# class GTK::V3::Glib::GObject

GObject — The base object type

## Table of Contents

## Synopsis

Top level class of almost all classes in the GTK, GDK and Glib libraries.

This object is almost never used directly. Most of the classes inherit from this class. The below example can be made much simpler by setting the label directly in the init of GtKLabel. The purpose of this example, however, is that there are other properties which can only be set this way. Also not all types are covered yet by GValue and GType.

```
use GTK::V3::Glib::GObject;
use GTK::V3::Glib::GValue;
use GTK::V3::Glib::GType;
use GTK::V3::Gtk::GtkLabel;

my GTK::V3::Glib::GType $gt .= new;
my GTK::V3::Glib::GValue $gv .= new(:init(G_TYPE_STRING));

my GTK::V3::Gtk::GtkLabel $label1 .= new(:label(''));
$gv.g-value-set-string('label string');
$label1.g-object-set-property( 'label', $gv);
```

### [g_object_] set_property

```
method g_object_set_property (
  Str $property_name, GTK::V3::Glib::GValue $value
)
```

Sets a property on an object.

- $property_name; the name of the property to set.
- $value; the value.

### [g_object_] get_property

```
method g_object_get_property (
  Str $property_name, GTK::V3::Glib::GValue $value is rw
)
```

Gets a property of an object. value must have been initialized to the expected type of the property (or a type to which the expected type can be transformed) using g_value_init().

In general, a copy is made of the property contents and the caller is responsible for freeing the memory by calling g_value_unset().

- $property_name; the name of the property to get.
- $value; return location for the property value.

## g_object_notify

```
method g_object_notify ( Str $property_name )
```

Emits a notify signal for the property property_name on object .

When possible, e.g. when signaling a property change from within the class that registered the property, you should use g_object_notify_by_pspec()(not supported yet) instead.

Note that emission of the notify signal may be blocked withg_object_freeze_notify(). In this case, the signal emissions are queued and will be emitted (in reverse order) when g_object_thaw_notify() is called.

- $property_name; the name of a property installed on the class of object.

## [g_object_] freeze_notify

```
method g_object_freeze_notify ( )
```

Increases the freeze count on object . If the freeze count is non-zero, the emission ofnotify signals on object is stopped. The signals are queued until the freeze count is decreased to zero. Duplicate notifications are squashed so that at most one notify signal is emitted for each property modified while the object is frozen.

This is necessary for accessors that modify multiple properties to prevent premature notification while the object is still being modified.

## [g_object_] thaw_notify

```
method g_object_thaw_notify ( )
```

Reverts the effect of a previous call tog_object_freeze_notify(). The freeze count is decreased on object and when it reaches zero, queued notify signals are emitted.

Duplicate notifications for each property are squashed so that at most onenotify signal is emitted for each property, in the reverse order in which they have been queued.

It is an error to call this function when the freeze count is zero.

## new

### multi submethod BUILD ( :$widget! )

Please note that this class is mostly not instantiated directly but is used indirectly when a child class is instantiated.

Create a Perl6 widget object using a native widget from elsewhere. $widget can be a N-GOBject or a Perl6 widget like GTK::V3::Gtk::GtkButton.

```
# some set of radio buttons grouped together
my GTK::V3::Gtk::GtkRadioButton $rb1 .= new(:label('Download everything'));
my GTK::V3::Gtk::GtkRadioButton $rb2 .= new(
  :group-from($rb1), :label('Download core only')
);

# get all radio buttons of group of button $rb2
my GTK::V3::Glib::GSList $rb-list .= new(:gslist($rb2.get-group));
loop ( Int $i = 0; $i < $rb-list.g_slist_length; $i++ ) {
  # get button from the list
  my GTK::V3::Gtk::GtkRadioButton $rb .= new(
    :widget($rb-list.nth-data-gobject($i))
  );

  if $rb.get-active == 1 {
    # execute task for this radio button

    last;
  }
}
```

Another example is a difficult way to get a button.

```
my GTK::V3::Gtk::GtkButton $start-button .= new(
  :widget(GTK::V3::Gtk::GtkButton.gtk_button_new_with_label('Start'))
);
```

### multi submethod BUILD ( Str :$build-id! )

Create a Perl6 widg #`{{ if $setup-event-handler { $handler = -> N-GObject $w, GdkEvent $event, OpaquePointer $d { $handler-object."$handler-name"( :widget(self), :$event, |%user-options ); }

```
$!g-signal._g_signal_connect_object_event(
  $signal-name, $handler, OpaquePointer, $connect-flags
);
```

```
}

elsif $setup-nativewidget-handler {
  $handler = -> N-GObject $w, OpaquePointer $d1, OpaquePointer $d2 {
    $handler-object."$handler-name"(
      :widget(self), :nativewidget($d1), |%user-options
    );
  }

  $!g-signal._g_signal_connect_object_nativewidget(
    $signal-name, $handler, OpaquePointer, $connect-flags
  );
}

else {
  $handler = -> N-GObject $w, OpaquePointer $d {
    $handler-object."$handler-name"( :widget(self), |%user-options);
  }

  $!g-signal._g_signal_connect_object_signal(
    $signal-name, $handler, OpaquePointer, $connect-flags
  );
}
```

}} et object using a GtkBuilder. The GtkBuilder class will handover its object address to theGObject and can then be used to search for id's defined in the GUI glade design.

```
my GTK::V3::Gtk::GtkBuilder $builder .= new(:filename<my-gui.glade>);
my GTK::V3::Gtk::GtkButton $button .= new(:build-id<my-gui-button>);
```

## debug

```
method debug ( Bool :$on )
```

There are many situations when exceptions are retrown within code of a callback method, Perl6 is not able to display the error properly (yet). In those cases you need another way to display errors and show extra messages leading up to it.

## register-signal

```
method register-signal (
  $handler-object, Str:D $handler-name, Str:D $signal-name,
  Int :$connect-flags = 0, *%user-options
  --> Bool
)
```

Register a handler to process a signal or an event. There are several types of callbacks which can be handled by this regstration. They can be controlled by using a named argument with a special name.

- Events. The GTK will call a function with a structure holding the event information. When the user handler has a **named argument** named **event** it is assumed that the handler code is made is to handle events like key presses.
- There are also callbacks which get extra native widgets. An example of this is therow-selected signal from GtkListBox. The callback gets a native GtkListBoxRow widget. To handle these signals, the user can define a handler with a **named argument** that is named **nativewidget**.
- Otherwise it is assumed that the code handles signals like button clicks. Information about signal names available to widgets can be found at the GTK developers site, e.g. for GtkButton click signals here and for a mouse button press event here. Notice that in the latter case you can see that one of the arguments has an argument type of **GdkEvent**.

- $handler-object is the object wherein the handler is defined.
- $handler-name is name of the method. Its signature is one of

```
handler ( object: :$widget, :$user-option1, ..., :$user-optionN )
```

or

```
handler ( object: :$widget, :$event, :$user-option1, ..., :$user-optionN )
```

or

```
handler ( object: :$widget, :$nativewidget, :$user-option1, ..., :$user-optionN )
```

The arguments are all optional but to register an event handler, the**:$event** argument must be present.

- $signal-name is the name of the event to be handled. Each gtk widget has its own series of signals, please look for it in the documentation of gtk.
- $connect-flags can be one of G_CONNECT_AFTER or G_CONNECT_SWAPPED. See documentation here.
- %user-options. Any other user data in whatever type. These arguments are provided to the user handler when an event for the handler is fired. There will always be one named argument :$widget which holds the class object on which the signal was registered. The name 'widget' is therefore reserved. An other reserved named argument is of course :$event.

```
# create a class holding a handler method to process a click event
# of a button.
class X {
  method click-handler ( :widget($button), Array :$user-data ) {
    say $user-data.join(' ');
  }
}

# create a button and some data to send with the signal
my GTK::V3::Gtk::GtkButton $button .= new(:label('xyz'));
my Array $data = [<Hello World>];

# register button signal
my X $x .= new(:empty);
$button.register-signal( $x, 'click-handler', 'clicked', :user-data($data));
```

# Signals

Registering example

```
class MyHandlers {
  method my-click-handler ( :$widget, :$my-data ) { ... }
}

# elsewhere
my MyHandlers $mh .= new;
$button.register-signal( $mh, 'click-handler', 'clicked', :$my-data);
```

See also method register-signal in GTK::V3::Glib::GObject.

## Not yet supported signals

### notify

The notify signal is emitted on an object when one of its properties has its value set through g_object_set_property(), g_object_set(), et al.

Note that getting this signal doesn’t itself guarantee that the value of the property has actually changed. When it is emitted is determined by the derived GObject class. If the implementor did not create the property with G_PARAM_EXPLICIT_NOTIFY, then any call to g_object_set_property() results innotify being emitted, even if the new value is the same as the old. If they did pass G_PARAM_EXPLICIT_NOTIFY, then this signal is emitted only when they explicitly call g_object_notify() or g_object_notify_by_pspec(), and common practice is to do that only when the value has actually changed.

This signal is typically used to obtain change notification for a single property, by specifying the property name as a detail in the g_signal_connect() call, like this:

Signal notify is not yet supported.