NAME

Glib::Object::Subclass – register a perl class as a GObject class

SYNOPSIS

```
use Glib::Object::Subclass
   Some::Base::Class::,   # parent class, derived from Glib::Object
   signals => {
          something_changed => {
             class_closure => sub { do_something_fun () },
             flags         => [qw(run-first)],
             return_type   => undef,
             param_types   => [],
          },
          some_existing_signal => \&class_closure_override,
   },
   properties => [
      Glib::ParamSpec->string (
         'some_string',
         'Some String Property',
         'This property is a string that is used as an example',
         'default value',
         [qw/readable writable/]
      ),
   ];
```

DESCRIPTION

This module allows you to create your own GObject classes, which is useful to e.g. implement your own Gtk2 widgets.

It doesn't ''export'' anything into your namespace, but acts more like a pragmatic module that modifies your class to make it work as a GObject class.

You may be wondering why you can't just bless a Glib::Object into a different package and add some subs. Well, if you aren't interested in object parameters, signals, or having your new class interoperate transparently with other GObject-based modules (e.g., Gtk2 and friends), then you can just re-bless.

However, a GObject's signals, properties, virtual functions, and GInterface implementations are specific to its GObjectClass. If you want to create a new GObject which was a derivative of GtkDrawingArea, but adds a new signal, you must create a new GObjectClass to which to add the new signal. If you don't, then *all* of the GtkDrawingAreas in your application will get that new signal!

Thus, the only way to create a new signal or object property in the Perl bindings for Glib is to register a new subclass with the GLib type system via **Glib::Type::register_object**(). The Glib::Object::Subclass module is a Perl-developer-friendly interface to this bit of paradigm mismatch.

USAGE

This module works similar to the `use base` pragma in that it registers the current package as a subclass of some other class (which must be a GObjectClass implemented either in C or some other language).

The pragma requires at least one argument, the parent class name. The remaining arguments are key/value pairs, in any order, all optional:

– properties => []
: Add object properties; see ''PROPERTIES''.

– signals => {}
: Add or override signals; see ''SIGNALS'' and ''OVERRIDING BASE METHODS''.

– interfaces => []
: Add GInterfaces to your class; see ''INTERFACES''.

(Actually, these parameters are all passed straight through to **Glib::Type::register_object**(), adding

__PACKAGE__ (the current package name) as the name of the new child class.)

OBJECT METHODS AND FUNCTIONS

The following methods are either added to your class on request (not yet implemented), or by default unless your own class implements them itself. This means that all these methods and functions will get sensible default implementations unless explicitly overwritten by you (by defining your own version).

Except for `new`, all of the following are *functions* and no *methods*. That means that you should *not* call the superclass method. Instead, the GObject system will call these functions per class as required, emulating normal inheritance.

`$class->new` (attr => value, ...)

The default constructor just calls `Glib::Object::new`, which allows you to set properties on the newly created object. This is done because many `new` methods inherited by Gtk2 or other libraries don't have `new` methods suitable for subclassing.

INIT_INSTANCE `$self` [not a method]

`INIT_INSTANCE` is called on each class in the hierarchy as the object is being created (i.e., from `Glib::Object::new` or our default `new`). Use this function to initialize any member data. The default implementation will leave the object untouched.

GET_PROPERTY `$self`, `$pspec` [not a method]

SET_PROPERTY `$self`, `$pspec`, `$newval` [not a method]

`GET_PROPERTY` and `SET_PROPERTY` are called whenever somebody does `$object->get ($propname)` or `$object->set ($propname => $newval)` (from other languages, too). The default implementations hold property values in the object hash, equivalent to

```
sub GET_PROPERTY {
  my ($self, $pspec) = @_;
  my $pname = $pspec->get_name;
  return (exists $self->{$pname} ? $self->{$pname}
          : $pspec->get_default_value);  # until set
}
sub SET_PROPERTY {
  my ($self, $pspec, $newval) = @_;
  $self->{$pspec->get_name} = $newval;
}
```

Because `$pspec->get_name` converts hyphens to underscores, a property `"line-style"` is in the hash as `line_style`.

These methods let you store/fetch properties in any way you need to. They don't have to be in the hash, you can calculate something, read a file, whatever.

Most often you'll write your own `SET_PROPERTY` so you can take action when a property changes, like redraw or resize a widget. Eg.

```
sub SET_PROPERTY {
  my ($self, $pspec, $newval) = @_;
  my $pname = $pspec->get_name
  $self->{$pname} = $newval; # ready for default GET_PROPERTY

  if ($pname eq 'line_style') {
    $self->queue_draw;  # redraw with new lines
  }
}
```

Care must be taken with boxed non-reference-counted types such as `Gtk2::Gdk::Color`. In `SET_PROPERTY` the `$newval` is generally good only for the duration of the call. Use `copy` or similar if keeping it longer (see Glib::Boxed). In `GET_PROPERTY` the returned memory must last

long enough to reach the caller, which generally means returning a field, not a newly created object (which is destroyed with the scalar holding it).

`GET_PROPERTY` is different from a C get_property method in that the perl method returns the retrieved value. For symmetry, the `$newval` and `$pspec` args on `SET_PROPERTY` are swapped from the C usage.

FINALIZE_INSTANCE `$self` [not a method]

`FINALIZE_INSTANCE` is called as the GObject is being finalized, that is, as it is being really destroyed. This is independent of the more common DESTROY on the perl object; in fact, you must *NOT* override `DESTROY` (it's not useful to you, in any case, as it is being called multiple times!).

Use this hook to release anything you have to clean up manually. FINALIZE_INSTANCE will be called for each perl instance, in reverse order of construction.

The default finalizer does nothing.

`$object->`DESTROY [DO NOT OVERWRITE]

Don't *ever* overwrite `DESTROY`, use `FINALIZE_INSTANCE` instead.

The DESTROY method of all perl classes derived from GTypes is implemented in the Glib module and (ab−)used for its own internal purposes. Overwriting it is not useful as it will be called *multiple* times, and often long before the object actually gets destroyed. Overwriting might be very harmful to your program, so *never* do that. Especially watch out for other classes in your ISA tree.

PROPERTIES

To create gobject properties, supply a list of Glib::ParamSpec objects as the value for the key 'properties'. There are lots of different paramspec constructors, documented in the C API reference's Parameters and Values page, as well as Glib::ParamSpec.

As of Glib 1.060, you can also specify explicit getters and setters for your properties at creation time. The default values in your properties are also honored if you don't set anything else. See Glib::Type::register_object in Glib::Type for an example.

SIGNALS

Creating new signals for your new object is easy. Just provide a hash of signal names and signal descriptions under the key 'signals'. Each signal description is also a hash, with a few expected keys. All the keys are allowed to default.

flags => GSignalFlags

If not present, assumed to be 'run−first'.

param_types => reference to a list of package names

If not present, assumed to be empty (no parameters).

class_closure => reference to a subroutine to call as the class closure.

may also be a string interpreted as the name of a subroutine to call, but you should be very very very careful about that.

If not present, the library will attempt to call the method named ''do_signal_name'' for the signal ''signal_name'' (uses underscores).

You'll want to be careful not to let this handler method be a publically callable method, or one that has the name name as something that emits the signal. Due to the funky ways in which Glib is different from Perl, the class closures *should not* inherit through normal perl inheritance.

return_type => package name for return value.

If undefined or not present, the signal expects no return value. if defined, the signal is expected to return a value; flags must be set such that the signal does not run only first (at least use 'run−last').

accumulator => signal return value accumulator

quoting the Glib manual: ''The signal accumulator is a special callback function that can be used to collect return values of the various callbacks that are called during a signal emission.''

If not specified, the default accumulator is used, and you just get the return value of the last handler to run.

Accumulators are not really documented very much in the C reference, and the perl interface here is slightly different, so here's an inordinate amount of detail for this arcane feature:

The accumulator function is called for every handler as

```
($cont, $acc) = &$func ($invocation_hint, $acc, $ret)
```

`$invocation_hint` is an anonymous hash (including the signal name); `$acc` is the current accumulated return value; `$ret` is the value from the most recent handler.

The two return values are a boolean `$cont` for whether signal emission should continue (false to stop); and a new `$acc` accumulated return value. (This is different from the C version, which writes through a return_accu.)

OVERRIDING BASE METHODS

GLib pulls some fancy tricks with function pointers to implement methods in C. This is not very language-binding-friendly, as you might guess.

However, as described above, every signal allows a ''class closure''; you may override the class closure with your own function, and you can chain from the overridden method to the original. This serves to implement virtual overrides for language bindings.

So, to override a method, you supply a subroutine reference instead of a signal description hash as the value for the name of the existing signal in the ''signals'' hash described in ''SIGNALS''.

```
# override some important widget methods:
use Glib::Object::Subclass
      Gtk2::Widget::,
      signals => {
              expose_event => \&expose_event,
              configure_event => \&configure_event,
              button_press_event => \&button_press_event,
              button_release_event => \&button_release_event,
              motion_notify_event => \&motion_notify_event,
              # note the choice of names here... see the discussion.
              size_request => \&do_size_request,
      }
```

It's important to note that the handlers you supply for these are class-specific, and that normal perl method inheritance rules are not followed to invoke them from within the library. However, perl code can still find them! Therefore it's rather important that you choose your handlers' names carefully, avoiding any public interfaces that you might call from perl. Case in point, since size_request is a widget method, i chose do_size_request as the override handler.

INTERFACES

GObject supports only single inheritance; in place of multiple inheritance, GObject uses GInterfaces. In the Perl bindings we have mostly masqueraded this with multiple inheritance (that is, simply adding the GInterface class to the `@ISA` of the implementing class), but in deriving new objects the facade breaks and the magic leaks out.

In order to derive an object that implements a GInterface, you have to tell the GLib type system you want your class to include a GInterface. To do this, simply pass a list of package names through the ''interfaces'' key; this will add these packages to your `@ISA`, and cause perl to invoke methods that you must provide.

```
package Mup::MultilineEntry;
use Glib::Object::Subclass
    'Gtk2::TextView',
    interfaces => [ 'Gtk2::CellEditable' ],
    ;

# perl will now invoke these methods, which are part of the
# GtkCellEditable GInterface, when somebody invokes the
# corresponding lower-case methods on your objects.
sub START_EDITING { warn "start editing\n"; }
sub EDITING_DONE { warn "editing done\n"; }
sub REMOVE_WIDGET { warn "remove widget\n"; }
```

SEE ALSO

GObject - http://developer.gnome.org/doc/API/2.0/gobject/

AUTHORS

Marc Lehmann <schmorp@schmorp.de>, muppet <scott at asofyet dot org>

COPYRIGHT AND LICENSE