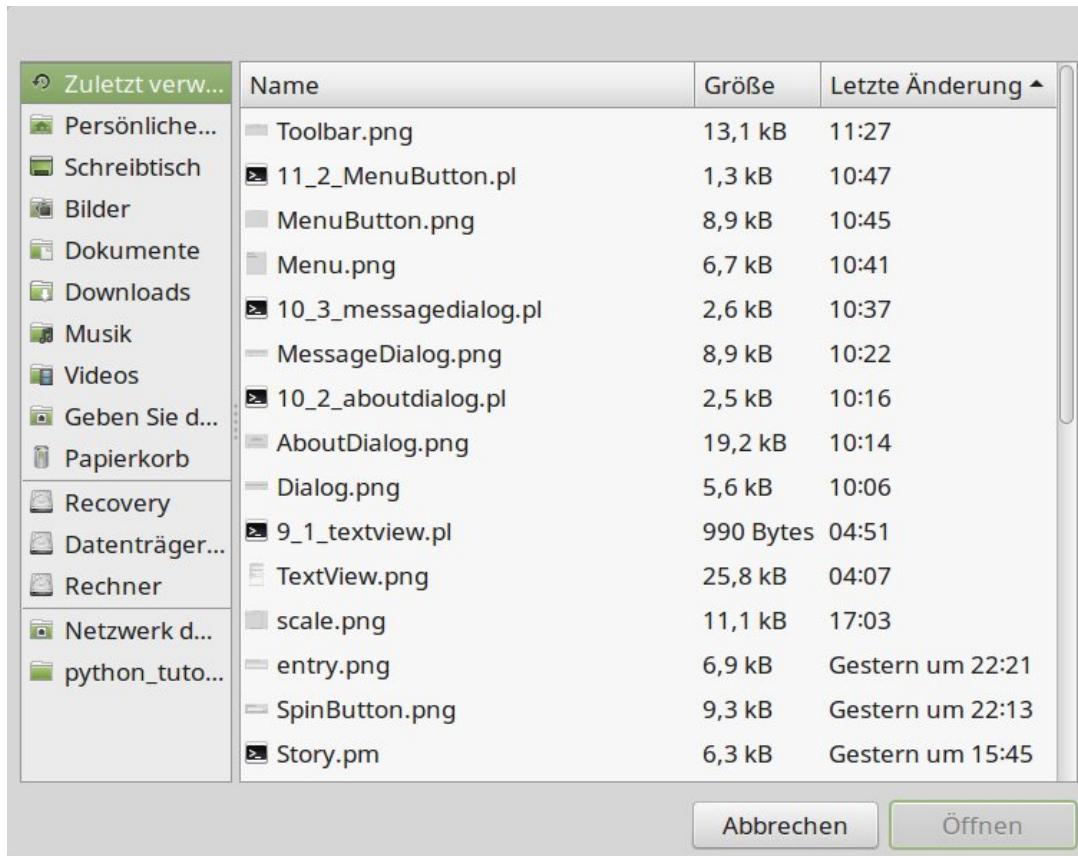
```

## 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 bytestring 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 perlsh 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***

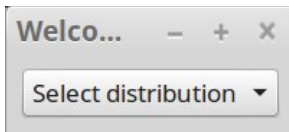
Note that the *action* of the FileChooserDialog can be one of the following: *"open"* (the file chooser will only let the user pick an existing file), *"save"* (the file chooser will let the user pick an existing file, or type in a new filename), *"select\_folder"* (the file chooser will let the user 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) {

    # Das Hinzufügen von Daten geschieht in 2 Schritten
    # 1) Dem Treestore wird eine leere Reihe hinzugefügt und zu dieser Reihe
    # wird eine Referenz bzw. ein Pointer ($iter) erstellt
    # 2) Um die Reihe mit Inhalt zu füllen, muss die Gtk3::Treestore
    # set-Methode auf diese angewendet werden
    # !!! Die Liste der Paare muss so viele Elemente enthalten wie die
    # Zahl der Spalten im Tree- bzw. ListStore!!!
    # example: $liststore ->set ($iter, 0 => "Inhalt der Zeile in Spalte 1",
    # 1 => ("Inhalt der Zeile in der Spalte 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

First Name	Last Name	Phone Number
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

Jurg

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; $i++) {
    # Das Hinzufügen von Daten geschieht in 2 Schritten
    # 1) Dem Treestore wird eine leere Reihe hinzugefügt und zu
    # dieser Reihe wird eine Referenz bzw. ein Pointer ($iter)
    # erstellt
    # 2) Um die Reihe mit Inhalt zu füllen, muss die
    # Gtk3::Treestore set-Methode auf diese angewendet werden
    # !!! Die Liste der Paare muss so viele Elemente enthalten wie
    # die Zahl der Spalten im Tree- bzw. ListStore!!!
    # example: $liststore ->set ($iter, 0 => "Inhalt der Zeile in
```

```
# Spalte 1", 1 => ("Inhalt der Zeile in der Spalte 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;
}

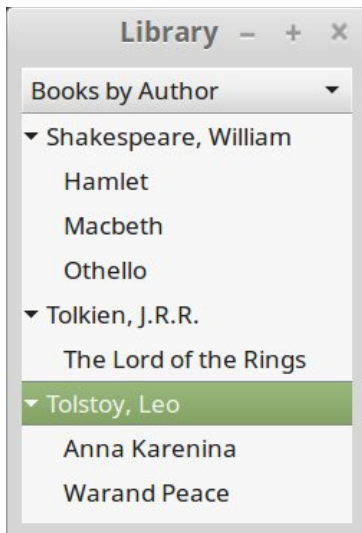
```

### ***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 model
# 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; $i++) {
    # Vorbemerkung: Das Hinzufügen von Daten geschieht grds. in 2 Schritten
    # 1) Dem Treestore wird eine leere Reihe hinzugefügt und zu
    # dieser Reihe wird eine Referenz bzw. ein Pointer ($iter)
    # erstellt
    # 2) Um die Reihe mit Inhalt zu füllen, muss die
    # Gtk3::TreeStore set-Methode auf diese angewendet werden
    # !!! Die Liste der Paare muss so viele Elemente enthalten wie
    # die Zahl der Spalten im Tree- bzw. ListStore!!!
    # example: $liststore ->set ($iter, 0 => "Inhalt der Zeile in
```



```

# Spalte 1", 1 => ("Inhalt der Zeile in der Spalte 2 etc.)

# Zunächst wird das Eltern Iter erstellt und die Eltern Zellen eingefügt
# diese befinden sich jeweils an erster Stelle (d.h. $books[$i][0]
# i.Ü. wird nur eine Spalte benötigt
my $iter = $store->append();
$store->set($iter, 0 => "$books[$i][0]");

for (my $j=1; $j <= $#{$books[$i]}; $j++) {
    # in dieser Spalte 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++) {

    # Das Hinzufügen von Daten geschieht in 2 Schritten
    # 1) Dem Treestore wird eine leere Reihe hinzugefügt und zu dieser Reihe
    # wird eine Referenz bzw. ein Pointer ($iter) erstellt
    # 2) Um die Reihe mit Inhalt zu füllen, muss die Gtk3::Treestore
    # set-Methode auf diese angewendet werden
    # !!! Die Liste der Paare muss so viele Elemente enthalten wie die
    # Zahl der Spalten im Tree- bzw. ListStore!!!
    # example: $liststore ->set ($iter, 0 => "Inhalt der Zeile in Spalte 1",
    # 1 => ("Inhalt der Zeile in der Spalte 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');

# speichere die Anzahl der Zeilen in einer eignen Variablen
my $row_count = 0;

# append the values in the model
for (my $i=0; $i <= $#list_of_dvds; $i++) {
    # Das Hinzufügen von Daten geschieht in 2 Schritten
    # 1) Dem Treestore wird eine leere Reihe hinzugefügt und zu
    # dieser Reihe wird eine Referenz bzw. ein Pointer ($iter)
    # erstellt
    # 2) Um die Reihe mit Inhalt zu füllen, muss die
```

```
# Gtk3::Treestore set-Methode auf diese angewendet werden
# !!! Die Liste der Paare muss so viele Elemente enthalten wie
# die Zahl der Spalten im Tree- bzw. ListStore!!!
# example: $liststore ->set ($iter, 0 => "Inhalt der Zeile in
# Spalte 1", 1 => ("Inhalt der Zeile in der Spalte 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 add 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: $row_count (see below)
if ($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";
    }
}

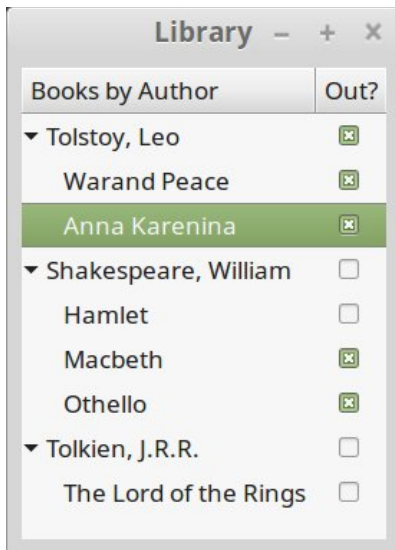
```

### ***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 model
# 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; $i++) {
    # Vorbemerkung: Das Hinzufügen von Daten geschieht grds. in 2 Schritten
    # 1) Dem Treestore wird eine leere Reihe hinzugefügt und zu
    # dieser Reihe wird eine Referenz bzw. ein Pointer ($iter)
    # erstellt
    # 2) Um die Reihe mit Inhalt zu füllen, muss die
    # Gtk3::TreeStore set-Methode auf diese angewendet werden
    # !!! Die Liste der Paare muss so viele Elemente enthalten wie
    # die Zahl der Spalten im Tree- bzw. ListStore!!!
    # example: $liststore ->set ($iter, 0 => "Inhalt der Zeile in
```



```
# Spalte 1", 1 => ("Inhalt der Zeile in der Spalte 2 etc.)
```

```
# Zunächst wird das Eltern Iter erstellt und die Eltern Zellen eingefügt
```

```
# diese befinden sich jeweils an erster Stelle (d.h. $books[$i][0]
```

```
# i.Ü. wird nur eine Spalte benötigt
```

```
my $iter = $store->append();
```

```
$store->set($iter, 0 => "$books[$i][0]", 1 => FALSE);
```

```
for (my $j=1; $j <= ${$books[$i]}; $j++) {
```

```
    # in dieser Spalte 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 signal 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 Gtk3::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](mailto: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](mailto:gtk-perl-list@gnome.org)) is a better place to ask.