🍀 Gnome::GObject::Object

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 Gnome::GObject::Object;
use Gnome::GObject::Value;
use Gnome::GObject::Type;
use Gnome::Gtk::Label;

my Gnome::GObject::Type $gt .= new;
my Gnome::GObject::Value $gv .= new(:init(G_TYPE_STRING));

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

new

multi method new (:$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 Gnome::Gtk::Button.

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

# get all radio buttons of group of button $rb2
my Gnome::GObject::SList $rb-list .= new(:gslist($rb2.get-group));
loop ( Int $i = 0; $i < $rb-list.g_slist_length; $i++ ) {
  # get button from the list
  my Gnome::Gtk::RadioButton $rb .= new(
    :widget($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 Gnome::Gtk::Button $start-button .= new(
  :widget(Gnome::Gtk::Button.gtk_button_new_with_label('Start'))
);
```

multi method new (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 the `GObject` and can then be used to search for id's defined in the GUI glade design.

```
my Gnome::Gtk::Builder $builder .= new(:filename<my-gui.glade>);
my Gnome::Gtk::Button $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

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.

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

- $handler-object is the object wherein the handler is defined.

- $handler-name is name of the method. Commonly used signatures for those handlers are

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

Other forms are explained in the widget documentations when signals are provided.

- $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.

- %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 Gnome::Gtk::Button $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))
```

start-thread

Start a thread in such a way that the function can modify the user interface in a save way and that these updates are automatically made visible without explicitly process events queued and waiting in the main loop.

```
method start-thread (
   $handler-object, Str:D $handler-name, Int $priority = G_PRIORITY_DEFAULT,
   Bool :$new-context = False, *%user-options
   --> Promise
)
```

- $handler-object is the object wherein the handler is defined.

- $handler-name is name of the method.

- $priority; The priority to which the handler is started. The default is G_PRIORITY_DEFAULT. These are constants defined in Gnome::GObject::GMain.

- $new-context; Whether to run the handler in a new context or to run it in the context of the main loop. Default is to run in the main loop.

- *%user-options; Any name not used above is provided to the handler

Returns a Promise object. If the call fails, the object is undefined.

The handlers signature is at least :$widget of the object on which the call was made. Furthermore all users named arguments to the call defined in *%user-options. The handler may return any value which becomes the result of the Promise returned from start-thread.

[g_object_] set_property

```
method g_object_set_property (
   Str $property_name, Gnome::GObject::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, Gnome::GObject::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 with `g_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 of `notify` 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 to `g_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 one `notify` 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.

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 Gnome::GObject::Object.

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 in `notify` 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.