GTK+ 2.0 Tree View Tutorial using Ocaml

Tim-Philipp Muller

Ocaml adaptation: SooHyoung Oh

GTK+ 2.0 Tree View Tutorial using Ocaml

by Tim-Philipp Muller and Ocaml adaptation: SooHyoung Oh

This is a tutorial on how to use GTK+ 2.0 GtkTreeView widget in Ocaml language. Ocaml has a very nice Gtk+ interface called LablGtk in which GtkTreeView widget is implemented as GTree module.

The LablGtk version used here is LablGTK 2.4.0.

This tutorial is adapted from the original C language version written by Tim-Philipp Muller.

Please mail all comments and suggestions about Ocaml adaptation to `<shoh at compiler dot kaist dot ac dot kr>`

There is also a version in PDF format (for easier printing) and the raw docbook XML source document.

Some examples can be found in treeview_tutorial_examples.tar.gz tarball.

This tutorial is work-in-progress. The latest version can be found at http://compiler.kaist.ac.kr/~shoh/ocaml/lablgtk2/treev tutorial/index.html.

"Gtk+ 2.0 Tutorial using Ocaml" can be found at http://compiler.kaist.ac.kr/~shoh/ocaml/lablgtk2/lablgtk2-tutorial/index.html.

Last updated: 2004-09-18

Table of Contents

Chapter 1. Lists and Trees: the GTree Module

`GTree` is a module that displays single- or multi-columned lists and trees. It replaces the old Gtk+-1.2 GList and GTree modules. Even though `GTree` is slightly harder to master than its predecessors, it is so much more powerful and flexible that most application developers will not want to miss it once they have come to know it.

The purpose of this chapter is not to provide an exhaustive documentation of `GtkTreeView` - that is what the API documentation is for, which should be read alongside with this tutorial. The goal is rather to present an introduction to the most commonly-used aspects of `GTree (GtkTreeView)`, and to demonstrate how the various `GTree (GtkTreeView)` components and concepts work together. Furthermore, an attempt has been made to shed some light on custom tree models and custom cell renderers, which seem to be often-mentioned, but rarely explained.

Developers looking for a quick and dirty introduction that teaches them everything they need to know in less than five paragraphs will not find it here. In the author's experience, developers who do not understand how the tree view and the models work together will run into problems once they try to modify the given examples, whereas developers who have worked with other toolkits that employ the Model/View/Controller-design will find that the API reference provides all the information they need to know in more condensed form anyway. Those who disagree may jump straight to the working example code of course.

Please note that the code examples in the following sections do not necessarily demonstrate how `GTree` is used best in a particular situation. There are different ways to achieve the same result, and the examples merely show those different ways, so that developers are able to decide which one is most suitable for the task at hand.

1.1. Hello World

For the impatient, here is a small treeview 'Hello World' program.

```
(* file: helloworld.ml *)
(* Compile with:
 ocamlc -I +lablgtk2 -o helloworld lablgtk.cma gtkInit.cmo helloworld.ml
*)

open Gobject.Data

let cols = new GTree.column_list
let col_name = cols#add string (* string column *)
let col_age = cols#add int (* int column *)

let create_model () =
  let data = [("Heinz El-Mann", 51); ("Jane Doe", 23); ("Joe Bungop", 91)] in
  let store = GTree.list_store cols in
  let fill (name, age) =
    let iter = store#append () in
    store#set ~row:iter ~column:col_name name;
    store#set ~row:iter ~column:col_age age
  in
  List.iter fill data;
  store

let create_view ~model ~packing () =
  let view = GTree.view ~model ~packing () in

  (* Column #1: col_name is string column *)
  let col = GTree.view_column ~title:"Name"
      ~renderer:(GTree.cell_renderer_text [], ["text", col_name]) () in
  ignore (view#append_column col);

  (* Column #2: col_age is int column *)
  let col = GTree.view_column ~title:"Age"
      ~renderer:(GTree.cell_renderer_text [], ["text", col_age]) () in
  ignore (view#append_column col);

  view

let main () =
  let window = GWindow.window ~title:"GTree Demo" () in
  window#connect#destroy ~callback:GMain.Main.quit;
  let model = create_model () in
  create_view ~model ~packing:window#add ();
  window#show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

Chapter 2. Components: Model, Renderer, Column, View

The most important concept underlying `GTree` is that of complete separation between data and how that data is displayed on the screen. This is commonly known as Model/View/Controller-design (MVC). Data of various type (strings, numbers, images, etc.) is stored in a 'model'. The 'view' is then told which data to display, where to display it, and how to display it. One of the advantages of this approach is that you can have multiple views that display the same data (a directory tree for example) in different ways, or in the same way multiple times, with only one copy of the underlying data. This avoids duplication of data and programming effort if the same data is re-used in different contexts. Also, when the data in the model is updated, all views automatically get updated as well.

So, while `GTree.model` is used to store data, there are other components that determine which data is displayed in the `GTree.view` and how it is displayed. These components are `GTree.view_column` and `GTree.cell_renderer_*`. A `GTree.view` is made up of tree view columns. These are the columns that users perceive as columns. They have a clickable column header with a column title that can be hidden, and can be resized and sorted. Tree view columns do not display any data, they are only used as a device to represent the user-side of the tree view (sorting etc.) and serve as packing widgets for the components that do the actual rendering of data onto the screen, namely the `GTree.cell_renderer*` family of objects (I call them 'objects' because they are not widgets). There are a number of different cell renderers that specialise in rendering certain data like strings, pixbufs, or toggle buttons. More on this later.

Cell renderers are packed into tree view columns to display data. A tree view column needs to contain at least one cell renderer, but can contain multiple cell renderers. For example, if one wanted to display a 'Filename' column where each filename has a little icon on the left indicating the file type, one would pack a `GTree.cell_renderer_pixbuf` and a `GTree.cell_renderer_text` into one tree view column. Packing renderers into a tree view column is similar to packing widgets into a horizontal `GPack.box`.

Chapter 3. GTree.models for Data Storage: GTree.list_store and GTree.tree_store

It is important to realise what `GTree.model` (`GtkTreeModel`) is and what it is not. `GTree.model` is basically just an 'interface' to the data store, meaning that it is a standardised set of functions that allows a `GTree.view` widget (and the application programmer) to query certain characteristics of a data store, for example how many rows there are, which rows have children, and how many children a particular row has. It also provides functions to retrieve data from the data store, and tell the tree view what type of data is stored in the model. Every data store must inherit the `GTree.model` class and provide these functions. `GTree.model` itself only provides a way to query a data store's characteristics and to retrieve existing data, it does not provide a way to remove or add rows to the store or put data into the store. This is done using the specific store's functions.

GTree module comes with two built-in data stores (models): `GTree.list_store` (`GtkListStore`) and `GTree.tree_store` (`GtkTreeStore`). As the names imply, `GTree.list_store` is used for simple lists of data items where items have no hierarchical parent-child relationships, and `GTree.tree_store` is used for tree-like data structures, where items can have parent-child relationships. A list of files in a directory would be an example of a simple list structure, whereas a directory tree is an example for a tree structure. A list is basically just a special case of a tree with none of the items having any children, so one could use a tree store to maintain a simple list of items as well. The only reason `GTree.list_store` exists is in order to provide an easier interface that does not need to cater for child-parent relationships, and because a simple list model can be optimised for the special case where no children exist, which makes it faster and more efficient.

`GTree.list_store` and `GTree.tree_store` should cater for most types of data an application developer might want to display in a `GTree.view`. However, it should be noted that `GTree.list_store` and `GTree.tree_store` have been designed with flexibility in mind. If you plan to store a lot of data, or have a large number of rows, you should consider implementing your own custom model that stores and manipulates data your own way and implements the inherited class from `GTree.model` class. This will not only be more efficient, but probably also lead to saner code in the long run, and give you more control over your data. See below for more details on how to implement custom models.

Tree model implementations like `GTree.list_store` and `GTree.tree_store` will take care of the view side for you once you have configured the `GTree.view` to display what you want. If you change data in the store, the model will notify the tree view and your data display will be updated. If you add or remove rows, the model will also notify the store, and your row will appear in or disappear from the view as well.

3.1. How Data is Organised in a Store

A model (data store) has model columns and rows. While a tree view will display each row in the model as a row in the view, the model's columns are not to be confused with a view's columns. A model column represents a certain data field of an item that has a fixed data type. You need to know what kind of data you want to store when you create a list store or a tree store, as you can not add new fields later on.

For example, we might want to display a list of files. We would create a list store with two fields: a field that stores the filename (ie. a string) and a field that stores the file size (ie. an unsigned integer). The filename would be stored in column 0 of the model, and the file size would be stored in column 1 of the model. For each file we would add a row to the list store, and set the row's fields to the filename and the file size.

The Gobject.Data(GLib type system (GType)) is used to indicate what type of data is stored in a model column. These are the most commonly used types:

- `Gobject.Data.boolean`
- `Gobject.Data.int`, `Gobject.Data.uint`
- `Gobject.Data.long`, `Gobject.Data.ulong`, `Gobject.Data.int64`, `Gobject.Data.uint64` (these are not supported in early gtk+-2.0.x versions)
- `Gobject.Data.float`, `Gobject.Data.double`
- `Gobject.Data.string` - stores a string in the store (makes a copy of the original string)
- `Gobject.Data.pointer` - stores a pointer value (does not copy any data into the store, just stores the pointer value!)
- `Gobject.Data.gobject` - stores gobject in the store. For example, the type `GdkPixbuf.pixbuf` is gobject.

You do not need to understand the type system, it will usually suffice to know the above types, so you can tell a list store or tree store what kind of data you want to store.

Here is an example of how to create a list store:

```
let cols = new GTree.column_list in
let str_col = cols#add Gobject.Data.string in
let uint_col = cols#add Gobject.Data.uint in
let list_store = GTree.list_store cols in
...
```

This creates a new list store with two columns. Column 0 stores a string and column 1 stores an unsigned integer for each row. At this point the model has no rows yet of course. Before we start to add rows, let’s have a look at the different ways used to refer to a particular row.

3.2. Refering to Rows: Gtk.tree_iter, Gtk.tree_path, Gtk.row_reference, GTree.row_reference

There are different ways to refer to a specific row. The two you will have to deal with are `Gtk.tree_iter` and `Gtk.tree_path`.

3.2.1. Gtk.tree_path

A `Gtk.tree_path` is a comparatively straight-forward way to describe the logical position of a row in the model. As a `GTree.view` always displays *all* rows in a model, a tree path always describes the same row in both model and view.

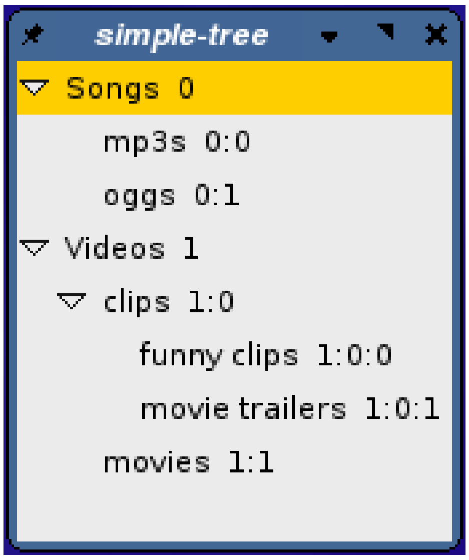


Figure 3-1. Tree Paths

The picture shows the tree path in string form next to the label. Basically, it just counts the children from the imaginary root of the tree view. An empty tree path string would specify that imaginary invisible root. Now ’Songs’ is the first child (from the root) and thus its tree path is just "0". ’Videos’ is the second child from the root, and its tree path is "1". ’oggs’ is the second child of the first item from the root, so its tree path is "0:1". So you just count your way down from the root to the row in question, and you get your tree path.

To clarify this, a tree path of "3:9:4:1" would basically mean *in human language* (attention - this is not what it really means!) something along the lines of: go to the 3rd top-level row. Now go to the 9th child of that row. Proceed to the 4th child of the previous row. Then continue to the 1st child of that. Now you are at the row this tree path describes. This is not what it means for Gtk+ though. While humans start counting at 1, computers usually start counting at 0. So the real meaning of the tree path "3:9:4:1" is: Go to the 4th top-level row. Then go to the 10th child of that row. Then pick the 4th child of the last row. Continue to the 2nd child of the previous row. Now you are at the row this tree path describes. :)

The implication of this way of refering to rows is as follows: if you insert or delete rows in the middle or the rows are resorted, a tree path might suddenly refer to a completely different row than it refered to before the insertion/deletion/resorting. This is important to keep in mind. (See the section on `Gtk.row_references` below for a tree path that keeps updating itself to make sure it always refers to the same row when the model changes).

This effect becomes apparent if you imagine what would happen if we were to delete the row entitled 'funny clips' from the tree in the above picture. The row 'movie clips' would suddenly be the first and only child of 'clips', and be described by the tree path that formerly belonged to 'movie clips', ie. "1:0:0".

You can get a new `Gtk.tree_path` from a path in string form using `GTree.Path.from_string`, and you can convert a given `Gtk.tree_path` into its string notation with `GTree.Path.to_string`. Usually you will rarely have to handle the string notation, it is described here merely to demonstrate the concept of tree paths.

Instead of the string notation, `Gtk.tree_path` uses an integer array internally. You can get the depth (ie. the nesting level) of a tree path with `GTree.Path.get_depth` (`gtk_tree_path_get_depth`). A depth of 0 is the imaginary invisible root node of the tree view and model. A depth of 1 means that the tree path describes a top-level row. As lists are just trees without child nodes, all rows in a list always have tree paths of depth 1. `GTree.Path.get_indices` (`gtk_tree_path_get_indices`) returns the internal integer array of a tree path. You will rarely need to operate with those either.

If you operate with tree paths, you are most likely to use a given tree path, and use functions like `GTree.Path.up` (`gtk_tree_path_up`), `GTree.Path.down` (`gtk_tree_path_down`), `GTree.Path.next` (`gtk_tree_path_next`), `GTree.Path.prev` (`gtk_tree_path_prev`), or `GTree.Path.is_ancestor` (`gtk_tree_path_is_ancestor`). Note that this way you can construct and operate on tree paths that refer to rows that do not exist in model or view! The only way to check whether a path is valid for a specific model (ie. the row described by the path exists) is to convert the path into an iter using `GTree.model#get_iter` (`gtk_tree_model_get_iter`).

`Gtk.tree_path` is an opaque structure, with its details hidden from the compiler. If you need to make a copy of a tree path, use `GTree.Path.copy` (`gtk_tree_path_copy`).

3.2.2. Gtk.tree_iter

Another way to refer to a row in a list or tree is `Gtk.tree_iter`. A tree iter is just a structure that contains a couple of pointers that mean something to the model you are using. Tree iters are used internally by models, and they often contain a direct pointer to the internal data of the row in question. You should never look at the content of a tree iter and you must not modify it directly either.

All tree models (and therefore also `GTree.list_store` and `GTree.tree_store`) must support the `GTree.model` (`GtkTreeModel`) functions that operate on tree iters (e.g. get the tree iter for the first child of the row specified by a given tree iter, get the first row in the list/tree, get the n-th child of a given iter etc.). Some of these functions are:

- `GTree.model#get_iter_first` (`gtk_tree_model_get_iter_first`) - sets the given iter to the first top-level item in the list or tree
- `GTree.model#iter_next` (`gtk_tree_model_iter_next`) - sets the given iter to the next item at the current level in a list or tree.
- `GTree.model#iter_children` (`gtk_tree_model_iter_children`) - sets the first given iter to the first child of the row referenced by the second iter (not very useful for lists, mostly useful for trees).
- `GTree.model#iter_n_children` (`gtk_tree_model_iter_n_children`) - returns the number of children the row referenced by the provided iter has. If you pass `NULL` instead of a pointer to an iter structure, this function will return the number of top-level rows. You can also use this function to count the number of items in a list store.
- `GTree.model#iter_children` (`gtk_tree_model_iter_nth_child`) - sets the first iter to the n-th child of the row referenced by the second iter. If you pass `NULL` instead of a pointer to an iter structure as the second iter, you can get the first iter set to the n-th row of a list.
- `GTree.model#iter_parent` (`gtk_tree_model_iter_parent`) - returns the iter to the parent of the row referenced by the given iter (does nothing for lists, only useful for trees).

There are more functions that operate on iters. Check out the `GTree.model` (`GtkTreeModel` API reference) for details.

You might notice that there is no `GTree.model#iter_prev`. This is unlikely to be implemented for a variety of reasons. It should be fairly simple to write a helper function that provides this functionality though once you have read this section.

Tree iters are used to retrieve data from the store, and to put data into the store. You also get a tree iter as result if you add a new row to the store using `GTree.list_store#append` or `GTree.tree_store#append`.

Tree iters are often only valid for a short time, and might become invalid if the store changes with some models. There is a better way to keep track of a row over time: `Gtk.row_reference`

3.2.3. Gtk.row_reference, GTree.row_reference

A `Gtk.row_reference` is basically an object that takes a tree path, and watches a model for changes. If anything changes, like rows getting inserted or removed, or rows getting re-ordered, the tree row reference object will keep the given tree path up to date, so that it always points to the same row as before. In case the given row is removed, the tree row reference will become invalid.

`GTree.row_reference` class is the wrapper class of `Gtk.row_reference`.

A new tree row reference `GTree.row_reference` can be created with `GTree.model#get_row_reference` (`gtk_tree_row_reference_new`), given a model and a tree path. After that, the tree row reference will keep updating the path whenever the model changes. The current tree path of the row originally refered to when the tree row reference was created can be retrieved with `GTree.row_reference#path` (`gtk_tree_row_reference_get_path`).

You can check whether the row referenced still exists with `GTree.row_reference#valid` (`gtk_tree_row_reference_valid`).

For the curious: internally, the tree row reference connects to the tree model's `"row-inserted"`, `"row-deleted"`, and `"rows-reordered"` signals and updates its internal tree path whenever something happened to the model that affects the position of the referenced row.

Note that using tree row references entails a small overhead. This is hardly significant for 99.9% of all applications out there, but when you have multiple thousands of rows and/or row references, this might be something to keep in mind (because whenever rows are inserted, removed, or reordered, a signal will be sent out and processed for each row reference).

If you have read the tutorial only up to here so far, it is hard to explain really what tree row references are good for. An example where tree row references come in handy can be found further below in the section on removing multiple rows in one go.

In practice, a programmer can either use tree row references to keep track of rows over time, or store tree iters directly (if, and only if, the model has persistent iters). Both `GTree.list_store` and `GTree.tree_store` have persistent iters, so storing iters is possible. However, using tree row references is definitively the Right Way(tm) to do things, even though it comes with some overhead that might impact performance in case of trees that have a very large number of rows (in that case it might be preferable to write a custom model anyway though). Especially beginners might find it easier to handle and store tree row references than iters, because tree row references are handled by pointer value, which you can easily add to a `GList` or pointer array, while it is easy to store tree iters in a wrong way.

3.2.4. Usage

Tree iters can easily be converted into tree paths using `GTree.model#get_path` (`gtk_tree_model_get_path`), and tree paths can easily be converted into tree iters using `GTree.model#get_iter` (`gtk_tree_model_get_iter`). Here is an example that shows how to get the iter from the tree path that is passed to us from the tree view in the `"row-activated"` signal callback. We need the iter here to retrieve data from the store

```
(************************************************************
 *                                                          *
 * Converting a GtkTreePath into a GtkTreeIter              *
 *                                                          *
 ************************************************************)

(************************************************************
 *
 * on_row_activated: a row has been double-clicked
 *
 ************************************************************)
let on_row_activated view path col =
  let model = view#model in
  let iter = model#get_iter path in
  let name = model#get iter col_name in
  Printf.printf "The row containing the name ’%s’ has been double-clicked.\n" name
```

Tree row references reveal the current path of a row with `gtk_tree_row_reference_get_path`. There is no direct way to get a tree iter from a tree row reference, you have to retrieve the tree row reference's path first and then convert that into a tree iter.

As tree iters are only valid for a short time, they are usually allocated on the stack, as in the following example (keep in mind that `Gtk.tree_iter` is just a structure that contains data fields you do not need to know anything about):

```
(**********************************************************
 *                                                        *
 *  Going through every row in a list store               *
 *                                                        *
 **********************************************************)

 let traverse_list_store liststore =
   (* Get first row in list store *)
   let first = liststore#get_iter_first in
   match first with
   | Some iter ->
      (* ... do something with that row using the iter ...          *)
      (* Here set the col_first_name column *)
      liststore#set ~row:iter ~column:col_first_name "Joe";

      (* Make iter point to the next row in the list store *)
      while liststore#iter_next iter do
        (* Again, do something with that row using the iter ...          *)
        (* Again, here set the col_first_name column *)
        liststore#set ~row:iter ~column:col_first_name "Jane";
      done
   | None -> ()
```

The code above asks the model to fill the iter structure to make it point to the first row in the list store. If there is a first row and the list store is not empty, the iter will be set, and `GTree.model#get_iter_first` (`gtk_tree_model_get_iter_first`) will return `Some _`. If there is no first row, it will just return `None`. If a first row exists, the while loop will be entered and we change some of the first row's data. Then we ask the model to make the given iter point to the next row, until there are no more rows, which is when `GTree.model#iter_next` (`gtk_tree_model_iter_next`) returns `false`. Instead of traversing the list store we could also have used `GTree.model#foreach` (`gtk_tree_model_foreach`).

3.3. Adding Rows to a Store

3.3.1. Adding Rows to a List Store

Rows are added to a list store with `GTree.list_store#append` (`gtk_list_store_append`). This will insert a new empty row at the end of the list. There are other functions, documented in the `GTree.list_store` (GtkListStore API reference), that give you more control about where exactly the new row is inserted, but as they work very similar to `GTree.list_store#append` and are fairly straight-forward to use, we will not deal with them here.

Here is a simple example of how to create a list store and add some (empty) rows to it:

```
 let cols = new GTree.column_list in
 let col_name = cols#add Gobject.Data.string in
 let liststore = GTree.list_store cols in

 (* Append an empty row to the list store. Iter will point to the new row *)
 let iter = liststore#append () in

 (* Append an empty row to the list store. Iter will point to the new row *)
 let iter = liststore#append () in

 (* Append an empty row to the list store. Iter will point to the new row *)
 let iter = liststore#append () in
```

This in itself is not very useful yet of course. We will add data to the rows in the next section.

3.3.2. Adding Rows to a Tree Store

Adding rows to a tree store works similar to adding rows to a list store, only that `GTree.tree_store#append` (`gtk_tree_store_append`) is the function to use and one more argument is required, namely the tree iter to the parent of the row to insert. If you do NOT supply instead of providing the tree iter of another row, a new top-level row will be inserted. If you do provide a parent tree iter, the new empty row will be inserted after any already existing children of the parent. Again, there are other ways to insert a row into the tree store and they are documented in the `GTree.tree_store` (GtkTreeStore API reference manual). Another short example:

```
let cols = new GTree.column_list in
let col_name = cols#add Gobject.Data.string in
let treestore = GTree.tree_store cols in

(* Append an empty top-level row to the tree store.
 *  Iter will point to the new row *)
let iter = treestore#append () in

(* Append another empty top-level row to the tree store.
 *  Iter will point to the new row *)
let iter = treestore#append () in

(* Append a child to the row we just added.
 *  Child will point to the new row *)
let child = treestore#append ~parent:iter () in

(* Get the first row, and add a child to it as well (could have been done
 *  right away earlier of course, this is just for demonstration purposes) *)
match treestore#get_iter_first with
| Some iter ->
    (* Child will point to new row *)
    let child = treestore#append ~parent:iter () in
    ...
| None ->
    prerr_endline "Oops, we should have a first row in the tree store!"
```

3.3.3. Speed Issues when Adding a Lot of Rows

A common scenario is that a model needs to be filled with a lot of rows at some point, either at start-up, or when some file is opened. An equally common scenario is that this takes an awfully long time even on powerful machines once the model contains more than a couple of thousand rows, with an exponentially decreasing rate of insertion. As already pointed out above, writing a custom model might be the best thing to do in this case. Nevertheless, there are some things you can do to work around this problem and speed things up a bit even with the stock Gtk+ models:

Firstly, you should detach your list store or tree store from the tree view before doing your mass insertions, then do your insertions, and only connect your store to the tree view again when you are done with your insertions. Like this:

```
...
let model = view#model in
view#set_model None;  (* Detach model from view *)
... (* insert a couple of thousand rows *) ...
view#set_model (Some model); (* Re-attach model to view *)
...
```

Secondly, you should make sure that sorting is disabled while you are doing your mass insertions, otherwise your store might be resorted after each and every single row insertion, which is going to be everything but fast.

Thirdly, you should not keep around a lot of tree row references if you have so many rows, because with each insertion (or removal) every single tree row reference will check whether its path needs to be updated or not.

3.4. Manipulating Row Data

Adding empty rows to a data store is not terribly exciting, so let's see how we can add or change data in the store.

`GTree.list_store#set` (`gtk_list_store_set`) and `GTree.tree_store#set` (`gtk_tree_store_set`) are used to manipulate a given row's data. Both `GTree.list_store#set` and `GTree.tree_store#set` take three arguments. The first argument is the iter pointing to the row whose data we want to change. The second argument, the column, refers to the model column which we want to change. The third argument, the data, should be of the same data type as the model column.

Here is an example where we create a store that stores two strings and one integer for each row:

```
let cols = new GTree.column_list in
let col_first_name = cols#add Gobject.Data.string in
let col_last_name = cols#add Gobject.Data.string in
let col_year_born = cols#add Gobject.Data.int in
let liststore = GTree.list_store cols in

(* Append an empty row to the list store. Iter will point to the new row *)
let row = liststore#append () in

(* Fill fields with some data *)
liststore#set ~row ~column:col_first_name "Joe";
liststore#set ~row ~column:col_last_name "Average";
liststore#set ~row ~column:col_year_born 1970;
```

3.5. Retrieving Row Data

Storing data is not very useful if it cannot be retrieved again. This is done using `get` method (`gtk_tree_model_get`), which takes similar arguments as `gtk_list_store_set` or `gtk_tree_store_set` do, only that it takes iter and column as arguments and returns the value of the column. The return value is of the same type as the data stored in the particular model column.

Here is the previous example extended to traverse the list store and print out the data stored. As an extra, we use `gtk_tree_model_foreach` to traverse the store and retrieve the row number from the `GtkTreePath` passed to us in the foreach callback function:

```
let cols = new GTree.column_list
let col_first_name = cols#add Gobject.Data.string
let col_last_name = cols#add Gobject.Data.string
let col_year_born = cols#add Gobject.Data.int

let foreach_fun model path iter =
  let first_name = model#get iter col_first_name in
  let last_name = model#get iter col_last_name in
  let year_of_birth = model#get iter col_year_born in

  let tree_path_str = GTree.Path.to_string path in
  Printf.printf "Row %s: %s %s, born %u\n" tree_path_str
    first_name last_name year_of_birth;

  false (* do not stop walking the store, call us with next row *)

let create_and_fill_and_dump_store () =
  let liststore = GTree.list_store cols in

  (* Append an empty row to the list store. Iter will point to the new row *)
  let row = liststore#append () in

  (* Fill fields with some data *)
  liststore#set ~row ~column:col_first_name "Joe";
  liststore#set ~row ~column:col_last_name "Average";
  liststore#set ~row ~column:col_year_born 1970;

  (* Append another row, and fill in some data *)
  let row = liststore#append () in

  liststore#set ~row ~column:col_first_name "Jane";
  liststore#set ~row ~column:col_last_name "Common";
```

```
    liststore#set ~row ~column:col_year_born 1967;

    (* Append yet another row, and fill it *)
    let row = liststore#append () in

    liststore#set ~row ~column:col_first_name "Yo";
    liststore#set ~row ~column:col_last_name "Da";
    liststore#set ~row ~column:col_year_born 1873;

    (* Now traverse the list *)

    liststore#foreach (foreach_func liststore)

  let main () = create_and_fill_and_dump_store ()

  let _ = Printexc.print main ()
```

3.6. Removing Rows

Rows can easily be removed with `GTree.list_store#remove` (`gtk_list_store_remove`) and `GTree.tree_store#remove` (`gtk_tree_store_remove`). The removed row will automatically be removed from the tree view as well.

Removing a single row is fairly straight forward: you need to get the iter that identifies the row you want to remove, and then use one of the above functions. Here is a simple example that removes a row when you double-click on it (bad from a user interface point of view, but then it is just an example):

```
  let on_row_activated view path col =
    print_endline "Row has been double-clicked. Removing row.";

    let model = view#model in

    let iter = model#get_iter path in
    model#remove iter; (* It returns bool *)
    ()   (* We have to return () in callback. *)

  let create_treeview () =
    ...
    treeview#connect#row_activated ~callback:(on_row_activated treeview);
    ...
```

Note: `GTree.list_store#remove` (`gtk_list_store_remove`) and `GTree.tree_store#remove` (`gtk_tree_store_remove`) both have slightly different semantics in Gtk+-2.0 and Gtk+-2.2 and later. In Gtk+-2.0, both functions do not return a value, while in later Gtk+ versions those functions return either `true` or `false` to indicate whether the iter given has been set to the next valid row (or invalidated if there is no next row). (The current LablGTK2 supports Gtk+-2.4 and `remove` method returns `true` or `false`.) This is important to keep in mind when writing code that is supposed to work with all Gtk+-2.x versions. In that case you should just ignore the value returned (as in the call above) and check the iter with `GTree.list_store#iter_is_valid` (`gtk_list_store_iter_is_valid`) if you need it.

If you want to remove the n-th row from a list (or the n-th child of a tree node), you have two approaches: either you first create a `GtkTreePath` that describes that row and then turn it into an iter and remove it; or you take the iter of the parent node and use `iter_children` method (`gtk_tree_model_iter_nth_child`) (which will also work for list stores if you use `None` as the parent iter). Of course you could also start with the iter of the first top-level row, and then step-by-step move it to the row you want, although that seems a rather awkward way of doing it.

The following code snippet will remove the n-th row of a list if it exists:

```
   (*******************************************************************
    *
    *  list_store_remove_nth_row
    *
    *  Removes the nth row of a list store if it exists.
    *
    *  Returns true on success or false if the row does not exist.
```

```
  *
  *****************************************************************)

 let list_store_remove_nth_row store n =
   try
     let iter = store#iter_children ~nth:n None in
     store#remove iter;
     true
   with Invalid_arg _ -> false
```

3.7. Removing Multiple Rows

Removing multiple rows at once can be a bit tricky at times, and requires some thought on how to do this best. For example, it is not possible to traverse a store with `foreach` method, check in the callback function whether the given row should be removed and then just remove it by calling one of the stores' remove functions. This will not work, because the model is changed from within the foreach loop, which might suddenly invalidate formerly valid tree iters in the foreach function, and thus lead to unpredictable results.

You could traverse the store in a `while` loop of course, and call `list_store#remove` or `tree_store#remove` whenever you want to remove a row, and then just continue if the remove functions returns `true` (meaning that the iter is still valid and now points to the row after the row that was removed). However, this approach will only work with Gtk+-2.2 or later and will not work if you want your programs to compile and work with Gtk+-2.0 as well, for the reasons outlined above (in Gtk+-2.0 the remove functions did not set the passed iter to the next valid row). Also, while this approach might be feasable for a list store, it gets a bit awkward for a tree store. (The current LablGTK2 supports Gtk+-2.4 and `remove` method returns `true` or `false`.)

Here is an example for an alternative approach to removing multiple rows in one go (here we want to remove all rows from the store that contain persons that have been born after 1980, but it could just as well be all selected rows or some other criterion):

```
(*****************************************************************
 *
 *  Removing multiple rows in one go
 *
 *****************************************************************)

 ...

 let filter store pred =
   let rr_list = ref [] in (* list of GTree.row_reference to remove *)
   let foreach_func path iter =
     if pred iter
     then rr_list := store#get_row_reference path :: !rr_list;
     false (* do not stop walking the store, call us with next row *)
   in
   store#foreach foreach_func;
   !rr_list

 let remove_people_born_after_1980 store =
   let born_after_1980 iter =
     let year_of_birth = store#get iter col_year_born in
     year_of_birth > 1980
   in
   let r_list = filter store born_after_1980 in
   let remove reference =
     let iter = reference#iter in
     store#remove iter; (* This returns bool *)
     ()
   in
   List.iter remove r_list

 ...
```

`GTree.list_store#clear` (`gtk_list_store_clear`) and `GTree.tree_store#clear` (`gtk_tree_store_clear`) come in handy if you want to remove all rows.

Chapter 4. Creating a Tree View

In order to display data in a tree view widget, we need to create one first, and we need to instruct it where to get the data to display from.

A new tree view is created with:

```
let view = GTree.view () in
...
```

4.1. Connecting Tree View and Model

Before we proceed to the next section where we display data on the screen, we need connect our data store to the tree view, so it knows where to get the data to display from. This is achieved with `GTree.view#set_model` (`gtk_tree_view_set_model`), which will by itself do very little. However, it is a prerequisite for what we do in the following sections. `GTree.view` with ~`model` option (`gtk_tree_view_new_with_model`) is a convenience function for the previous two.

`GTree.view#model` (`gtk_tree_view_get_model`) will return the model that is currently attached to a given tree view, which is particularly useful in callbacks where you only get passed the tree view widget (after all, we do not want to go down the road of global variables, which will inevitably lead to the Dark Side, do we?).

4.2. Tree View Look and Feel

There are a couple of ways to influence the look and feel of the tree view. You can hide or show column headers with `GTree.view#set_headers_visible` (`gtk_tree_view_set_headers_visible`), and set them clickable or not with `GTree.view#set_headers_clickable` (`gtk_tree_view_set_headers_clickable`) (which will be done automatically for you if you enable sorting).

`GTree.view#set_rules_hint` (`gtk_tree_view_set_rules_hint`) will enable or disable rules in the tree view. [1] As the function name implies, this setting is only a hint; in the end it depends on the active Gtk+ theme engine if the tree view shows ruled lines or not. Users seem to have strong feelings about rules in tree views, so it is probably a good idea to provide an option somewhere to disable rule hinting if you set it on tree views (but then, people also seem to have strong feelings about options abundance and 'sensible' default options, so whatever you do will probably upset someone at some point).

The expander column can be set with `GTree.view#set_expander_column` (`gtk_tree_view_set_expander_column`). This is the column where child elements are indented with respect to their parents, and where rows with children have an 'expander' arrow with which a node's children can be collapsed (hidden) or expanded (shown). By default, this is the first column.

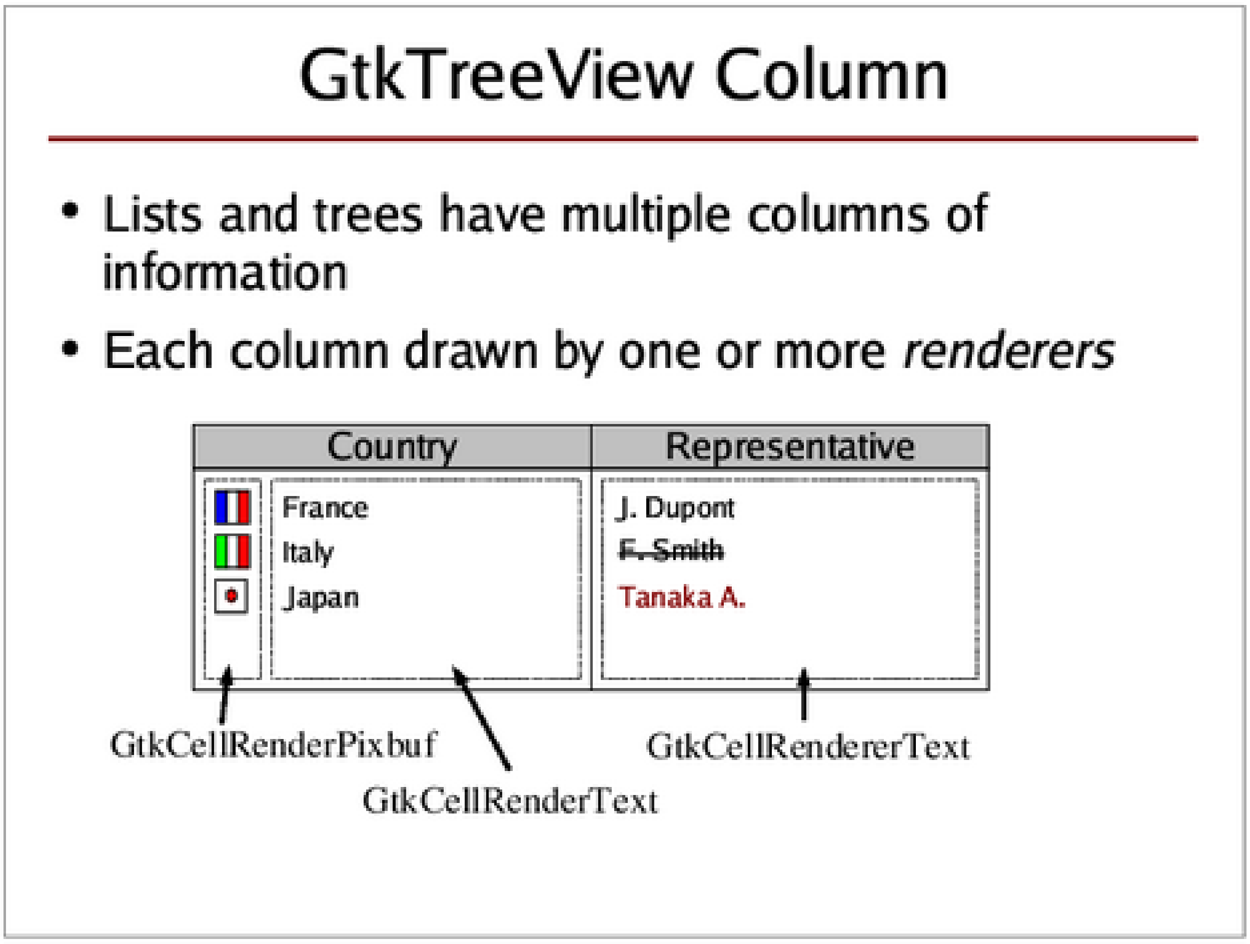
Notes

1. 'Rules' means that every second line of the tree view has a shaded background, which makes it easier to see which cell belongs to which row in tree views that have a lot of columns.

Chapter 5. Mapping Data to the Screen: GTree.view_column and GTree.cell_renderer_*

As outlined above, tree view columns represent the visible columns on the screen that have a column header with a column name and can be resized or sorted. A tree view is made up of tree view columns, and you need at least one tree view column in order to display something in the tree view. Tree view columns, however, do not display anything by themselves, this is done by specialised `GTree.cell_renderer_*` objects. Cell renderers are packed into tree view columns much like widgets are packed into horizontal `GBox.box`.

Here is a diagram (courtesy of Owen Taylor) that pictures the relationship between tree view columns and cell renderers:



GtkTreeView Column

- Lists and trees have multiple columns of information
- Each column drawn by one or more *renderers*

Figure 5-1. Cell Renderer Properties

In the above diagram, both ’Country’ and ’Representative’ are tree view columns, where the ’Country’ and ’Representative’ labels are the column headers. The ’Country’ column contains two cell renderers, one to display the flag icons, and one to display the country name. The ’Representative’ column only contains one cell renderer to display the representative’s name.

5.1. Cell Renderers

Cell renderers are objects that are responsible for the actual rendering of data within a `GTree.view_column`. They are basically just GObjects (ie. not widgets) that have certain properties, and those properties determine how a single cell is drawn.

In order to draw cells in different rows with different content, a cell renderer’s properties need to be set accordingly for each single row/cell to render. This is done either via attributes or cell data functions (see below). If you set up attributes, you tell Gtk which model column contains the data from which a property should be set before rendering a certain row. Then the properties of a cell renderer are set automatically according to the data in the model before each row is rendered. Alternatively, you can set up cell data functions, which are called for each row to be rendererd, so that you can manually set the properties of the cell renderer before it is rendered. Both approaches can be used at the same time as well. Lastly, you can set a cell renderer property when you create the cell renderer. That way it will be used for all rows/cells to be rendered (unless it is changed later of course).

Different cell renderers exist for different purposes:

- `GTree.cell_renderer_text` (`GtkCellRendererText`) renders strings or numbers or boolean values as text ("Joe", "99.32", "true")
- `GTree.cell_renderer_pixbuf` (`GtkCellRendererPixbuf`) is used to display images; either user-defined images, or one of the stock icons that come with Gtk+.
- `GTree.cell_renderer_toggle` (`GtkCellRendererToggle`) displays a boolean value in form of a check box or as a radio button.

Contrary to what one may think, a cell renderer does not render just one single cell, but is responsible for rendering part or whole of a tree view column for each single row. It basically starts in the first row and renders its part of the column there. Then it proceeds to the next row and renders its part of the column there again. And so on.

How does a cell renderer know what to render? A cell renderer object has certain 'properties' that are documented in the `GTree.html` (API reference) (just like most other objects, and widgets). These properties determine what the cell renderer is going to render and how it is going to be rendered. Whenever the cell renderer is called upon to render a certain cell, it looks at its properties and renders the cell accordingly. This means that whenever you set a property or change a property of the cell renderer, this will affect all rows that are rendered after the change, until you change the property again.

Here is a diagram (courtesy of Owen Taylor) that tries to show what is going on when rows are rendered:

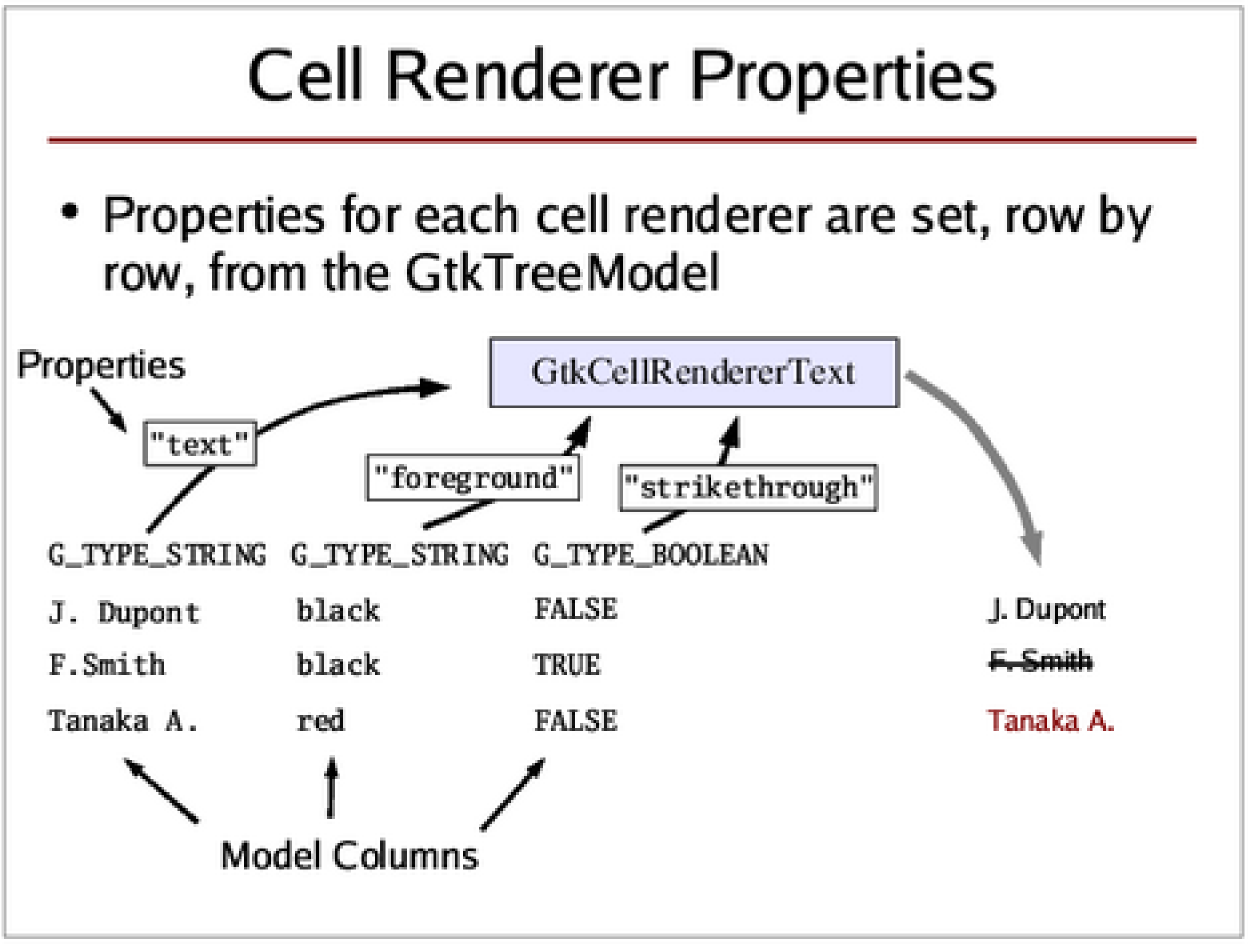


Figure 5-2. GtkTreeViewColumns and GtkCellRenderers

The above diagram shows the process when attributes are used. In the example, a text cell renderer's `"text"` property has been linked to the first model column. The `"text"` property contains the string to be rendered. The `"foreground"` property, which contains the colour of the text to be shown, has been linked to the second model column. Finally, the `"strikethrough"` property, which determines whether the text should be with a horizontal line that strikes through the text, has been connected to the third model column (of type `boolean(G_TYPE_BOOLEAN)`).

With this setup, the cell renderer's properties are 'loaded' from the model before each cell is rendered.

Here is a silly and utterly useless little example that demonstrates this behaviour, and introduces some of the most commonly used properties of `GTree.cell_renderer_text`:

```
(* file: cell_renderer.ml *)

open Gobject.Data

let cols = new GTree.column_list
let col_first_name = cols#add string
let col_last_name = cols#add string

let create_and_fill_model () =
  let treestore = GTree.tree_store cols in

  (* Append a top level row and leave it empty *)
  let toplevel = treestore#append () in

  (* Append a second top level row, and fill it with some data *)
  let toplevel = treestore#append () in
  treestore#set ~row:toplevel ~column:col_first_name "Joe";
  treestore#set ~row:toplevel ~column:col_last_name "Average";

  (* Append a child to the second top level row, and fill in some data *)
  let child = treestore#append ~parent:toplevel () in
  treestore#set ~row:child ~column:col_first_name "Jane";
  treestore#set ~row:child ~column:col_last_name "Average";

  treestore

let create_view_and_model ~packing () =
  let view = GTree.view ~packing () in

  (* Column #1 *)
  (* create text cell renderer
     and ‘TEXT property of the cell renderer *)
  let renderer_text = GTree.cell_renderer_text [‘TEXT "Boooo!"] in
  (* create tree view column and pack cell renderer into tree view column *)
  let col = GTree.view_column ~title:"First Name"
      ~renderer:(renderer_text, []) () in
  (* pack tree view column into tree view *)
  view#append_column col;

  (* Column #2 *)
  (* create text cell renderer
     and set ‘BACKGROUND property of the cell renderer *)
  let renderer_text = GTree.cell_renderer_text [
    ‘BACKGROUND "Orange";
    ‘BACKGROUND_SET true
  ] in
  (* create tree view column and pack cell renderer into tree view column *)
  let col = GTree.view_column ~title:"Last Name"
      ~renderer:(renderer_text, []) () in
  (* pack tree view column into tree view *)
  view#append_column col;

  let model = create_and_fill_model () in
  view#set_model (Some (model#coerce));
  view#selection#set_mode ‘NONE;

  view

let main () =
  let window = GWindow.window ~title:"Name" ~border_width:10 () in
  window#connect#destroy ~callback:GMain.Main.quit;
  create_view_and_model ~packing:window#add ();
  window#show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

The above code should produce something looking like this:

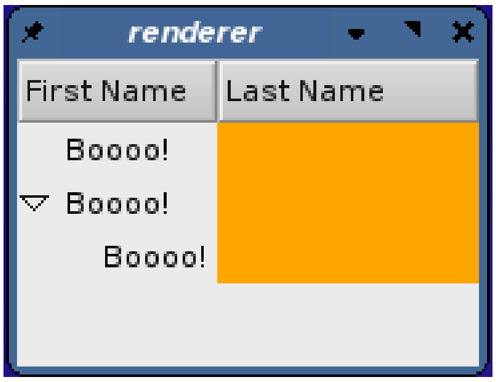


Figure 5-3. Persistent Cell Renderer Properties

It looks like the tree view display is partly correct and partly incomplete. On the one hand the tree view renders the correct number of rows (note how there is no orange on the right after row 3), and it displays the hierarchy correctly (on the left), but it does not display any of the data that we have stored in the model. This is because we have made no connection between what the cell renderers should render and the data in the model. We have simply set some cell renderer properties on start-up, and the cell renderers adhere to those set properties meticulously.

There are two different ways to connect cell renderers to data in the model: attributes and cell data functions.

5.2. Attributes

An attribute is a connection between a cell renderer property and a field/column in the model. Whenever a cell is to be rendered, a cell renderer property will be set to the values of the specified model column of the row that is to be rendered. It is very important that the column's data type is the same type that a property takes according to the API reference manual. Here is some code to look at:

```
  ...
  let renderer_text = GTree.cell_renderer_text [] in
  let col = GTree.view_column
      ~renderer:(renderer_text, ["text", col_first_name]) () in
  ...
```

This means that the text cell renderer property `"text"` will be set to the string in model column `col_first_name` of each row to be drawn. (The property "text" is called as `‘TEXT` property in Ocaml-style.) Cell properties can be set using one of `GTree.cell_renderer_*`, `GTree.cell_renderer_*#set_properties`, `GTree.view_column`, or `GTree.view_column#add_attribute` (`gtk_tree_view_column_add_attribute`). These functions set properties to whatever is in the specified _model column_ at the time of rendering.

`GTree.cell_renderer_*` are types of:

```
val GTree.cell_renderer_pixbuf : cell_properties_pixbuf list -> cell_renderer_pixbuf
val GTree.cell_renderer_text : cell_properties_text list -> cell_renderer_text
val GTree.cell_renderer_toggle : cell_properties_toggle list -> cell_renderer_toggle
```

so you can give the list of properties as their argument. After creating `GTree.cell_renderer_*`, you can use:

```
method GTree.cell_renderer_xxx#set_properties : ’b list -> unit
```

(*cell_renderer_xxx* is one of `cell_renderer_text`, `cell_renderer_toggle` or `cell_renderer_pixbuf`.)

`GTree.view_column` is type of:

```
val GTree.view_column :
 ?title:string ->
 ?renderer:#cell_renderer * (string * ’a column) list ->
 unit -> view_column
```

Internally, this function uses:

```
method GTree.view_column#add_attribute : 'b 'c. (#cell_renderer as 'b) -> string -> 'c column -> unit
```

Note that the name of property is given as `string` type and the value of the property is given as `'a column` type. When setting attributes it is very important that the data type stored in a model column is the same as the data type that a property requires as argument.

This method adds an attribute mapping to the list in `view_column`. The `string` parameter is the C-style name of attribute on cell_renderer to be set from the value, and the `column` is the column of the model to get a value from. So for example if column *col_first_name* of the model contains strings, you could have the "text" attribute of a `GTree.cell_renderer_text` get its values from column *col_first_name*.

As for the name of properties, `GTree.cell_renderer_*` use Ocaml-style name such as `‘TEXT`, while `GTree.view_column` or `GTree.view_column#add_attribute` uses C-style name like "text", which indicate the same property.

For cell properties, please refer `GTree.cell_properties` (`GtkCellRenderer`), `GTree.cell_properties_text` (`GtkCellRendererText`), `GTree.cell_properties_toggle` (`GtkCellRendererToggle`) and `GTree.cell_properties_pixbuf` (`GtkCellRendererPixbuf`).

There are two more noteworthy things about `GTree.cell_renderer_*` properties: one is that sometimes there are different properties which do the same, but take different arguments, such as the `‘FOREGROUND` and `‘FOREGROUND_GDK` properties of `GTree.cell_renderer_text` (which specify the text colour). The `‘FOREGROUND` property take a colour in string form, such as "Orange" or "CornflowerBlue", whereas `‘FOREGROUND_GDK` takes a `Gdk.color` (GdkColor) argument. It is up to you to decide which one to use - the effect will be the same. The other thing worth mentioning is that most properties have a `‘FOO_SET` property taking a boolean value as argument, such as `‘FOREGROUND_SET`. This is useful when you want to have a certain setting have an effect or not. If you set the `‘FOREGROUND` property, but set `‘FOREGROUND_SET` to `false`, then your foreground color setting will be disregarded. This is useful in cell data functions (see below), or, for example, if you want set the foreground colour to a certain value at start-up, but only want this to be in effect in some columns, but not in others (in which case you could just connect the `‘FOREGROUND_SET` property to a model column of type `bool` with the functions explained previously).

Setting column attributes is the most straight-forward way to get your model data to be displayed. This is usually used whenever you want the data in the model to be displayed exactly as it is in the model.

Another way to get your model data displayed on the screen is to set up cell data functions.

5.3. Cell Data Functions

A cell data function is a function that is called for a specific cell renderer for each single row before that row is rendered. It gives you maximum control over what exactly is going to be rendered, as you can set the cell renderer's properties just like you want to have them. Remember not only to *set* a property if you want it to be active, but also to *unset* a property if it should not be active (and it might have been set in the previous row).

Cell data functions are often used if you want more fine-grained control over what is to be displayed, or if the standard way to display something is not quite like you want it to be. A case in point are floating point numbers. If you want floating point numbers to be displayed in a certain way, say with only one digit after the colon/comma, then you need to use a cell data function. Use `GTree.view_column#set_cell_data_func` (`gtk_tree_view_column_set_cell_data_func`) to set up a cell data function for a particular cell renderer. Here is an example:

```
(* file: data_func.ml *)

open Gobject.Data

let cols = new GTree.column_list
let col_name = cols#add string
let col_age = cols#add float

let create_model data =
  let store = GTree.tree_store cols in
  ...

let age_cell_data_func renderer (model:GTree.model) iter =
  let age = model#get ~row:iter ~column:col_age in
  renderer#set_properties [‘TEXT (Printf.sprintf "%.1f" age)]

let create_view ~model ~packing () =
  ...
  let renderer_text = GTree.cell_renderer_text [] in
  let col = GTree.view_column ~title:"Age" ~renderer:(renderer_text, []) () in
  col#set_cell_data_func renderer_text (age_cell_data_func renderer_text);
  ...
```

for each row to be rendered by this particular cell renderer, the cell data function is going to be called, which then retrieves the float from the model, and turns it into a string where the float has only one digit after the colon/comma, and renders that with the text cell renderer.

This is only a simple example, you can make cell data functions a lot more complicated if you want to. As always, there is a trade-off to keep in mind though. Your cell data function is going to be called every single time a cell in that (renderer) column is going to be rendered. Go and check how often this function is called in your program if you ever use one. If you do time-consuming operations within a cell data function, things are not going to be fast, especially if you have a lot of rows. The alternative in this case would have been to make an additional column *column_age_float_string* of type `string`, and to set the float in string form whenever you set the float itself in a row, and then hook up the string column to a text cell renderer using attributes. This way the float to string conversion would only need to be done once. This is a cpu cycles / memory trade-off, and it depends on your particular case which one is more suitable. Things you should probably not do is to convert long strings into UTF8 format in a cell data function, for example.

You might notice that your cell data function is called at times even for rows that are not visible at the moment. This is because the tree view needs to know its total height, and in order to calculate this it needs to know the height of each and every single row, and it can only know that by having it measured, which is going to be slow when you have a lot of rows with different heights (if your rows all have the same height, there should not be any visible delay though).

5.4. GTree.cell_renderer_text and Integer, Boolean and Float Types

It has been said before that, when using attributes to connect data from the model to a cell renderer property, the data in the model column specified in `GTree.view_column#add_attribute` (`gtk_tree_view_column_add_attribute`) must always be of the same type as the data type that the property requires.

This is usually true, but there is an exception: if you use `GTree.view_column#add_attribute` (`gtk_tree_view_column_add_attribute`) to connect a text cell renderer’s `"text"` property to a model column, the model column does not need to be of `string`, it can also be one of most other fundamental `Gobject.Data` types, e.g. `boolean`, `int`, `uint`, `long`, `ulong`, `int64`, `uint64`, `float`, or `double`. The text cell renderer will automatically display the values of these types correctly in the tree view. For example:

```
(* file: born.ml *)

open Gobject.Data

let cols = new GTree.column_list
let col_name = cols#add string
let col_born = cols#add uint

let create_model data =
  let store = GTree.tree_store cols in
  ...

let create_view ~packing () =
  ...
  let cell = GTree.cell_renderer_text [] in
  let col = GTree.view_column ~title:"Born"
      ~renderer:(cell, ["text", col_born]) () in
  ...
```

Even though the `"text"` property would require a model column of a string value, we use a model column of an integer type when setting attributes. The integer will then automatically be converted into a string before the cell renderer property is set.

If you are using a floating point type, ie. `float` or `double`, there is no way to tell the text cell renderer how many digits after the floating point (or comma) should be rendered. If you only want a certain amount of digits after the point/comma, you will need to use a cell data function.

5.5. GTree.cell_renderer_text, UTF8, and pango markup

All text used in Gtk+-2.0 widgets needs to be in UTF8 encoding, and `GtkCellRendererText` is no exception. Text in plain ASCII is automatically valid UTF8, but as soon as you have special characters that do not exist in plain ASCII (usually characters that are not used in the English language alphabet), they need to be in UTF8 encoding. There are many different character encodings that all specify different ways to tell the computer which character is meant. Gtk+-2.0 uses UTF8, and whenever you have text that is in a different encoding, you need to convert it to UTF8 encoding first, using some functions of `Glib.Convert` module (GLib `g_convert` family of functions). If you only use text input from other Gtk+ widgets, you are on the safe side, as they will return all text in UTF8 as well.

However, if you use 'external' sources of text input, then you must convert that text from the text's encoding (or the user's locale) to UTF8, or it will not be rendered correctly (either not at all, or it will be cut off after the first invalid character). Filenames are especially hard, because there is no indication whatsoever what character encoding a filename is in (it might have been created when the user was using a different locale, so filename encoding is basically unreliable and broken). You may want to convert to UTF8 with fallback characters in that case. You can check whether a string is valid UTF8 with `Glib.Utf8.validate` (`g_utf8_validate`). You should, in this author's opinion at least, put these checks into your code at crucial places wherever it is not affecting performance, especially if you are an English-speaking programmer that has little experience with non-English locales. It will make it easier for others and yourself to spot problems with non-English locales later on.

In addition to the "text" property, GTree.cell_renderer_text also has a "markup" property that takes text with pango markup as input. Pango markup allows you to place special tags into a text string that affect the style the text is rendered (see the pango documentation). Basically you can achieve everything you can achieve with the other properties also with pango markup (only that using properties is more efficient and less messy). Pango markup has one distinct advantage though that you cannot achieve with text cell renderer properties: with pango markup, you can change the text style in the middle of the text, so you could, for example, render one part of a text string in bold print, and the rest of the text in normal. Here is an example of a string with pango markup:

`"You can have text in <b>bold</b> or in a <span color='Orange'>different color</span>"`

When using the `"markup"` property, you need to take into account that the `"markup"` and `"text"` properties do not seem to be mutually exclusive (I suppose this could be called a bug). In other words: whenever you set `"markup"` (and have used the `"text"` property before), do not set the `"text"` property, and vice versa. Example:

```
...
let foo_cell_data_func renderer (model:GTree.model) iter =
  ...
  let important = model#get ~row:iter ~column:col_important in
  if important
  then renderer#set_properties [‘MARKUP "<b>important</b>"]
  else renderer#set_properties [‘TEXT "not important"];
  ...
```

Another thing to keep in mind when using pango markup text is that you might need to escape text if you construct strings with pango markup on the fly using random input data. For example:

```
...
let foo_cell_data_func renderer (model:GTree.model) iter =
  ...
  (* This might be problematic if artist_string or title_string
   * contain markup characters/entities: *)
  let markuptxt = Printf.sprintf "<b>%s</b> - <i>%s</i>"
    artist_string title_string in
  renderer#set_properties [‘MARKUP markuptxt];
  ...
```

The above example will not work if artist_string is "Simon & Garfunkel" for example, because the & character is one of the characters that is special. They need to be escaped, so that pango knows that they do not refer to any pango markup, but are just characters. In this case the string would need to be "Simon &amp; Garfunkel" in order to make sense in between the pango markup in which it is going to be pasted. You can escape a string with `Glib.Markup.escape_text` (`g_markup_escape`).

It is possible to combine both pango markup and text cell renderer properties. Both will be 'added' together to render the string in question, only that the text cell renderer properties will be applied to the whole string. If you set the `"markup"` property to normal text without any pango markup, it will render as normal text just as if you had used the `"text"` property. However, as opposed to the `"text"` property, special characters in the `"markup"` property text would still need to be escaped, even if you do not use pango markup in the text.

5.6. A Working Example

Here is our example from the very beginning again (with an additional column though), only that the contents of the model are rendered properly on the screen this time. Both attributes and a cell data function are used for demonstration purposes.

```
(* file: name_born.ml *)

open Gobject.Data

let cols = new GTree.column_list
let col_first_name = cols#add string
let col_last_name = cols#add string
let col_year_born = cols#add uint

let create_and_fill_model data =
  let treestore = GTree.tree_store cols in
  let append (first_name, last_name, born) =
    let toplevel = treestore#append () in
    treestore#set ~row:toplevel ~column:col_first_name first_name;
    treestore#set ~row:toplevel ~column:col_last_name last_name;
    if born > 0
    then treestore#set ~row:toplevel ~column:col_year_born born
  in
  List.iter append [
    ("Maria", "Incognito", 0);
    ("Jane", "Average", 1962);
    ("Janinita", "Average", 1985)
  ];
  treestore

let age_cell_data_func renderer (model:GTree.model) iter =
  let year_now = 2004 in
  let year_born = model#get ~row:iter ~column:col_year_born in
  if year_born <= year_now && year_born > 0
  then (
    let age = year_now - year_born in
    renderer#set_properties [
      ‘TEXT (Printf.sprintf "%u years old" age);
      ‘FOREGROUND_SET false
    ]
  ) else (
    renderer#set_properties [
      ‘TEXT "age unknown";
      ‘FOREGROUND "Red";
      ‘FOREGROUND_SET true;
    ]
  )

let create_view_and_model ~packing () =
  let view = GTree.view ~packing () in

  (* Column #1 *)
  (* pack cell renderer into tree view column *)
  (* connect 'text' property of the cell renderer to
   * model column that contains the first name *)
  let col = GTree.view_column ~title:"First Name"
      ~renderer:(GTree.cell_renderer_text [], ["text", col_first_name]) () in
  (* pack tree view column into tree view *)
  view#append_column col;

  (* Column #2 *)
  (* create cell_renderer and set 'weight' property of it to
   * bold print (we want all last name in bold) *)
  let cell_renderer = GTree.cell_renderer_text [‘WEIGHT ‘BOLD] in
  (* pack cell renderer into tree view column *)
  (* connect ’text’ property of the cell renderer to
   * model column that contains the last name *)
  let col = GTree.view_column ~title:"Last Name"
      ~renderer:(cell_renderer, ["text", col_last_name]) () in
  (* pack tree view column into tree view *)
  view#append_column col;

  let renderer = GTree.cell_renderer_text [] in
  (* pack cell renderer into tree view column *)
  let col = GTree.view_column ~title:"Age"
      ~renderer:(renderer, []) () in
  (* connect a cell data function *)
  col#set_cell_data_func renderer (age_cell_data_func renderer);
  (* pack tree view column into tree view *)
  view#append_column col;

  let model = create_and_fill_model () in
  view#set_model (Some (model#coerce));

  view#selection#set_mode ‘NONE;
  view

let main () =
  let window = GWindow.window ~title:"Treeview" ~border_width:10 () in
  window#connect#destroy ~callback:GMain.Main.quit;
  let view = create_view_and_model ~packing:window#add () in
  window#show ();
  GMain.Main.main ()

let _ = Printexc.print main ()
```

5.7. How to Make a Whole Row Bold or Coloured

This seems to be a frequently asked question, so it is worth mentioning it here. You have the two approaches mentioned above: either you use cell data functions, and check in each whether a particular row should be highlighted in a particular way (bold, coloured, whatever), and then set the renderer properties accordingly (and unset them if you want that row to look normal), or you use attributes. Cell data functions are most likely not the right choice in this case though.

If you only want every second line to have a gray background to make it easier for the user to see which data belongs to which line in wide tree views, then you do not have to bother with the stuff mentioned here. Instead just set the rules hint on the tree view as described in the here, and everything will be done automatically, in colours that conform to the chosen theme even (unless the theme disables rule hints, that is).

Otherwise, the most suitable approach for most cases is that you add two columns to your model, one for the property itself (e.g. a column *col_row_color* of type `string`), and one for the boolean flag of the property (e.g. a column *col_row_color_set* of type `boolean`). You would then connect these columns with the `"foreground"` and `"foreground-set"` properties of each renderer. Now, whenever you set a row’s *col_row_color* field to a colour, and set that row’s *col_row_color_set* field to `true`, then this column will be rendered in the colour of your choice. If you only want either the default text colour or one special other colour, you could even achieve the same thing with just one extra model column: in this case you could just set all renderer’s `"foreground"` property to whatever special color you want, and only connect the *col_row_color_set* column to all renderer’s `"foreground-set"` property using attributes. This works similar with any other attribute, only that you need to adjust the data type for the property of course (e.g. `"weight"` would take a `int`, in form of a `PANGO_WEIGHT_FOO` define in this case).

As a general rule, you should not change the text colour or the background colour of a cell unless you have a really good reason for it. To quote Havoc Pennington: “Because colors in GTK+ represent a theme the user has chosen, you should never set colors purely for aesthetic reasons. If users don’t like GTK+ gray, they can change it themselves to their favorite shade of orange.”

5.8. How to Pack Icons into the Tree View

So far we have only put text in the tree view. While everything you need to know to display icons (in the form of `GdkPixbufs`) has been introduced in the previous sections, a short example might help to make things clearer. The following code will pack an icon and some text into the same tree view column:

```
(* file: icon.ml *)

let cols = new GTree.column_list
let col_icon: GdkPixbuf.pixbuf GTree.column = cols#add Gobject.Data.gobject
let col_text = cols#add Gobject.Data.string

let create_liststore () =
  let store = GTree.list_store cols in

  let icon = GdkPixbuf.from_file "gtk.xpm" in
  let row = store#append () in
  store#set ~row ~column:col_icon icon;
  store#set ~row ~column:col_text "example";
  store

let create_treeview ~packing () =
  let model = create_liststore () in

  let view = GTree.view ~model ~packing () in

  let renderer = (GTree.cell_renderer_pixbuf [], [("pixbuf", col_icon)]) in
  let col = GTree.view_column ~title:"Title" ~renderer () in
  view#append_column col;

  let renderer_text = GTree.cell_renderer_text [] in
  let col = GTree.view_column ~title:"Text"
      ~renderer:(renderer_text, [("text", col_text)]) () in
  view#append_column col;

  view

...
```

Note that the tree view will not resize icons for you, but displays them in their original size. If you want to display stock icons instead of `GdkPixbufs` loaded from file, you should have a look at the `'STOCK_ID` property of `GTree.cell_properties_pixbuf` (`GTree.cell_renderer_pixbuf`) (and your model column should be of type `string`, as all stock IDs are just strings by which to identify the stock icon).

Chapter 6. Selections, Double-Clicks and Context Menus

6.1. Handling Selections

One of the most basic features of a list or tree view is that rows can be selected or unselected. Selections are handled using the `GTree.selection` (`GtkTreeSelection`) object of a tree view. Every tree view automatically has a `GTree.selection` associated with it, and you can get it using `GTree.view#selection` method (`gtk_tree_view_get_selection`). Selections are handled completely on the tree view side, which means that the model knows nothing about which rows are selected or not. There is no particular reason why selection handling could not have been implemented with functions that access the tree view widget directly, but for reasons of API cleanliness and code clarity the Gtk+ developers decided to create this special `GTree.selection` object that then internally deals with the tree view widget. You will never need to create a tree selection object, it will be created for you automatically when you create a new tree view. You only need to use said `selection` method to get the selection object.

There are three ways to deal with tree view selections: either you get a list of the currently selected rows whenever you need it, for example within a context menu function, or you keep track of all select and unselect actions and keep a list of the currently selected rows around for whenever you need them; as a last resort, you can also traverse your list or tree and check each single row for whether it is selected or not (which you need to do if you want all rows that are *not* selected for example).

6.1.1. Selection Modes

You can use `set_mode` method of the `GTree.selection` object to influence the way that selections are handled. There are four selection modes:

- `‘NONE` - no items can be selected
- `‘SINGLE` - no more than one item can be selected
- `‘BROWSE` - exactly one item is always selected
- `‘MULTIPLE` - anything between no item and all items can be selected

6.1.2. Getting the Currently Selected Rows

You can get a `Gtk.tree_path list` of the selected rows using `get_selected_rows` method (`gtk_tree_selection_get_selected_rows`).

It is used like this:

```
...
let cols = new GTree.column_list
let col_name = cols#add Gobject.Data.string
let col_age = cols#add Gobject.Data.int
...

let selection_changed (model:#GTree.model) selection () =
  let pr path =
    let row = model#get_iter path in
    let name = model#get ~row ~column:col_name in
    let age = model#get ~row ~column:col_age in
    Printf.printf "(%s, %d)\n" name age;
    flush stdout
  in
  List.iter pr selection#get_selected_rows

let create_view ~model ~packing () =
  ...
  view#selection#set_mode ‘MULTIPLE;
  view#selection#connect#changed ~callback:(selection_changed model view#selection);
  ...
```

One thing you need to be aware of is that you need to take care when looping through the list that `get_selected_rows` returns (because it contains paths, and when you remove rows in the middle, then the old paths will point to either a non-existing row, or to another row than the one selected). You have two ways around this problem: one way is to use the solution to removing multiple rows that has been described above, ie. to get tree row references for all selected rows and then remove the rows one by one; the other solution is to sort the list of selected tree paths so that the last rows come first in the list, so that you remove rows from the end of the list or tree. You cannot remove rows from within a foreach callback in any case, that is simply not allowed.

6.1.3. Using Selection Functions

You can set up a custom selection function with `GTree.selection#set_select_function` (`gtk_tree_selection_set_select_function`). This function will then be called every time a row is going to be selected or unselected (meaning: it will be called before the selection status of that row is changed). Selection functions are commonly used for the following things:

1. ... to keep track of the currently selected items (then you maintain a list of selected items yourself). In this case, note again that your selection function is called *before* the row's selection status is changed. In other words: if the row is *going to be* selected, then the boolean parameter that is passed to the selection function is still `false`. Also note that the selection function might not always be called when a row is removed, so you either have to unselect a row before you remove it to make sure your selection function is called and removes the row from your list, or check the validity of a row when you process the selection list you keep. You should not store tree paths in your self-maintained list of of selected rows, because whenever rows are added or removed or the model is resorted the paths might point to other rows. Use tree row references or other unique means of identifying a row instead.
2. ... to tell Gtk+ whether it is allowed to select or unselect that specific row (you should make sure though that it is otherwise obvious to a user whether a row can be selected or not, otherwise the user will be confused if she just cannot select or unselect a row). This is done by returning `true` or `false` in the selection function.
3. ... to take additional action whenever a row is selected or unselected.

Yet another simple example:

```
let view_selection_func (model:#GTree.model) path currently_selected =
  let row = model#get_iter path in
  let name = model#get ~row ~column:col_name in
  if not currently_selected
  then Printf.printf "%s is going to be selected.\n" name
  else Printf.printf "%s is going to be unselected.\n" name;
  flush stdout;
  true (* allow selection state to change *)

let create_view ~model ~packing () =
  let view = GTree.view ~model ~packing () in
  ..
  view#selection#set_select_function (view_selection_func model);
  ...
```

6.1.4. Checking Whether a Row is Selected

You can check whether a given row is selected or not using the functions `GTree.selection#iter_is_selected` (`gtk_tree_selection_iter_is_selected`) or `GTree.selection#path_is_selected` (`gtk_tree_selection_path_is_selected`). If you want to know all rows that are *not* selected, for example, you could just traverse the whole list or tree, and use the above functions to check for each row whether it is selected or not.

6.1.5. Selecting and Unselecting Rows

You can select or unselect rows manually with `GTree.selection#select_iter` (`gtk_tree_selection_select_iter`), `GTree.selection#select_path` (`gtk_tree_selection_select_path`), `GTree.selection#unselect_iter` (`gtk_tree_selection_unselect_iter`), `GTree.selection#unselect_path` (`gtk_tree_selection_unselect_path`), `GTree.selection#select_all` (`gtk_tree_selection_select_all`), and `GTree.selection#unselect_all` (`gtk_tree_selection_unselect_all`) should you ever need to do that.

6.2. Double-Clicks on a Row

Catching double-clicks on a row is quite easy and is done by connecting to a tree view's "row-activated" signal, like this:

```
...

let on_row_activated (view:GTree.view) path column =
  let model = view#model in
  let row = model#get_iter path in
  let name = model#get ~row ~column:col_name in
  Printf.printf "Double-clicked row contains name %s\n" name;
  flush stdout

let create_view ~model ~packing () =
  let view = GTree.view ~model ~packing () in
  ...
  view#connect#row_activated ~callback:(on_row_activated view);
  ...

...
```

6.3. Context Menus on Right Click

Context menus are context-dependent menus that pop up when a user right-clicks on a list or tree and usually let the user do something with the selected items or manipulate the list or tree in other ways.

Right-clicks on a tree view are caught just like mouse button clicks are caught with any other widgets, namely by connecting to the tree view's "button_press_event" signal handler (which is a GtkWidget signal, and as GtkTreeView is derived from GtkWidget it has this signal as well). Also, you should make sure that all functions provided in your context menu can also be accessed by other means such as the application's main menu. See the GNOME Human Interface Guidelines (HIG) for more details. Straight from the a-snippet-of-code-says-more-than-a-thousand-words-department, some code to look at:

```
(* file: popup.ml *)

...
let cols = new GTree.column_list
let col_name = cols#add Gobject.Data.string
let col_age = cols#add Gobject.Data.int
...

let on_doSomething treeview () =
  Printf.printf "Do something!\n";
  flush stdout

let view_popup_menu treeview ev =
  let menu = GMenu.menu () in
  let menuitem = GMenu.menu_item ~label:"Do something" ~packing:menu#append () in
  menuitem#connect#activate ~callback:(on_doSomething treeview);
  menu#popup ~button:(GdkEvent.Button.button ev) ~time:(GdkEvent.Button.time ev)

let on_button_pressed treeview ev =
  if GdkEvent.Button.button ev = 3 then (
    Printf.printf "Single right click on the tree view.\n";

    (* optional: select row if no row is selected or only
     *  one other row is selected (will only do something
     *  if you set a tree selection mode as described later
     *  in the tutorial) *)
    if true then begin
      let selection = treeview#selection in

      (* Note: gtk_tree_selection_count_selected_rows() does not
       *   exist in gtk+-2.0, only in gtk+ >= v2.2 ! *)
      if selection#count_selected_rows <= 1 then (
     let x = int_of_float (GdkEvent.Button.x ev) in
     let y = int_of_float (GdkEvent.Button.y ev) in
        let Some (path, _, _, _) = treeview#get_path_at_pos ~x ~y in
        selection#unselect_all ();
        selection#select_path path
         )
      end; (* end of optional bit *)

      view_popup_menu treeview ev;
      true (* we handled this *)
    ) else
      false (* we did not handle this *)

let create_view ~packing () =
  let view = GTree.view ~packing () in
  ...
  view#event#connect#button_press ~callback:(on_button_pressed view);
  ...
```

Chapter 7. Sorting

Lists and trees are meant to be sorted. This is done using the `GTree.tree_sortable` (`GtkTreeSortable`) interface. `GTree.list_store` and `GTree.tree_store` inherit from `GTree.tree_sortable` which in turn inherits from `GTree.model`.

The most straight forward way to sort a list store or tree store is to directly use the tree sortable interface on them. This will sort the store in place, meaning that rows will actually be reordered in the store if required. This has the advantage that the position of a row in the tree view will always be the same as the position of a row in the model, in other words: a tree path refering to a row in the view will always refer to the same row in the model, so you can get a row's iter easily with `Gtree.model.get_iter` using a tree path supplied by the tree view. This is not only convenient, but also sufficient for most scenarios.

7.1. GTree.tree_sortable

The tree sortable interface is fairly simple and should be easy to use. Basically you define a 'sort column ID' integer for every criterion you might want to sort by and tell the tree sortable which function should be called to compare two rows (represented by two tree iters) for every sort ID with `GTree.tree_sortable#set_sort_func` method (`gtk_tree_sortable_set_sort_func`). Then you sort the model by setting the sort column ID and sort order with `GTree.tree_sortable#set_sort_column_id` method (`gtk_tree_sortable_set_sort_column_id`), and the model will be re-sorted using the compare function you have set up. Your sort column IDs can correspond to your model columns, but they do not have to (you might want to sort according to a criterion that is not directly represented by the data in one single model column, for example). Some code to illustrate this:

```
(* file: sortable.ml *)

open GTree

let cols = new GTree.column_list
let col_name = cols#add Gobject.Data.string
let col_year_born = cols#add Gobject.Data.uint

...

let compare a b =
  if a < b then -1
  else if a > b then 1
  else 0

let sort_by_name (model:#GTree.model) row1 row2 =
  let name1 = model#get ~row:row1 ~column:col_name in
  let name2 = model#get ~row:row2 ~column:col_name in
  compare name1 name2

...

let create_list_and_view ~packing () =
  let liststore = create_and_fill_model () in

  liststore#set_sort_func col_name.index sort_by_name;
  liststore#set_sort_func col_year_born.index sort_by_year_born;

  (* set initial sort order *)
  liststore#set_sort_column_id col_name.index `ASCENDING;

  let view = GTree.view ~model:liststore ~packing () in

  ...
```

Usually things are a bit easier if you make use of the tree view column headers for sorting, in which case you only need to assign sort column IDs and your compare functions, but do not need to set the current sort column ID or order yourself (see below).

Your tree iter compare function should return a negative value if the row specified by iter a comes before the row specified by iter b, and a positive value if row b comes before row a. It should return 0 if both rows are equal according to your sorting criterion (you might want to use a second sort criterion though to avoid 'jumping' of equal rows when the store gets resorted). Your tree iter compare function should not take the sort order into account, but assume an ascending sort order (otherwise bad things will happen).

7.2. Sorting and Tree View Column Headers

Unless you have hidden your tree view column headers or use custom tree view column header widgets, each tree view column's header can be made clickable. Clicking on a tree view column's header will then sort the list according to the data in that column. You need to do two things to make this happen: firstly, you need to tell your model which sort function to use for which sort column ID with `GTree.tree_sortable#set_sort_func` method (`gtk_tree_sortable_set_sort_func`). Once you have done this, you tell each tree view column which sort column ID should be active if this column's header is clicked. This is done with `GTree.view_column#set_sort_column_id` method (`gtk_tree_view_column_set_sort_column_id`).

And that is really all you need to do to get your list or tree sorted. The tree view columns will automatically set the active sort column ID and sort order for you if you click on a column header.

Chapter 8. Editable Cells

8.1. Editable Text Cells

With `GTree.cell_render_text` you can not only display text, but you can also allow the user to edit a single cell's text right in the tree view by double-clicking on a cell.

To make this work you need to tell the cell renderer that a cell is editable, which you can do by setting the `‘EDITABLE` property of the text cell renderer in question to `true`. You can either do this on a per-row basis (which allows you to set each single cell either editable or not) by connecting the `"editable"` property to a boolean type column in your tree model using attributes; or you can just do a ...

```
  let renderer = GTree.cell_renderer_text [‘EDITABLE true; ...] in
```

... when you create the renderer, which sets all rows in that particular renderer column to be editable.

Now that our cells are editable, we also want to be notified when a cell has been edited. This can be achieved by connecting to the cell renderer's `"edited"` signal like this:

```
  renderer#connect#edited ~callback:cell_edited_callback;
```

This callback is then called whenever a cell has been edited. We can pass a model as a parameter of the callback function, as we probably want to store the new value in the model.

The `"edited"` signal callback looks like this:

```
method edited : callback:(Gtk.tree_path -> string -> unit) -> GtkSignal.id
```

The tree path is passed to the `"edited"` signal callback. You can convert this into an iter with `GTree.model#get_iter`.

Note that the cell renderer will not change the data for you in the store. After a cell has been edited, you will only receive an `"edited"` signal. If you do not change the data in the store, the old text will be rendered again as if nothing had happened.

If you have multiple (renderer) columns with editable cells, it is not necessary to have a different callback for each renderer, you can use the same callback for all renderers, and attach some data to each renderer, which you can later retrieve again in the callback to know which renderer/column has been edited. This is done like this, for example:

```
...
let cols = new GTree.column_list
let col_name = cols#add Gobject.Data.string
let col_age = cols#add Gobject.Data.int
...

let cell_edited_callback index path str =
  ...

let create_view () =
  ...
  let renderer = GTree.cell_renderer_text [‘EDITABLE true] in
  renderer#connect#edited ~callback:(cell_edited_callback col_name.index);
  ...
  let renderer = GTree.cell_renderer_text [‘EDITABLE true] in
  renderer#connect#edited ~callback:(cell_edited_callback col_age.index);
  ...
```

The `column` such as `col_name` and `col_age` has a field called `index` which is unique in a `GTree.column_list`.

8.1.1. Setting the cursor to a specific cell

You can move the cursor to a specific cell in a tree view with `GTree.view#set_cursor` (`gtk_tree_view_set_cursor`), and start editing the cell if you want to. Similarly, you can get the current row and focus column with `GTree.view#get_cursor` (`gtk_tree_view_get_cursor`). Use `misc#grab_focus` method (`gtk_widget_grab_focus(treeview)`) will make sure that the tree view has the keyboard focus.

As the API reference points out, the tree view needs to be realised for cell editing to happen. In other words: If you want to start editing a specific cell right at program startup, you need to set up an idle timeout with `GMain.Idle.add` (`g_idle_add`) that does this for you as soon as the window and everything else has been realised (return `false` in the timeout to make it run only once).

Connect to the tree view's `"cursor-changed"` and/or `"move-cursor"` signals to keep track of the current position of the cursor.

8.2. Editable Toggle and Radio Button Cells

Just like you can set a `GTree.cell_renderer_text` editable, you can specify whether a `GTree.cell_renderer_toggle` should change its state when clicked by setting the `‘ACTIVATABLE` property - either when you create the renderer (in which case all cells in that column will be clickable) or by connecting the renderer property to a model column of boolean type via attributes.

Connect to the `"toggled"` signal of the toggle cell renderer to be notified when the user clicks on a toggle button (or radio button). The user click will not change the value in the store, or the appearance of the value rendered. The toggle button will only change state when you update the value in the store. Until then it will be in an "inconsistent" state, which is also why you should read the current value of that cell from the model, and not from the cell renderer.

The `"toggled"` signal callback looks like this:

```
method toggled : callback:(Gtk.tree_path -> unit) -> GtkSignal.id
```

Just like with the `"edited"` signal of the text cell renderer, the tree path is passed to the `"toggled"` signal callback. You can convert it into an iter with `GTree.model#get_iter`.

Chapter 9. Other Resources

A short tutorial like this cannot possibly cover everything. Luckily, there is a lot more information out there. Here is a list of links that you might find useful (if you have any links that should appear here as well, please send them to `tim at centricular dot net`).

- Gtk+ API Reference Manual
- Gdk API Reference Manual
- Pango API Reference Manual
- GLib API Reference Manual
- gtk-app-devel mailing list archives - search them!
- gtk-demo - part of the Gtk+ source code (look in gtk+-2.x.y/demos/gtk-demo), especially list_store.c, tree_store.c, and stock_browser.c
- TreeView tutorial using Gtk's C++ interface (gtkmm)
- TreeView tutorial using Gtk's python interface
- Some slides from Owen Taylor's GUADEC 2003 tutorial (postscript, pdf, see pages 13-15)
- Existing applications - yes, they exist, and *you* can look at their source code. SourceForge's WebCVS browse feature is quite useful, and the same goes for GNOME as well.
- If your intention is to display external data (from a database, or in XML form) as a list or tree or table, you might also be interested GnomeDB, especially libgda and libgnomedb (e.g. the GnomeDBGrid widget). See also this PDF presentation (page 24ff).
- your link here!

Chapter 10. Copyright, License, Credits, and Revision History

10.1. Copyright and License

10.1.1. Ocaml Adaptation

10.1.2. Original Document

10.2. Credits

10.2.1. Ocaml Adaptation

Thanks to Tim-Philipp Muller for writing such wonderful source material.

And of course thanks to the Ocaml and LablGtk teams.

10.2.2. Original Document

Thanks to Axel C. for proof-reading the first drafts, for many suggestions, and for introducing me to the tree view widget in the first place (back then when I was still convinced that porting to Gtk+-2.x was unnecessary, Gtk+-1.2 applications looked nice, and Aristotle had already said everything about politics that needs to be said).

Harring Figueiredo shed some light on how GtkListStore and GtkTreeStore deal with pixbufs.

Ken Rastatter suggested some additional topics (with complete references even).

Both Andrej Prsa and Alan B. Canon sent me a couple of suggestions, and 'taf2', Massimo Mangoni and others spotted some typos.

Many thanks to all of them, and of course also to kris and everyone else in #gtk+.

10.3. Revision History

10.3.1. Ocaml Adaptation

September 2004

- First release of Ocaml adaptation version. Version 0.1

10.3.2. Original Document

30th April 2004

- Added Hello World

31st March 2004

- Fixed fatal typo in custom list code: g_assert() in custom_list_init() should be ==, not != (spotted by mmc).

- Added link to Owen Taylor's mail on the GtkTreeView Drag'n'Drop API.

24th January 2004

- Fixed typo in code example (remove n-th row example) (Thanks to roel for spotting it).
- Changed 'Context menus' section title

19th January 2004

- Expanded section on GtkTreeRowReferences, and on removing multiple rows.

8th January 2004

- Added tiny section on Glade and treeviews
- Added more detail to the section describing GtkTreePath, GtkTreeIter et.al.
- Reformatted document structure: instead of one single chapter with lots of sections, have multiple chapters (this tutorial is way to big to become part of the Gtk+ tutorial anyway); enumerate chapters and sections.
- Expanded the section on tree view columns and cell renderers, with help of two diagrams by Owen Taylor (from the GUADEC 2003 Gtk+ tutorial slides).

10th December 2003

- Added more information about how to remove a single row, or more specifically, the n-th row of a list store
- Added a short example about how to pack icons into the tree view.

28th October 2003

- Editable cells will work fine even if selection is set to GTK_SELECTION_NONE. Removed sentences that say otherwise.

23rd October 2003

- fix 'jumpy' selections in custom model GtkTreeSortable interface implementation. gtk_tree_model_rows_reordered() does not seem to work like the API reference implies (see bug #124790)
- added section about how to get the cell renderer a button click happened on
- added section about editable cells with spin buttons (and a CellRendererSpin implementation to the examples)

10th October 2003

- make custom model GtkTreeSortable implementation emit "sort-column-changed" signal when sortid is changed
- fixed code typo in selection function section; added a paragraph about rule hint to 'make whole row coloured or bold' section

7th October 2003

- Reformatted source code to make it fit on pages when generating ps/pdf output
- Added link to PDF and docbook XML versions.