

# Programming GUIs using Qt

## 1.1 An introductory example

To get a small feel for how one programs a GUI using `qtbases`, the R package that interfaces R with the Qt libraries, we show how to produce a simple dialog to collect a date from a user.

If the underlying libraries and package are installed, the package is loaded as any other R package:

```
require(qtbases)
```

**Constructors** As with all other toolkits, in Qt, GUI components are created with constructors. For this example, we will set various properties later, rather than at construction time. For our GUI we have four basic widgets: a widget used as a container to hold the others, a label, a single line edit area and a button.

```
w <- Qt$QWidget()
l <- Qt$QLabel()
e <- Qt$QLineEdit()
b <- Qt$QPushButton()
```

The constructors are not found in the global environment, but rather are found in the Qt environment provided through `qtbases`. As such, the `$` lookup operator is used. For this example, we use a `QWidget` as a top-level window, leaving for to discuss the `QMainWindow` object and its task-tailored features.

Widgets in Qt have various properties that set the state of the object. For example, the window object, `w`, has the `windowTitle` property that is adjusted as follows:

```
w$windowTitle <- "An example"
```

Qt objects are essentially environments. In the above, the named component `windowTitle` of the environment holds the value of the `windowTitle` property of the object, so the `$` use is simply that for environments.

Figure 1.1: Screenshot of our sample GUI to collect a date from the user.

More typical, is a method call. Qt overloads the `$` operator for method calls (as does RGtk2). For example, both the button object and label object have a text property. The setter `setText` can be used to assign a value. For example,

```
l$setText("Date:")
b$setText("Ok")
```

Although, the calling mechanism is more complicated than just the lookup of a function stored as the component `setText` (cf. ), as the object is passed into the body of the function, the usage is similar.

**Layout Managers** Qt uses layout managers to organize widgets. This is similar to Java/Swing and `tcltk`, but not RGtk2. Layout managers will be discussed more thoroughly in , but in this example we will use a grid layout to organize our widgets. The placement of child widgets into the grid is done through the `addWidget` method and requires a specification, by index and span, of the cells the child will occupy.

```
lyt <- Qt$QGridLayout()
lyt$addWidget(l, row=0, column=0, rowSpan=1, columnSpan=1)
lyt$addWidget(e, 0, 1, 1, 1)
lyt$addWidget(b, 1, 1, 1, 1)
```

One can adjust properties of the layout, but we leave that discussion for later.

We need to attach our layout to the widget `w`, which is done through the `setLayout` method:

```
w$setLayout(lyt)
```

To view our GUI (Figure 1.1), we must call its `show` method.

```
w$show()
```

**Callbacks** As with other GUI toolkits, we add interactivity to our GUI by binding callbacks to certain events. To add a command to the clicking or pressing of the button is done by attaching a handler to the “pressed” signal for the button (the “clicked” signal is only for mouse clicks). Widgets have various signals they emit. Additionally, there are window-manager events that may be of interest, but using them requires more work than is used below. The `qconnect` function is used to add a handler for a signal. The function needs, as a minimum, the object, the signal name and the handler. Here we print the value stored in the “Date” field.

```
handler <- function(checked) print(e$text)
id <- qconnect(b, "pressed", handler)
```

We will discuss callbacks more completely in .

**Refinements** At this point, we have a working dialog built with `qtbase`. There is much room for refinement, which due to Qt's many features are relatively easy to implement. For this example, we want to guide the user to fill out the date in the proper format. We could have used Qt's `QDateEdit` widget to allow point-and-click selection, but instead show two ways to help the user fill in the information with the keyboard

The `QLineEdit` widget has a number of ways to adjust its behavior. For example, an input mask provides a pattern for the user to fill out. For a date, we may want the value to be in the form "year-month-date." This would be specified with "0000-00-00", as seen by consulting the help page for `QLineEdit`. To add an input mask we have:

```
e$setInputMask("0000-00-00")
```

Further, for the line edit widget Qt can implement validation of the entered text. There are a few built-in validators, and for this purpose the regular expression validator could be used, but instead we wish to determine if we have a valid date by seeing if we can coerce the string value to a date via R's `as.Date` function with a format of "%Y-%m-%d". The method `setValidator` can be used to set the validator that is in charge of the validation. However, rather than passing a function, one must pass an instance of a validator class. For our specific needs, we need to create a new class.

**Object-oriented support** The underlying Qt libraries are written in C++. The object oriented nature is preserved by `qtbase`. Not only are the classes and methods implemented in R, the ability to implement new subclasses and methods is also possible. For this task, we need to implement a subclass of the `QValidator` class, and for this subclass implement a `validate` method. More detail on working with classes and methods in `qtbase` is provided in .

The `qsetClass` function is used to set a new class. To derive a subclass, we need just this:

```
qsetClass("dateValidator", Qt$QValidator, function(parent = NULL) {  
  super(parent)  
})
```

The `validate` method is implemented as a virtual class in Qt, in R we implement a method of our sub class. The `qsetMethod` is employed. The signature of the `validate` method is a string containing the input and an index indicating where the cursor is in the text box. The return value of this method indicates a state of "Acceptable", "Invalid", or if neither can be determined "Intermediate." These values are actually integers, and in this case are stored within an enumeration in the `Qt$QValidator` class.

```
qsetMethod("validate", dateValidator, function(input, pos) {  
  if(!grepl("[0-9]{4}-[0-9]{1,2}-[0-9]{1,2}$", input))  
    return(Qt$QValidator$Intermediate)  
  else if(is.na(as.Date(input, format="%Y-%m-%d")))  
    return(Qt$QValidator$Invalid)  
  else  
    return(Qt$QValidator$Acceptable)  
})
```

To use this new class, we call its constructor, which has the same name as the class, and then set it as a validator for the line edit widget:

```
validator <- dateValidator()  
e$setValidator(validator)
```

### 1.2 Overview

Qt is an open-sourced, cross-platform application and UI framework. Its history begins with Haavard Nord and Eirik Chambe-Eng in 1991, the Trolltech company until 2008, and now Nokia, a major cell-phone producer. While originally not an open-source project, it now has licensing under the LGPL that allows its use in open-source software.

Qt is developed in C++ with extensions. There are several languages with bindings to Qt with R being one through the `qtbases` and `qtbase` packages. While these packages are quite new as compared to `tcltk` and `RGtk2`, they are included here, as Qt provides arguably the richest GUI environment from within R will likely be the GUI toolkit of choice going forward.

Qt, a commercially supported package, has excellent documentation of its API and has several examples of its use using C++.

### 1.3 The qtbase package

The `qtbase` provides the primary interface between R and the underlying Qt libraries, provided the latter are installed. The Qt framework is available as a binary install from <http://qt.nokia.com/>.

The package exports very few items. The main one is an environment, `Qt`, that contains the bulk of the functionality.<sup>1</sup> The components of this environment are class objects that represent an actual C++ class, an R derivative or a C++ namespace. For example, the `QWidget` class being represented through the component `Qt$QWidget`.

These components inherit from class `RQtClass`

---

<sup>1</sup> The `Qt` object provides a binding between a library and an R object. There are others, and these all inherit from the `RQtLibrary` class which in turn inherits from `environment`. As such, these library objects can be used where environments are, such as with `ls` or `$`.

```
class(Qt$QWidget)
```

```
[1] "RQtSmokeClass" "RQtClass"      "function"
```

As well, they inherit from the functions class, as they serve as constructors for instances of the class. For example

```
w <- Qt$QWidget()
```

The `w` object has a class structure that reflects the class inheritance structure of Qt:

```
class(w)
```

```
[1] "QWidget"          "QObject"          "QPaintDevice"
[4] "UserDefinedDatabase" "environment"      "RQtObject"
```

The base class, `RQtObject`, is also an environment and the properties and methods for this instance of the Qt class that are available from within R comprise its components. For `w` the first few listed using `ls`:

```
ls(w)
```

```
[1] "setShown"          "releaseShortcut"
[3] "actions"           "setAccessibleDescription"
[5] "isWindowModified"  "size"
```

Properties and methods are accessed from the environment in the usual manners available. The most convenient extractor is the `$` operator, but `[[` and `get` will also work. (With the `$` operator at the command line, completion works.) The properties may be accessed like a component of an environment. For example, a `QWidget` has a `windowTitle` property which is used when the widget draws itself with a window. The following shows how this property can be accessed and set.

```
w$windowTitle
```

```
NULL
```

```
w$windowTitle <- "a new title"
w$windowTitle
```

```
[1] "a new title"
```

However, properties in Qt are best set through setter methods. In this case, we have the setter `setWindowTitle` available

```
w$setWindowTitle("an even newer title")
```

## 1. PROGRAMMING GUIs USING QT

---

Setter methods are typically named with the word "set" followed by the property name, all written in lower camel case, the convention Qt uses for its properties and method. (Class names are in upper camel case.)

The environment structure of the object masks the fact that the properties and methods may be defined in a parent class of the object. For example, a button widget is provided by the `QPushButton` constructor, as in

```
b <- Qt::QPushButton()
```

This too has a `windowTitle` property, but this is inherited from the fact that the `QPushButton` subclasses the `QWidget` class, as may be seen from:

```
head(class(b), n=3)
```

```
[1] "QPushButton" "QAbstractButton" "QWidget"
```

The reason this distinction is important to know, is that the documentation for the method will be found with the class where the method is defined, not in the subclass. As there is no easy way even to tell the signature of these methods, being able to consult the documentation is crucial.

### Constructors

As mentioned, the class name is the same as the constructor, but constructors may have different signatures. For example, a simple push button can be produced in several different ways:

```
b <- Qt::QPushButton()
```

Qt allows one to specify a parent object at construction time, although generally this happens when the widget is added to a layout. The child gets added to the list of children of the parent thereby creating an object heirarchy. This allows such things as the communication between components during resizing of layouts or the automatic deletion of ancestors when a parent widget is destroyed. This happens by assigning `NULL` as a parent.

```
w <- Qt::QWidget()
b <- Qt::QPushButton(parent=w)
```

In addition, there are convenience constructors. To set the text property for a button, one can pass the value to the `text` argument:

```
b <- Qt::QPushButton(text="Button text")
```

We used a named argument, but the matching is done by position and type of object.

Buttons may also have icons, for example

```
i <- Qt::QIcon(system.file("images/ok.gif", package="gWidgets"))
b <- Qt::QPushButton(icon=i, text="Ok")
```

It Qt, the class name prefaced with a tilde is the destructor for a widget, but in qtbase it suffices to assign NULL as the parent through `setParent`

```
b$setParent(NULL)
```

## Common methods for QWidgets and QObjects

The widgets we discuss in the sequel inherit many properties and methods from the base `QObject` and `QWidget` classes. The `QObject` class is the base class and forms the basis for the object heirarchy and the event processing system. The `QWidget` class is the base class for objects with a user interface. Defined in this class are several methods inherited by the widgets we discuss.

The function `qmethods` will show the methods defined for a class. It returns a data frame with variables indicating the name, return value, signature, and whether the method is protected and static. For example, for a simple button we have many methods.

```
out <- qmethods(Qt$QPushButton)
dim(out)                                # many methods
```

```
[1] 436 5
```

**Showing or hiding a widget** Widgets must have their `show` method called in order to have them drawn to the screen. This call happens through the `print` method for an object inheriting from `QWidget`, but more typically is called by Qt recursively showing the children when a top-level window is drawn. The method `raise` will raise the window to the top of the stack of windows, in case it is covered. The method `hide` will hide the widget.

A widget can also be hidden by calling its `setVisible` method with a value of `FALSE` and reshown using a value of `TRUE`. Similarly, the method `setEnabled` can be used to toggle whether a widget is sensitive to user input, including mouse events.

Only one widget can have the keyboard focus. This is changed by the user through tab-navigation or mouse clicks (unless customized, see `focusPolicy`), but can be set programatically through the `setFocus` method, and tested through the `hasFocus` method.

Qt has a number of means to notify the user about a widget when the mouse hovers over it. The `setTooltip` method is used to specify a tooltip as a string. The message can be made to appear in the status bar of a top-level window through the method `setStatusTip`.

**The size of a widget** A widget may be drawn with its own window, or typically embedded in a more complicated GUI. The size of the widget can be adjusted through various methods.

First, we can get the size of the widget through the methods `frameGeometry` and `frameSize`. The `frameGeometry` method returns a `QRect` instance, Qt's rectangle class. Rectangles are parameterized by an  $x - y$  position and two dimensions ( $x$ ,  $y$ , width and height). In this case, the position refers to the upper left coordinate and dimensions are in pixels. The convenience function `qrect` is provided to construct `QRect` instances. The `frameSize` method returns a `QSize` object with properties `width` and `height`. The `qsize` function is a convenience constructor for objects of this class.

The widget's fixed size can be adjusted by modifying the rectangle and then resetting the geometry with `setGeometry`, or directly through the same method when integer values are given for the arguments.

```
«ChangeGeometry, results=hid>e>= w <- QtQWidget()rect <- wframeGeometry
rectwidth()rectsetWidth(2 * rectwidth())wsetGeometry(rect)
```

Although the above sets the size, it does not fix it. If that is desired, the methods `setFixedSize` or `setFixedWidth` are available.

When a widget is resized, one can constrain how it changes by specifying a minimum size or maximum size. These values work in combination with the size policy of the widget. The properties `minimumSize`, `minimumWidth`, `minimumHeight`, `maximumSize`, `maximumWidth` and `maximumHeight`, and their corresponding setters, are the germane ones. How these get used is determined by the `sizePolicy` property. For example, buttons will only grow in the  $x$  direction – not the  $y$  direction due to their default size policy.

The method `update` is used to request a repainting of a widget and the method `updateGeometry` is used to notify any layout manager that the size hint has changed.

### Properties and enumerations

As mentioned, widget properties are set via setters. For example a button may be drawn “flat” to remove the typical beveling indicating it is a button. The method `setFlat` accepts a logical indicating if the button is to be “flat.”

Often there can be more than two states for a property, in which case values other than logical ones must be used. For example, the label widget (among others) has a property for how its text is aligned. For alignment there are options left, right, center, top, bottom, etc. Such values are stored in an enumeration in the `Qt$Qt` class object, or for some widgets the `RQtClass` object. An enumeration uses powers of 2 so that a single number can easily represent multiple values. A flag is a combination of enumerations. For example, alignment values are from the `AlignmentFlag` enumeration and right alignment is specified by:

```
Qt$Qt$AlignRight
```

```
Enum value: AlignRight (2)
```



Whereas, the size policy enumeration are in the `QSizePolicy` class and these are in the `Qt::QSizePolicy` object:

```
Qt::QSizePolicy::Expanding
```

```
NULL
```

Although these enumerations can be specified as integers, in qtbase they are of class `QtEnum` and have the overloaded operator `|`, to combine values, and `&` to check if a value exists in a flag. (The `|` operator essentially adds the values.) For example, aligning text in label in the upper right can be done through

```
l <- Qt::QLabel("Our text")
flag <- Qt::Qt::AlignRight | Qt::Qt::AlignTop
l$setAlignment(flag)
```

To check if a value is in the flag, we have something like:

```
as.logical(flag & (Qt::Qt::AlignRight | Qt::Qt::AlignLeft))
```

```
[1] TRUE
```

## Fonts

Fonts in Qt are handled through the `QFont` class. In addition to the basic constructor, one constructor allows the programmer to specify a family, such as `helvetica`; pointsize, an integer; weight, an enumerated value such as `Qt::QFont::Light` (or `Normal`, `DemiBold`, `Bold`, or `Black`); and whether the italic version should be used, as a logical.

For example, a typical font specification may be given as follows:

```
f <- Qt::QFont(family="helvetica", ps=12,
               weight=Qt::QFont::Bold,
               italics=TRUE)
```

For widgets, the `setFont` method can be used to adjust the font. For example, to change the font for a label we have

```
l <- Qt::QLabel("Text for the label")
l$setFont(f)
```

The `QFont` class has several methods to query the font and to adjust properties. For example, there are the methods `setFamily`, `setUnderline`, `setStrikeout` and `setBold` among others.

## Styles

Qt uses styles to provide a means to customize the look and feel of an application for the underlying operating system. Each style implements a palette

of colors to indicate the states “active” (has focus), “inactive” (does not have focus), and “disabled” (not sensitive to user input). Many widgets do not have a visible distinction between active or inactive.

The role an object plays determines the type of coloring it should receive. A palette has an enumeration of `ColorRoles`. Sample ones are `Qt::QPalette::Highlight`, to indicate a selected item, or `Qt::QPalette::WindowText` to indicate a foreground color.

These roles are used for setting the foreground or background role to give a widget a different look, as illustrated in Example 1.2.

### Style Sheets

Cascading style sheets (CSS) are used by web designers to decouple the layout and look and feel of a web page from the content of the page. Qt implements the mechanism in the `QWidget` class to customize a widget through the CSS syntax. The implemented syntax is described in the overview on stylesheets provided with Qt documentation and is not rewritten here, as it is quite readable.

To implement a change through a style sheet involves the `setStyleSheet` method. For example, to change the background and text color for a button we could have

```
b <- Qt::QPushButton("Style sheet example")
b$show()
b$setStyleSheet("QPushButton {color: red; background: white;}")
```

One can also set a background image:

```
ssheet <- sprintf("* {background-image: url(%s)}", "logo.png")
b$setStyleSheet(ssheet)
```

## 1.4 Signals

Qt in C++ uses an architecture of signals and slots to have components communicate with each other. A component emits a signal when some event happens, such as a user clicking on a button. Qt allows one to register a slot in another component (or the same) that will be passed information when the signal is emitted. The two components are decoupled as the emitter does not need to know about the receiver except through the coupling. In R, this isn’t quite the case. A function takes the role of a slot (similar then to how RGtk2 works via callbacks) and is called when the signal is emitted. The function `qconnect` does the work. For example

```
b <- Qt::QPushButton("click me")
qconnect(b, "clicked", function(clicked) print("ouch"))
b$show()
```

The signal names are defined within a class or inherited through subclasses. Sometimes the callback has arguments.<sup>2</sup> The `clicked` signal used above inherits from `QAbstractButton`, which also is a base class for check boxes. As such, this signal passes in information if the button is checked.

The optional argument `user.data` can be used to specify data to parameterize the callback. This data is passed in through the last argument.

The `qconnect` function has no return value. In `RGkt2` the return value is used to remove a callback. For Qt, the `disconnect` method can be used to remove connections with some degree of granularity, but this method is not implemented in `qtbases`. Rather, one can block all signals from being emitted with the `blockSignals` method, which takes a logical value to toggle whether the signals should be blocked.

In addition to signals, Qt widgets can also have event handlers for various events. For example, the button may have event handlers for things such as `mouseMoveEvent`. In C++ one uses virtual function (functions defined for instances), but in R these are implemented as methods of sub-classes in R. That is, you define a subclass, and then implement the desired methods, such as `mouseMoveEvent`. This will be illustrated in Example 1.2.

## 1.5 Defining Classes and Methods

The `qtbases` package allows the R user the ability to define classes and subclasses to extend the features of Qt, as deemed necessary. Classes are related to the constructor that produces an instance of the class. In R classes are implemented as functions along with static methods in an environment. This is done with a bit of R voodoo and carried out when the base constructor is defined.

The `qsetClass` creates a new subclass and defines the constructor. The signature includes the arguments `x` to specify the name of the new class; `parent` to specify the class the new class will be a subclass of (for example, `Qt$QWidget` – the class, not the constructor, so no parentheses); `constructor` to specify the function used for construction; and a `where` argument to override where the class is defined. When `qsetClass` is called, a variable is assigned into the global environment (with scoping similar to `methods::setClass`).

In the introductory example we saw this minimal use of `qsetClass`.

```
qsetClass("dateValidator", Qt$QValidator, function(parent = NULL) {
  super(parent)
})
```

Here we see `dateValidator` is the new class name, a sub-class of the `QValidator` class. The constructor has a single call to `super`. The `super` function does not exist outside the scope of a constructor or a method. Within a constructor,

---

<sup>2</sup>One advantage of the signal-slot architecture is the type-checking of arguments.

super invokes the constructor of the parent (super) class, with the given arguments. Whereas, within a method implementation, super will call a method (named in the first parameter) in the parent class.

Within the body of the constructor and a method, the variable `this` is a reference to the instance of the class and the inherited method names are also attached, so need not be referenced through the `$` notation.

To define a method for a class the `qsetMethod` function is used. This overrides the virtual method for the class. The signature includes `name`, for the new name; `class`, for the class being extended; and `FUN` to define the method. The dispatch happens on the class and method name, not on the signature of the method. In addition, there is an argument `access` to specify if the method is "public" (the default), "protected", or "private".

Within the method, the special `super` function can be used to call the next method, if the sub class overrides a method. An example is in Example 1.2. The basic call looks like `super(meth_name, arg1, arg2, ...)`.

### Example 1.1: A “error label”

A common practice when validation is used for text entry is to have a “buddy label.” That is an accompanying label to set an error message. As Qt uses “buddy” for something else, we call this an “error label” below. We show how to implement such a widget in `qtbases` where we subclass the single-line text edit widget constructed by `QLineEdit`. We begin by defining our subclass and constructor

```
qsetClass("ErrorLabel", Qt$QLineEdit,
         function(text, parent=NULL, message="") {
           super(parent)

           this$widget <- Qt$QWidget()           # for attaching
           this$error <- Qt$QLabel()             # set height=0
           this$error$setStyleSheet("* {color: red}") # set color

           lyt <- Qt$QGridLayout()               # layout
           lyt$setVerticalSpacing(0)
           lyt$addWidget(this, 0, 0, 1, 1)
           lyt$addWidget(error, 1, 0, 1, 1)
           this$widget$setLayout(lyt)

           if(nchar(message) > 0)
             setMessage(message)
           else
             setErrorHeight(FALSE)
           if(!missing(text)) setText(text)
         })
```

In addition to the call to `super`, we define a `QWidget` instance to contain the line edit widget and its label. These are placed within a grid layout. The use of `this` refers to the object we are creating. The new method `setErrorHeight` is used to flatten the height of the label when it is not needed and is defined below. The final line sets the initial text in the line edit widget. The R environment where `setText` is defined is extended by the environment of this constructor, so no prefix is needed in the call.

The widget component is needed to actually show the widget. We create an accessor method for this

```
qsetMethod("widget", ErrorLabel, function() widget)
```

```
[1] "widget"
```

We extend the API of the line edit widget for this sub class to modify the message. We define three methods, one to get the message, one to set it and a convenience function to clear the message.

```
qsetMethod("message", ErrorLabel, function() error$text)
qsetMethod("setMessage", ErrorLabel, function(msg="") {
  if(nchar(msg) > 0)
    error$setText(msg)
  setErrorHeight(nchar(msg) > 0)
})
qsetMethod("clear", ErrorLabel, function() setMessage())
```

Finally, we define the method to set the height of the label, so that when there is no message it has no height. We use a combination of setting both the minimum and maximum height.

```
qsetMethod("setErrorHeight", ErrorLabel, function(do.height=FALSE)
{
  if(do.height) {
    m <- 18; M <- 100
  } else {
    m <- 0; M <- 0
  }
  error$setMinimumHeight(m)
  error$setMaximumHeight(m)
})
```

To use this widget, we have the extra call to `widget()` to retrieve the widget to add to a GUI. In the following, we just show the widget.

```
e <- ErrorLabel()
w <- e$widget()           # get widget to show
w$show()                 # to view widget
e$setMessage("A label")   # opens message
e$clear()                 # clear message, shrink space
```

## 1.6 Drag and drop

Qt has native support for basic drag and drop activities for some of its widgets, such as text editing widgets, but for more complicated situations such support must be programmed in. The toolkit provides a clear interface to allow this.

A drag and drop event consists of several stages: the user selects the object that initiates the drag event, the user then drags the object to a target, and finally a drop event occurs. In addition, several decisions must be made, e.g., will the object “move” or simply be copied. This is determined by XXX. Or, what kind of object will be transferred (an image? text?, ...) etc. The type is specified using a standard MIME specification.

**Initiating a drag and drop source** When a drag and drop sequence is initiated, the widget receiving the mouse press event needs to set up a `QDrag` instance that will transfer the necessary information to the receiving widget. In addition, the programmer specifies the type of data to be passed, as an instance of the `QMimeData` class. Finally, the user must call the `exec` method with instructions indicating what happens on the drop event (the supported actions) and possibly what happens if no modifier keys are specified. These are given using the enumerations `CopyAction`, `MoveAction`, or `LinkAction`. This is all specified in a new method for `mousePressEvent`, so must be done in a subclass of the widget you wish to use.

**Creating a drop target** The application must also set up drop sources. Each source has its method `setAcceptDrops` called with a `TRUE` value. In addition, one must implement several methods so again, a subclass of the desired widget is needed. Typically one implements at a minimum a `dropEvent` method. This method has an `QDropEvent` instance passed in which has the method `mimeData` to get the data from the `QDrag` instance. This data has several methods for querying the type of data, as illustrated in the example. If everything is fine, one calls the event's `acceptProposedAction` method to set the drop action. One can also specify other drop actions.

Additionally, one can implement methods for `dragMoveEvent` and `dragLeaveEvent`. In the example, the move and leave event adjust properties of the widget to indicate it is a drop target.

### Example 1.2: Drag and drop

We will use sub classes of the label class to illustrate how one implements basic drag-and-drop functionality. Our treatment follows the Qt tutorial on the subject. We begin by setting up a label to be a drag target.

```
qsetClass("DragLabel", Qt$QLabel, function(text="", ...) {
```

```
parent(...)
setText(text)
})
```

Class 'R::.GlobalEnv::DragLabel' with 371 public **methods**

The main method to implement for the sub class is `mousePressEvent`. The argument `e` contains event information for the mouse, we don't need it here. We have the minimal structure here: implement mime data to pass through, set up a `QDrag` instance for the data, then call the `exec` method to initiate.

```
qsetMethod("mousePressEvent", DragLabel, function(e) {
  md <- Qt$QMimeData()
  md$setText(text)

  drag <- Qt$QDrag(this)
  drag$setMimeData(md)

  drag$exec(Qt$Qt$CopyAction | Qt$Qt$MoveAction, Qt$Qt$CopyAction)
})
```

[1] "mousePressEvent"

Implementing a label as a drop target is a bit more work, as we customize its appearance. Our basic constructor follows:

```
qsetClass("DropLabel", Qt$QLabel, function(text="", ...) {
  parent(...)

  setText(text)
  setAcceptDrops(TRUE)

  this$bgrole <- backgroundRole()
  setMinimumSize(200, 200)
  setAutoFillBackground(TRUE)
  clear()
})
```

Class 'R::.GlobalEnv::DropLabel' with 371 public **methods**

We wish to override the call to `setText` above, as we want to store the original text in a property of the widget. Note the use of `super` with a method definition below to call the next method.

```
qsetMethod("setText", DropLabel, function(str) {
  this$orig_text <- str
  super("setText", str) # next method
})
```

The `clear` method is used to restore the label to an initial state. We have saved the background role and original text as properties.

```
qsetMethod("clear", DropLabel, function() {  
    setText(this$orig_text)  
    setBackgroundRole(this$bgrole)  
})
```

```
[1] "clear"
```

The enter event notifies the user that a drop can occur on this target by changing the text and the background role.

```
qsetMethod("dragEnterEvent", DropLabel, function(e) {  
    super("setText", "<Drop Text Here>")  
    setBackgroundRole(Qt$QPalette$Highlight)  
  
    e$acceptProposedAction()  
})
```

The move and leave events are straightforward. We call `clear` when the drag leaves the target to restore the widget.

```
qsetMethod("dragMoveEvent", DropLabel, function(e) {  
    e$acceptProposedAction()  
})  
qsetMethod("dragLeaveEvent", DropLabel, function(e) {  
    clear()  
    e$acceptProposedAction()  
})
```

Finally, the important drop event. The following shows how to implement this in more generality than is needed for this example, as we only have text in our mime data. The `setPixmap` and `setTextFormat` methods for labels will be discussed in Section.

```
qsetMethod("dropEvent", DropLabel, function(e) {  
    md <- e$mimeTypeData()  
  
    if(md$hasImage()) {  
        setPixmap(md$imageData())  
    } else if(md$hasHtml()) {  
        setText(md$html)  
        setTextFormat(Qt$Qt$RichText)  
    } else if(md$hasText()) {  
        setText(md$text())  
        setTextFormat(Qt$Qt$PlainText)  
    } else {  
        setText("No match") # replace ...  
    }  
  
    setBackgroundRole(this$bgrole)  
    e$acceptProposedAction()  
})
```



Drag and drop

---

| })



## Layout managers

Qt provides a set of classes to facilitate the layout of child widgets of a component. These layout managers, derived from the `QLayout` class, are tasked with positioning of the child widgets, allocation of size to the child widgets, updating size when the parent is resized or when child widgets are hidden or removed. Unlike GTK+, where this management is tied to a container object, Qt decouples the layout from the widget.

In this chapter we discuss how to program GUI layouts. An alternative would be to use the Qt Designer application to specify the layout. We begin with an example that shows many of the different types of layouts.

### Example 2.1: Using layout managers to mock up an interface

This example illustrates how to layout a somewhat complicated GUI by hand using a combination of different layout managers provided by Qt. Figure 2.1 shows a screenshot of the finished layout.

Qt provides the standard layouts for box layouts and grid layouts, in addition there are notebook containers and special layouts for forms, as seen in the following. Our GUI is layed out from the outside in. The first layout used is a grid layout which will hold three main areas: one for table (we use a label for now), one for a notebook, a layout to hold some buttons.

A `QWidget` instance can hold one layout specified by the `setLayout` method. We use widget for a top-level window and specify a grid layout.

```
w <- Qt$QWidget()
w$setWindowTitle("Layout example")
gridLayout <- Qt$QGridLayout()
w$setLayout(gridLayout)
```

Here we define the two main widgets and the layout for our buttons.

```
tableWidget <- Qt$QLabel("Table widget") # for now
nbWidget <- Qt$QTabWidget()
buttonLayout <- Qt$QHBoxLayout()
```

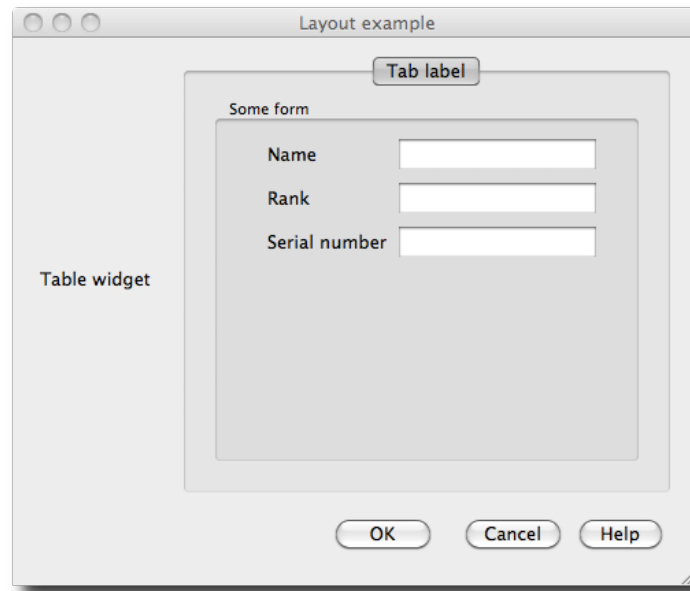


Figure 2.1: A mock GUI illustrating various layout managers provided by Qt.

Grid layouts have two main methods, `addWidget` which is inherited from a base `QLayout` class, and `addLayout`. In addition, we specify what part of the grid the child widget or layout will occupy through a 0-based row and column and spanning directions for both row and columns.

```
gridLayout$addWidget(tableWidget, row=0, column=0,  
                      rowspan=5, colspan=1)  
gridLayout$addWidget(nbWidget, 0, 1, 5, 4)  
gridLayout$addLayout(buttonLayout, 5, 0, 1, 5)
```

Buttons are added to our box layout through the `addWidget` method. In this case, we want to push the buttons to the right side of the GUI, so we first add a stretch. Stretches are specified by integers. Unallocated space is given first to widgets that have a non-zero stretch factor. We also set spacing of 12 pixels between the "OK" and "Cancel" buttons.

```
b <- sapply(c("OK", "Cancel", "Help"), function(i) Qt$QPushButton(i))  
buttonLayout$addStretch(1L)  
buttonLayout$addWidget(b$OK)  
buttonLayout$addSpacing(12L)  
buttonLayout$addWidget(b$Cancel)  
buttonLayout$addWidget(b$Help)
```

For our notebook widget we add pages through the `addTab` method. We pass in the child widget and a label below. In addition, we set a tooltip for the tab label to give more feedback to the user.

```
nbPage <- Qt$QWidget()
nbWidget$addTab(nbPage, "Tab label")
nbWidget$setTabToolTip(0, "A notebook page with a form")
```

We wish to layout a form inside of the notebook tab, but first create a framed widget using a `QGroupBox` widget. This widget allows us to easily specify a title. We add this to the page using a box layout.

```
f <- Qt$QGroupBox()
f$setTitle("Some form")
lyt <- Qt$QHBoxLayout()
nbPage$setLayout(lyt)
lyt$addWidget(f)
```

The form layout allows us to layout standardized forms where each row contains a label and a widget. This could be done with a grid layout, but using the form layout makes it easier for Qt to style the page as appropriate for the underlying operating system. For illustration purposes, we set the horizontal alignment of the widget and the left align the label.

```
formLayout <- Qt$QFormLayout()
f$setLayout(formLayout)
## can override defaults:
formLayout$setFormAlignment(Qt$Qt$AlignHCenter)
formLayout$setLabelAlignment(Qt$Qt$AlignLeft)
```

Our form template just uses 3 line edit widgets. The `addRow` method makes it easy to specify the label and the widget.

```
l <- sapply(c("name", "rank", "snumber"), function(i) Qt$QLineEdit())
formLayout$addRow("Name", l$name)
formLayout$addRow("Rank", l$rank)
formLayout$addRow("Serial number", l$snumber)
```

Finally, we set the minimum size for our GUI and call `show` on the top-level widget.

```
w$setMinimumSize(width=500, height=400)
w$show()
```

## 2.1 Box layouts

Box layouts arrange child widgets by packing in values horizontally (from left to right although right to left is possible) or vertical. The `QHBoxLayout` constructor provides a horizontal layout whereas `QVBoxLayout` provides a

vertical one. Both of these classes subclass the `QBoxLayout` class where most of the functionality is documented. The `direction` property specifies how the layout is done. By default, this is left to right or top to bottom, but can be set (e.g., using `Qt::LeftToRight`).

Child widgets are added to a box container through the `addWidget` method. The basic call specifies just the child widget, but one can specify an integer value for `stretch` and an alignment enumeration.

How size gets allocated to the widgets managed by a layout depends on many things. The Qt documentation for layouts described this as follows. A widget has methods `sizeHint` to provide the needed size for the widget. The default policy is that this is the preferred size, but it can be resized. One can change the policy through the `setSizePolicy` method. One sets the value for the horizontal and vertical sizing. Values are taken from the `QSizePolicy` enumeration with possible values of "Fixed" (`Qt::QSizePolicy::Fixed`) for using the size hint; "Minimum", for size hint being minimal and sufficient; "Maximum", for the size hint being the maximal size; "Preferred"; "Expanding", for able to expand to fill the space; "Ignored", to ignore the size hint and give the widget all the possible space. For example, a button, by default, will expand in the horizontal direction unless the size policy is adjusted:

```
b <- Qt::QPushButton("Stop the expansion")
b$setSizePolicy(Qt::QSizePolicy::Fixed, Qt::QSizePolicy::Fixed)
```

Stretch factors are used to proportionally allocate space to widgets when they expand. When more than one widget can expand, how the space gets allocated is determined by their stretch factors (for box layouts set by the `addWidget` method). Widgets with a stretch factor of 0 get extra space only if no other widgets have a positive stretch factor. Of those, one with an expanding size policy get allocated space first.

If after this allocation, widgets haven't received their minimal size request, then space is allocated to do so. Then if a widget is allocated more than its maximal size, the allocation is shrunk.

When a widget is allocated additional space, the child can have its alignment adjusted. The enumeration `Alignment` specifies values "Left" (`Qt::AlignLeft`), "Right", "HCenter" (horizontal center), "Justify", "Top", "Bottom", "VCenter" (vertical center), and "Center". Values may be "or"ed together.

In addition to adding child widgets, one can nest child layouts through `addLayout`.

The child widgets are indexed (0-based). A count of child widgets is returned by `count`, and individual widget is retrieved by calling the `widget` method on the return value of the layout's `itemAt` method. The `insertWidget` can be used to insert a widget, with arguments similar to `addWidget`. Its initial argument is an integer specifying the index. All child widgets with this index or higher have their index increased by 1.

**Removing a child** Qt provides the methods `removeItem` and `removeWidget` to remove an widget from a layout. Once removed from one layout, these may be reparented if desired, or destroyed. This is done by setting the widget's parent to `NULL` using `setParent`.

**Spacing** The space between two children is controlled through the `setSpacing` method. This sets the common width in pixels, which can be adjusted individually through the `addSpacing` method. The margin area around all the children can be adjusted with the `setContentsMargins` method, although this is often specified through the style.

**Springs and Struts** A stretchable blank widget can be added through the `addStretch` method, where an integer is specified to indicate the stretch factor. If no other widgets have a stretch specified then this widget will take all the non-requested space. A strut (`addStrut`) can be specified to restrict to a minimum height.

```
w <- Qt$QWidget()
w$setWindowTitle("Box container example")
g <- Qt$QHBoxLayout()
w$setLayout(g)
g$addWidget(Qt$QLabel("left"))
g$addStretch(1L)
g$addWidget(Qt$QLabel("middle"))
g$addSpacing(12L)
g$addWidget(Qt$QPushButton("right"))
g$count()
```

## Scrolling layouts

It may be desirable to constrain the size of a layout and allow the user to pan through its children with scrollbars. The `QScrollArea` class makes this very straightforward. The method `setWidget` is used to add the child widget. To ensure that a given widget is visible, the method `ensureWidgetVisible` is available, where the widget is passed as the argument.

```
sa <- Qt$QScrollArea()
w <- Qt$QWidget()
w$setMinimumSize(400,400)
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
for(i in rownames(state.x77)) {
  msg <- sprintf("%s had a population of %s thousand in 1977",
                 i, state.x77[i,"Population"])
  lyt$addWidget(Qt$QLabel(msg))
}
```

```
sa$setWidget(w)
sa$show()
sa$raise()
```

To ensure the value for New York (row 32 in the data set) is visible, we have:

```
widget <- lyt$itemAt(32 - 1)$widget()
sa$ensureWidgetVisible(widget)
```

### Framed Layouts

A frame with a title is a common decoration to a container often utilized to group together widgets that are naturally related. In Qt this layout is implemented through the `QGroupBox` widget. The method `setTitle` can be used to set the title, or it can be passed to the constructor. If the standard position of the title determined from the style is not to the liking, it can be adjusted through the `setAlignment` method. This method takes an enumerated value from `Qt::AlignLeft`, `Qt::AlignHCenter` or `Qt::AlignRight`. The property `flat` can be set to `TRUE` to minimize the allocated space.

Group boxes have a `checkable` property that if enabled the widget will be drawn with a checkbox to control whether the children of the group box are sensitive to user input.

### Separators

The `QGroupBox` widget provides a framed area to separate off related parts of GUI. Sometimes, just a separating line is all that is desired. There is no separate separator widget in Qt, unlike GTK+. However, the `QFrame` class provides a general method for framed widgets that can be used for this purpose with the appropriate settings.

The frame shape can be a box or others types but for this purpose a line is desired. The enumerations `Qt::FrameHLine` or `Qt::FrameVLine` can be passed to the method `setFrameShape` to specify a horizontal or vertical line. Its appearance can be altered by `setFrameShadow`. A value of `Qt::FrameSunken` or `Qt::FrameRaised` is suitable.

## 2.2 Grid Layouts

The `QGridLayout` class provides a grid layout for aligning its child widgets into rows and columns.

The `addWidget` method is used to add a child widget to the layout and the `addLayout` method adds a nested layout. They have similar arguments. The widget (or layout) is the first argument followed by either a row and column index, a row and column index with an alignment enumeration, a row and



column index and a row span and column span amount (defaulting to 1), or in addition an alignment enumeration.

The method `itemAtPosition` returns the `QLayoutItem` instance corresponding to the specified row and column. This has a `widget` method to find the corresponding widget attached at that row-column cell. The methods `rowCount` and `columnCount` can be used to find the size of the grid.

Removing a widget is similar for a box layout using `removeItem` or `removeWidget`.

Rows and columns are dual concepts and are so implemented. Consequently, both have similar methods differing only by the use of `column` or `row`/. We discuss columns. A column minimum width can be set through `setColumnMinimumWidth`. If more space is available to a column than requested, then the extra space is apportioned according to the stretch factors. This can be set for a column through the `setColumnStretch` method. Taking an integer value 0 or larger.

The spacing between widgets can be set in both directions with `setSpacing`, or fine-tuned with `setHorizontalSpacing` or `setVerticalSpacing`. The style may set these too wide for some tastes.

### Example 2.2: Using a grid layout

To illustrate grid layouts we mock up a GUI centered around a central text area widget. If the window is resized, we want that widget to get the extra space allocated to.

We begin by setting a grid layout for our parent widget.

```
w <- Qt$QWidget()
w$setWindowTitle("Layout example")
lyt <- Qt$QGridLayout()
w$setLayout(lyt)
```

We use the default left-alignment for labels in the following. Our first row has a label in column 1 and a text-entry widget spanning two columns.

```
lyt$addWidget(Qt$QLabel("Class:"), 0, 0)
lyt$addWidget(Qt$QLineEdit(), 0, 1, rowspan=1, colspan=2)
```

Our second row strats with a label and a combobox each spanning the default of one column.

```
lyt$addWidget(Qt$QLabel("Method:"), 1, 0)
lyt$addWidget(Qt$QComboBox(), 1, 1)
```

The third column of the second row and rest of the layout is managed by a sublayout, in this case a vertical box layout. We use a label as a stub and set a frame style to have it stand out.

```
lyt$addLayout(slyt <- Qt$QVBoxLayout(), 1, 2, rowspan=3, 1)
slyt$addWidget(l <- Qt$QLabel("Category\nSelector"))
l$setFrameStyle(Qt$QFrame$Box)
```

The text edit widget is added to the third row, second column. Since this widget will expand, we set an alignment for the label. Otherwise, the default alignment will center it in the vertical direction.

```
lyt$addWidget(Qt$QLabel("Text:"), 2, 0, Qt$Qt$AlignTop)
lyt$addWidget(l <- Qt$QTextEdit(), 2, 1)
```

Finally we add a space for information on the 4th row. Again we placed this in a box. By default the box would expand to fill the space of the two columns, but we fix this below as an illustration.

```
lyt$addWidget(l <- Qt$QLabel("More info:"), 3, 0, rowspan=1, colspan=2)
l$setSizePolicy(Qt$QSizePolicy$Fixed, Qt$QSizePolicy$Preferred)
l$setFrameStyle(Qt$QFrame$Box)
```

As it is, since there are no stretch factors set, the space allocated to each row and column would be identical when resized. To force the space to go to the text widget, we set a positive stretch factor for the 3rd row and 2nd column. Since the others have the default stretch factor of 0, this will allow that cell to grow when the widget is resized.

```
lyt$setRowStretch(2, 1)
lyt$setColumnStretch(1,1)
```

### 2.3 Form layouts

Forms can easily be generated with the grid layout, but Qt provides an even more convenient form layout (`QFormLayout`) that has the added benefit of conforming to the traditional styles for the underlying operating system. This can be used in combination with the `QDialogButtonBox`, which provides a container for buttons that also tries to maintain an appearance consistent with the underlying operating system.

To add a child widget with a label is done through the `addRow` method, where the label, specified first, may be given as a string and the widget is specified second. The first argument can also be a widget to replace the label, and the second a layout for nesting layouts. The `insertRow` method is similar only one first specifies the row number using a 0-based index. The `setSpacing` method can be used to adjust the default spacing between rows.

After construction, the widget may be retrieved through the `itemAt` method. This returns a layout item, to get the widget call its `widget` method. The `setWidget` method can be used to change a widget.

#### Example 2.3: Simple use of `QFormLayout`

The following illustrates a basic usage where three values are gathered.

```
w <- Qt$QWidget()
w$setWindowTitle("Example of a form layout")
```

```
w$setLayout(flyt <- Qt$QFormLayout())
l <- list()
flyt$addRow("mean", l$mean <- Qt$QLineEdit())
flyt$addRow("sd", l$sd <- Qt$QLineEdit())
flyt$addRow("n", l$n <- Qt$QLineEdit())
w$show(); w$raise()
```

The form style can be overridden using the `setFormAlignment` and `setLabelAlignment` methods. The Mac default is to have center aligned form with right-aligned labels, whereas the Windows default is to have left aligned labels. One can also override the default as to how the fields should grow when the widget is resized (`setFieldGrowthPolicy`).

```
flyt$setFormAlignment(Qt$Qt$AlignLeft | Qt$Qt$AlignTop)
flyt$setLabelAlignment(Qt$Qt$AlignLeft);
```

## 2.4 Notebooks

The notebook layout is provided by the widget `QTabWidget`. This is not a layout, rather a notebook page consists of a label and widget. Of course, you can use a layout for the widget.

Pages are added through the method `addTab`. One can specify a widget, a widget and label or a widget, icon and label. Pages are inserted by index with the `insertTab` method.

Tabs allow the user to select pages, and in Qt can be customized. The text for a tab is adjusted through `setTabText` and the icon through `setTabIcon`. These use a 0-based index to refer to the tab. A tooltip can be added through `setTabToolTip`. The tabs will have close buttons if the property `tabsClosable` is `TRUE`. One connects to the `tabCloseRequested` signal to close the tab. The tab position is adjusted through the `setTabPosition` method with enumerated values such as `Qt$QTabWidget$North`. Call `isMovable` with `TRUE` allows the pages to be reorganized by the user.

When there are numerous tabs, the method `setUsesScrollButtons` can indicate if the widget should expand to accommodate the labels or add scroll buttons.

The current tab is adjusted through the `currentIndex` property. The actual widget of the current tab is returned by `currentWidget`. To remove a page the `removeTab` is used, where tabs are referred to by index.

### Example 2.4: A tab widget

A simple example follows. First the widget is defined with several properties set.

```
nb <- Qt$QTabWidget()
nb$setTabsClosable(TRUE)
nb$setMovable(TRUE)
```

```
nb$setUsesScrollButtons(TRUE)
```

We can add pages with a label or with a label and an icon:

```
nb$addTab(Qt$QPushButton("page 1"), "page 1")
icon <- Qt$QIcon("small-R-logo.jpg")
nb$addTab(Qt$QPushButton("page 2"), icon, "page 2")
for(i in 3:10) nb$addTab(Qt$QPushButton(i), sprintf("path %s", i))
```

The close buttons put out a request for the page to be closed, but do not handle directly. Something along the lines of the following is then also needed.

```
qconnect(nb, "tabCloseRequested", function(index) {
  nb$removeTab(index)
})
```

### QStackedWidget

The `QStackedWidget` is provided by Qt to hold several widgets at once, with only one being visible. Similar to a notebook, only without the tab decorations to switch pages. For this widget children are added through the `addWidget` method and can be removed with `removeWidget`. The latter needs a widget reference. The currently displayed widget can be found from `currentWidget` (which returns `NULL` if there are no child widgets). Alternatively, one can refer to the widgets by index, with `count` returning their number, and `currentIndex` the current one and `indexOf` returns the index of the widget specified as an argument.

## 2.5 Paned Windows

Split windows with handles to allocate space are created by `QSplitter`. There is no limit on the number of child panes that can be created. The default orientation is horizontal, set the orientation property using `Qt$Qt$Vertical` to change this.

Child widgets are added through the `addWidget` method. These widgets are referred to by index and can be retrieved through the `widget` method.

The `moveSplitter` method is not implemented, so programmatically moving the a splitter handle is not possible.

```
sp <- Qt$QSplitter()
sp$addWidget(Qt$QLabel("One"))
sp$addWidget(Qt$QLabel("Two"))
sp$addWidget(Qt$QLabel("Three"))
sp$show()
sp$raise()
sp$setOrientation(Qt$Qt$Vertical)
```

```
sp$widget(0)$text
```

```
[1] "One"
```

## 2.6 Main windows

In Qt the `QMainWindow` class provides a widget for use as the primary widget in an interface. Although it is a subclass of `QWidget` – which we have used in our examples so far as a top-level window – the implementation also has preset areas for the standard menu bars, tool bars and status bars. As well, there is a possible dock area for “dockable widgets.”

A main window has just one central widget that is specified through the `setCentralWidget` method. The central widget can then have a layout and numerous children. The title of the window is set from the inherited method `setWindowTitle`.

### Actions

The menubars and toolbars are representations of collections of actions, defined through the `QAction` class. As with other toolkits, an action encapsulates a command that may be shared among parts of a GUI, in this case menu bars, toolbars and keyboard shortcuts. In Qt the action can hold the label (`setText`), an icon (`setIcon`), a status bar tip (`setStatusTip`), a tool tip (`setToolTip`), and a keyboard shortcut (`setShortcut`). The text and icon may be set at construction time, in addition to using the above methods.

Actions inherit the `setEnabled` method to toggle whether an action is sensitive to user input.

Actions emit a triggered signal when activated, sending to the callback a boolean for checked which applies for buttons.

**Keyboard shortcuts** A keyboard shortcut use a `QKeySequence` to bind a key sequence to the command. Key sequences can be found from the standard shortcuts provided in the enumeration `Qt::QKeySequence`, for example

```
Qt::QKeySequence::Cut
```

```
Enum value: Cut (8)
```

This value (or more simply the “Cut” string) can be passed to the constructor to create the shortcut.

Using these standard shortcuts ensures that the keyboard shortcut is the standard one for the underlying operating system. Alternatively, custom shortcuts can be used, such as

```
Qt::QKeySequence("Ctrl-X, Ctrl-C")
```

`QKeySequence` instance

This shows how a modifier can be specified (from "Ctrl") a key (case insensitive), and how a comma can be used to create multi-key shortcuts.

For buttons and labels, a shortcut key can be specified by prefixing the text with an ampersand &, such as

```
button <- Qt$QPushButton("&Save")
```

Then the shortcut will be Alt-S.

The shortcut event occurs when the shortcut key combination is pressed in the appropriate context. The default context is when the widget is a child of the parent window, but this can be adjusted through the method `setShortcutContext`.

**Checkable actions, and action groups** Actions can be set checkable through the `setCheckable` method. When in a checked state, the `checked` property is `TRUE`. When a checkable action is checked the `toggled` signal is emitted, the argument `checked` passes in the state.

A `QActionGroup` can be used to group together checkable actions so that only one is checkable (like radio buttons). To use this, you create an instance and use the `addAction` to link the actions to the group.

### Example 2.5: Creating an action

To create an action, say to "Save" an object requires a few steps. It is recommended that the main constructor is passed the parent widget the action will apply within.

```
parent <- Qt$QMainWindow()  
saveAction <- Qt$QAction("Save", parent)
```

We could also pass the icon to the constructor, but instead set the icon, a shortcut, a tooltip, and a statusbar tip through the action's methods.

```
saveAction$setShortcut(Qt$QKeySequence("Save"))  
iconFile <- system.file("images/save.gif", package="gWidgets")  
saveAction$setIcon(Qt$QIcon(iconFile))  
saveAction$setToolTip("Save the object")  
saveAction$setStatusTip("Save the current object")
```

The action encapsulates a command, in this case we have a stub:

```
qconnect(saveAction, "triggered", function(checked) print("Save object"))
```

## Menubars

Main windows may have a menubar. This may appear at the top of the window, or the menubar area on Mac OS X. Menubars are instances of `QMenuBar` which provides access to list of top-level submenus.

These submenus are added through `addMenu`, where a string with a possible shortcut are specified to label the menu. A `QMenu` instance is returned. To submenus one can add

1. Nested submenus through the `addMenu` method,
2. An action through the `addAction` method, or
3. a separator through `addSeparator`.

Actions may be removed from a window through `removeAction`, but usually menu items are just disabled if their command is not applicable.

### Example 2.6: Menu items

In a data editor application, one might imagine a menu item for coercion of a chosen column from one type to the next. In the following, we assume we have a function `colType` that returns the column type of the selected column or NA if no column is selected. We begin by making a menu bar, and a “Data” menu item. To this we add a few actions, and then a “Coerce” submenu. In the submenu, we use an action group so that only one type can be checked at a time. Actions must be added to both the action group and the submenu.

```
mb <- Qt$QMenuBar() ## or parent$menuBar()
menu <- mb$addMenu("Data")
#
menu$addAction(a <- Qt$QAction("Apply Function...", parent))
qconnect(a, "triggered", function() cat("apply ..."))
menu$addAction(a <- Qt$QAction("Relevel Factors...", parent))
qconnect(a, "triggered", function() cat("relevel ..."))
#
menu$addSeparator()
#
cmenu <- menu$addMenu("Coerce")
aList <- sapply(c("character", "factor", "numeric"), function(i) {
  a <- Qt$QAction(i, parent)
  a$setCheckable(TRUE)
  qconnect(a, "toggled", function(checked) print(i)) ## stub
  a
})
actionGroup <- Qt$QActionGroup(w)
QT <- sapply(aList, function(i) actionGroup$addAction(i))
QT <- sapply(aList, function(i) cmenu$addAction(i))
```

In the application, we might include logic to update the menu items along the lines of the following. If no column type is available (no column is selected) we disable the sub menu, otherwise we set the check accordingly. Of course, in the application we would ensure that checking the menu item updates the state in the data model through the triggered handler.

```
updateMenus <- function() {
  val <- colType()
```

```
cmenu$setEnabled(!is.na(val))
if(!is.na(val)) {
  aList[[val]]$setChecked(TRUE)
}
}
```

### Context menus

Context menus can be added to widgets using the same `QMenu` widget (not `QMenuBar`). The `popup` method will cause the menu to popup, but it needs to be told where. (The `exec` method will also popup a menu, but blocks other input.) The location of the popup is specified in terms of global screen coordinates, but typically the location known is in terms of the widgets coordinates. (For example, the point (0,0) being the upper-left corner of the widget.) The method `mapToGlobal` will convert for you. Position is in terms of a `QPoint` instance, which can be constructed or may be returned by an event handler. We illustrate both in the example.

Initiating the popup menu can be done in different ways. In the example below, we first show how to do it when a button is pressed. More natural ways are to respond to right mouse clicks, say. These events may be found within event handlers, say the `mousePressEvent` event. (The `QMouseEvent` object passed in has a `button` method that can be checked.) However, the operating system may provide other means to initiate a popup. Rather than program these, Qt provides the `contextMenuEvent`. We can override that in a subclass, as illustrated in the example.

### Example 2.7: Popup menus

We imagine a desire to popup possible function names that complete a string. Such suggestions are computed from a function in the `utils` package. We show how to offer these in a popup menu we do this for a button press (not the most natural case):

```
b <- Qt$QPushButton("Completion example")
qconnect(b, "pressed", function(...) {
  ## compute popup
  popup <- Qt$QMenu()
  comps <- utils::matchAvailableTopics("mean")
  l <- sapply(comps, function(i) popup$addAction(Qt$QAction(i, b)))
  popup$popup(b$mapToGlobal(Qt$QPoint(0,0)))
})
```

More naturally, we might want this menu to popup on a right mouse click in a line edit widget. To implement that, we define a subclass and reimplement the `contextMenuEvent` method. We use the `globalPos` method of the passed through event to get the appropriate position.



```

qsetClass("popupmenuexample", Qt$QLineEdit,
          function(parent=NULL) super(parent) )
#
qsetMethod("contextMenuEvent", popupmenuexample, function(e) {
  popup <- Qt$QMenu()
  comps <- utils::matchAvailableTopics(this$text)
  if(length(comps) > 10)
    comps <- comps[1:10]                # trim if large
  sapply(comps, function(i) {
    a <- Qt$QAction(i, this)
    qconnect(a, "triggered", function(...) this$setText(i))
    popup$addAction(a)
  })
  popup$popup(e$globalPos())
})
b <- popupmenuexample()

```

## Toolbars

Toolbars, giving easier access to a collection of actions or even widgets, can be readily added to a main window. The basic toolbar is an instance of the `QToolBar` class. Toolbars are added to the main window through the `addToolBar` method.

To add items to a toolbar we have:

1. `addAction` to add an action,
2. `addWidget` to embed a widget into the toolbar,
3. `addSeparator` to place a separator between items.

Actions can be removed through the `removeAction` method. The method `actions` will return a list of actions.

Toolbars can have a few styles. The orientation can be horizontal (the default) or vertical. The `setOrientation` method adjusts this with values specified by `Qt$Qt$Horizontal` or `Qt$Qt$Vertical`. The toolbuttons can show a combination of text and/or icons. This is specified through the method `setToolButtonStyle` with values taken from the `ToolButtonStyle` enumeration. The default is icon only, but one could use, say, `Qt$Qt$ToolButtonTextUnderIcon`.

### Example 2.8: A simple toolbar

The following illustrates how to put in toolbar items to open and save a file. We suppose we have a function `getIcon` that returns a `QIcon` instance from a string.

We define a top-level window to hold our toolbar and be the parent for our actions that will be placed in the toolbar. We store them in a list for ease in manipulation at a later time in the program.

```

w <- Qt$QMainWindow()

```

```
a <- list()
a$open <- Qt$QAction("Open", w)
a$open$setIcon(getIcon("open"))
a$save <- Qt$QAction("Save", w)
a$save$setIcon(getIcon("save"))
```

We define our toolbar, set its button style and then add to top-level window in the next few commands.

```
tb <- Qt$QToolBar()
tb$setToolButtonStyle(Qt$Qt$ToolButtonTextUnderIcon)
w$addToolBar(tb)
```

Finally, we add the actions to the toolbar.

```
sapply(a, function(i) tb$addAction(i))
```

### Statusbars

Main windows have room for a statusbar at the bottom of the window. The status bar is used to show programmed messages as well as any status tips assigned to actions.

A statusbar is an instance of the `QStatusBar` class. One may be added to a main window through the `setStatusBar` method. For some operating systems, a size grip is optional and its presence can be adjusted through the `sizeGripEnabled` property.

Messages may be placed in the status bar of three types: *temporary* where the message stays briefly, such as for status tips; *normal* where the message stays, but may be hidden by temporary messages; and *permanent* where the message is never hidden. In addition to messages, one can embed widgets into the statusbar.

The `showMessage` method places a temporary message. The duration can be set by specifying a time in milliseconds for a second argument. Otherwise, the message can be removed through `clearMessage`.

Use `addWidget` with a label to make a normal message, use `addPermanentWidget` to make a permanent message.

### Dockable widgets

In Qt main windows have dockable areas where one can anchor widgets that can be easily detached from the main window to float if the user desires. An example use might be a toolbar or in a large GUI, a place to dock a workspace browser. The main methods are `addDockWidget` and `removeDockWidget`. Adding a dock widget requires first creating an instance of `QDockWidget` and then setting the desired widget through the dock widget's `setWidget`. Widgets may go on any side of the central widget. The position is specified through the `DockWidgetArea` enumeration, with values such as `Qt$Qt$LeftDockWidgetArea`.

Dock widgets can be stacked or arranged in a notebook like manner. The latter is done by the `tabifyDockWidget`, which moves the second argument (a dock widget) on top of the first with tabs, like a notebook, for the user to select the widget.

Floating a dock widget is initiated by the user through clicking an icon in the title bar or programmatically through the `floating` property.

### Example 2.9: Using a main widget for the layout of an IDE

This example shows how to mock up a main window similar to the one presented by the web application, R-Studio. ([rstudio.org](http://rstudio.org)). We begin by setting a minimum size and a title for the main window.

```
w <- Qt$QMainWindow()
w$setMinimumSize(800, 500)
w$setWindowTitle("Rstudio-type layout")
```

We add in a menu bar and toolbar. The actions are minimal, not including icons, commands etc. We show the file menu definitions.

```
l <- list()
mb <- Qt$QMenuBar()
w$setMenuBar(mb)
fmenu <- mb$addMenu("File")
fmenu$addAction(l$new <- Qt$QAction("New", w))
fmenu$addSeparator()
fmenu$addAction(l$open <- Qt$QAction("Open", w))
fmenu$addAction(l$save <- Qt$QAction("Save", w))
fmenu$addSeparator()
fmenu$addAction(l$quit <- Qt$QAction("Quit", w))
```

The toolbar has just a few actions added.

```
tb <- Qt$QToolBar()
w$addToolBar(tb)
tb$addAction(l$new)
tb$addSeparator()
tb$addAction(l$quit)
tb$addSeparator()
tb$addAction(l$help <- Qt$QAction("Help", w))
```

Our central widget holds two main areas: one for editing files and one for a console. As we may want to edit multiple files, we use a tab widget for that. A `QSplitter` is used to divide the space between the two main widgets.

```
centralWidget <- Qt$QSplitter()
centralWidget$setOrientation(Qt$Qt$Vertical)
w$setCentralWidget(centralWidget)
```

## 2. LAYOUT MANAGERS

---

```
fileNotebook <- Qt$QTabWidget()
fileNotebook$addTab(Qt$QLabel("File notebook"), "About")
fileNotebook$setTabsClosable(TRUE)
centralWidget$addWidget(fileNotebook)
```

Our console widget is just a stub.

```
consoleWidget <- Qt$QLabel("Console goes here")
centralWidget$addWidget(consoleWidget)
```

On the right side of the layout we will put in various tools for interacting with the R session. We place these into dock widgets, in case the user would like to place them elsewhere on the screen. Defining dock widgets is straightforward. We show a stub for a workspace browser.

```
workspaceBrowser <- Qt$QLabel("Workspace browser goes here")
wbDockWidget <- Qt$QDockWidget("Workspace")
wbDockWidget$setWidget(workspaceBrowser)
```

The workspace and history browser are placed in a notebook to conserve space. We add the workspace browser on the right side, then tabify the history browser.

```
w$addDockWidget(Qt$Qt$RightDockWidgetArea, wbDockWidget)
w$tabifyDockWidget(wbDockWidget, hbDockWidget)
```

We next place a notebook to hold any graphics produced in a dock widget. This one occupies its own space.

```
plotNotebook <- Qt$QTabWidget()
pnDockWidget <- Qt$QDockWidget("Plots")
w$addDockWidget(Qt$Qt$RightDockWidgetArea, pnDockWidget)
```

Finally, we make status bar and add a transient message.

```
sb <- Qt$QStatusBar()
w$setStatusBar(sb)
sb$showMessage("Mock-up layout for an IDE", 2000)
```

## Widgets

### 3.1 Dialogs

Qt implements the familiar set of standard dialogs and includes support to make your own. The base class for these is `QDialog`.

The message box dialog is a simple dialog that has a primary message, perhaps additional informative text and or detailed text. In addition it may have a standard icon and buttons. Qt implements these in the `QMessageBox` class, a sub class of `QDialog`.

Qt provide two ways to create a message box. There are static functions where all properties are specified as arguments to the constructor, for example "warning" (in `Qt::QMessageBox::warning`). We prefer the basic constructor, and then to set properties through methods defined for the instance, although the other approach can be simpler.

For such use, a dialog is constructed by defining an instance. The parent is used to position the new dialog. If specified, the new dialog will be placed near the parent. After specifying the desired properties, the dialog is shown. There are two ways to do so. Calling the `open` method do so in modeless way. Invoking the dialog this way does not block the user input, unlike a modal dialog which is invoked through the `exec` method. For modal dialogs, the return value may be 0 or 1 for cancel or accept, or may be the enumeration of the button pressed. Modal dialogs may be window modal (`Qt::Qt::WindowModal`), where the dialog blocks all access to the windows which are ancestors of its parent and grandparents, or application modal (`Qt::Qt::ApplicationModal`) (the default) where all window are blocked. Such values are passed `setWindowModality`. On Mac OS X, window modal dialogs appear as sheets, within the parent window. When closed, `QDialog` instances are not destroyed and can still have their methods called.

The `QMessageBox` class can be quite simply used. Here we popup a message and move one, once the dialog is dismissed.

```
dlg <- Qt::QMessageBox()  
dlg$setText("Click to dismiss")
```

### 3. WIDGETS

---

```
dlg$exec()
```

Above we use the `setText` method to adjust the primary text. Additionally we have `setInformativeText` and `setDetailedText` to adding more information and detail. The window title is adjusted through `setWindowTitle`.

An icon be set from a standard one through the `setIcon` method, with values "Noicon", "Question", "Information", "Warning", "Critical" from `QMessageBox::Icon` enumeration. Alternatively, one can specify a non-standard one through `setIconPixmap`.

The standard button is an "Ok" button, but the method `setStandardButtons` uses the enumeration `QMessageBox::StandardButton` for others. For example, "Open" (`Qt::QMessageBox::Open`), "Save", "Cancel", etc. These can be combined through the `|` operator.

When more than one button is present, a choice of default button may be made to save the user from having to move the mouse to initiate the default command. The `setDefaultButton` takes either a standard button, or a `QPushButton` instance.

A general message box then can take this form:

```
dlg <- Qt::QMessageBox()
dlg$setWindowTitle("[This space for rent]")
dlg$setText("This is the main text")
dlg$setInformativeText("This should be informative")
dlg$setDetailedText("And this is to provide\nmore detail")
dlg$setIcon(Qt::QMessageBox::Critical)
dlg$setStandardButtons(Qt::QMessageBox::Cancel | Qt::QMessageBox::Ok)
dlg$setDefaultButton(Qt::QMessageBox::Ok)
```

We may initiate it as a modal dialog as follows:

```
if(dlg$exec() == Qt::QMessageBox::Ok)
  print("A Ok")
```

### Input dialogs

The `QInputDialog` class provides a convenient means to gather an piece of information from the user. One can ask for text, text chosen from a list of values, an integer or a numeric value. The type is chosen by which methods are used to set up the input dialog. Text is done through `setTextValue`, an integer spin box is created through `setIntRange`, a numeric spin box through `setDoubleRange` and a combobox thorough `setComboBoxItems`.

Input dialogs have a "Ok" and "Cancel" button, unless an option is set to disable these. When used in a modal manner, the return value is either a 0, for cancel, or 1.

We illustrate the various styles below. First a dialog to gather text:

```
dlg <- Qt::QInputDialog()
```

```
dlg$setWindowTitle("Gather text")
dlg$setLabelText("Enter some text")
dlg$setTextValue("Initial text")
```

```
if(dlg$exec())
  print(dlg$textValue())
```

To specify an integer spin box or double is similar. Here we see an integer:

```
dlg <- Qt$QInputDialog()
dlg$setWindowTitle("Gather integer")
dlg$setLabelText("Select a value")
dlg$setIntRange(0L, 100L)
dlg$setIntStep(5L)
dlg$setIntValue(50L)
```

```
if(dlg$exec())
  print(dlg$intValue())
```

Finally, the input can be combobox

```
dlg <- Qt$QInputDialog()
dlg$setWindowTitle("Select an item")
dlg$setLabelText("Select a value")
dlg$setComboBoxItems(state.name)
dlg$setComboBoxEditable(FALSE)
```

```
if(dlg$exec())
  print(dlg$textValue())
```

However, the combobox can be adjusted, through an option, into a list view:

```
dlg$setOption(Qt$QInputDialog$UseListViewForComboBoxItems)
```

```
if(dlg$exec())
  print(dlg$textValue())
```

## QDialog

The `QDialog` class can be used to make modal or no-modal dialogs. Laying out a dialog is similar to how any window is layed out, as `QDialog` inherits for `QWidget`. In fact modeless dialogs are very similar to using a widget for the layout, although the parent is treated differently, dialogs always have top-level windows. Instead the parent is used for placment of the dialog.

What makes a dialog different is when it is modal. That is the `exec` call is used. This state may be broken by quitting the window through the window manager, or by calling the `done` method with an integer specifying the return

### 3. WIDGETS

---

value (say `Qt$QDialog$Accepted` or `Qt$QDialog$Rejected`). This closes the window and returns control. If the `Qt$Qt$WA_DeleteOnClose` attribute is set, the dialog will be deleted, otherwise it may be reused.

A simple example follows. We assume some parent window is defined as `parent`

Our dialog is defined and we adjust our window modality accordingly. Under Mac OS X, this will appear as a drop down sheet of the parent, not in a separate window.

```
dlg <- Qt$QDialog(parent)
dlg$setWindowModality(Qt$Qt$WindowModal)
dlg$setWindowTitle("A simple dialog")
```

Our dialog here is just a basic mock up. We use a horizontal box for the buttons, but in a real application would use the `QDialogButtonBox`

```
dlg$setLayout(lyt <- Qt$QVBoxLayout())
lyt$addWidget(Qt$QLabel("Layout dialog components here"))
blyt <- Qt$QHBoxLayout() # for buttons
lyt$addLayout(blyt)
```

Our buttons have callbacks that call the `done` method. We use `user.data` to pass in the dialog reference.

```
ok <- Qt$QPushButton("Ok")
cancel <- Qt$QPushButton("Cancel")
blyt$addWidget(ok)
blyt$addWidget(cancel)
qconnect(ok, "pressed", function(dlg) dlg$done(1), user.data=dlg)
qconnect(cancel, "pressed", function(dlg) dlg$done(0), user.data=dlg)

if(dlg$exec())
  print("Yes")
```

### File and Directory choosing dialogs

The file and directory chooser dialogs are created through the `QFileDialog` class. As with other dialogs there are static functions to create dialogs directly in addition to the dialog object that can be modified through its method.

The static methods `"getOpenFileName"`, `"getOpenFileNames"`, `"getExistingDirectory"`, and `"getSaveFileName"` are straightforward to use. All take as initial arguments a parent, a caption and a directory. Other arguments allow one to set a filter, say. For basic use, these are nearly as easy to use as R's `file.choose`.

To select a file anem to open we would have:

```
fname <- Qt$QFileDialog$getOpenFileName(NULL, "Open a file...", getwd())
```

If a file is selected, `fname` will contain the full path to the file, otherwise it will be `NULL`.

(For multiple selection we have



```
fnames <- Qt$QFileDialog$getOpenFileNames(NULL, "Open file(s)...", getwd())
```

and `fnames` holds a character vector of the file paths or `NULL`.)

To select a file name for saving, we have

```
fname <- Qt$QFileDialog$getSaveFileName(NULL, "Save as...", getwd())
```

And to choose a directory,

```
dname <- Qt$QFileDialog$getExistingDirectory(NULL, "Select directory", getwd())
```

**Using the constructor and its methods** Although the static functions provide most of the functionality, if desired, to modify the behaviour, one can use the regular constructor and modify it through the various methods.

The title is adjusted through `setWindowTitle` and the starting directory through `setDirectory`. For file dialogs, the default suffix can be set through `setDefaultSuffix`. The file selection mode is adjusted via `setFileMode` with the values, from `QFileDialog::FileMode`, "AnyFile", "ExistingFile", "Directory" or "ExistingFiles".

There are various ways to restrict the files being shown. The `setFilter` method limits the display to certain file types specified through the enumeration `QDir::Filter`, for example "Dirs" (`Qt$QDir$Dirs`), "files", "Modified" etc.

To specify a filter by file extension, we use a name filter. A name filter is of the form `Description (*.ext *.ext2)` (no comma) where this would match files with extensions `ext` or `ext2`. Multiple filters can be used by separating them with two semicolons. For example, this would be a natural filter for R users:

```
rfilter <- "R files (*.R .RData);; Sweave files (*.Rnw);; All files (*.*)"
```

(Such a filter can be passed as the fourth argument to `getOpenFileName`.)

When the dialog is opened through `exec` it returns a value of 0 for cancel or 1 for accept. To get the specified files, the method `selectedFiles` is used.

### Example 3.1: File dialogs

A dialog to open an existing file can be constructed along the lines of the following

```
dlg <- Qt$QFileDialog(NULL)
dlg$setWindowTitle("Choose an R file")
dlg$setDirectory(getwd())
dlg$setFileMode(Qt$QFileDialog$ExistingFiles)
dlg$setNameFilter(paste("R files (*.R .RData)",
                        "Sweave files (*.Rnw)",
                        "All files (*.*)", sep=";"))
dlg$setFilter(Qt$QDir$AllEntries | Qt$QDir$CaseSensitive)
```

### 3. WIDGETS

---

```
if(dlg$exec())  
  print(dlg$selectedFiles())
```

#### 3.2 Labels

As seen in previous example, basic labels in Qt are instances of the `QLabel` class. Labels primarily display text which can be passed to the constructor, or set via `setText`. Labels can show plain text or rich text, which allows markup in an HTML style. The label text can be cleared through `clear`. (Labels can also hold images, set by `setPixmap`.)

Label text can be adjusted through several properties `alignment`, `indent` (in pixels), `margin` (with default of 0 pixels), and `wordWrap` (with default `FALSE`).

#### 3.3 Icons and pixmaps

Icons are instances of the `QIcon` class. Icons may be read in from files or from `QPixmap` instances. Although one may, it is not necessary to define icons for different sizes and display modes (normal, disabled, active, selected), as Qt can do so automatically.

For example, we define an icon and add it to a button as follows. The call to `setEnabled` shows how Qt can draw the icon in a disabled state.

```
ok <- system.file("images/ok.gif", package="gWidgets")  
icon <- Qt$QIcon(ok)  
b <- Qt$QPushButton("Ok")  
b$setIcon(icon)  
b$setEnabled(FALSE)
```

The `QPixmap` class creates a means to represent an image on a paint device. One can create icons from pixmaps, and the `pixmap` method will return a pixmap from an icon. (The size may be specified with integers, and the mode can be set). Pixmaps can be directly made through the constructor. For example, to display a graphics file, we might have

```
png(f <- tempfile())  
hist(rnorm(100))  
dev.off()  
pix <- Qt$QPixmap()  
pix$load(f)  
l <- Qt$QLabel()  
l$setPixmap(pix)
```

```
l$show()  
l$raise()
```

The `QPixmap` class has several methods for manipulating the image.

### 3.4 Buttons

Buttons are created by `QPushButton` which inherits most of its functionality from `QAbstractButton`. A button has a `text` property for storing a label and an `icon` property to show an accompanying image.

Buttons are associated with commands. One may bind to the inherited signal `clicked`, which is called when the button is activated by the mouse, a short cut key, or a call to `clicked`. The callback receives a logical value checked if the button is “checkable.” Otherwise, `pressed` and `released` signals are emitted.

#### Button boxes

`QDialogButtonBox`

Dialogs often have a standard button placement that varies among operating systems. Qt provides the `QDialogButtonBox` class to store buttons. This class accepts the standard buttons listed in the `QDialogButtonBox::StandardButton` enumeration. Each standard button has a default role from a list of roles specified in `QDialogButtonBox::ButtonRole`, if a non-standard button is desired, then a role must be specified. The method `addButton` method is used to add a button, as specified by either a standard button or by a label and role. This method returns a `QPushButton` instance.

In a dialog, a button may be designated as the default button for a dialog. To specify a default, the button’s `setDefault` is called with a value of `TRUE`.

To get feedback, one can connect each button to the desired signal. More conveniently, the button box has signals `accepted`, which is emitted when a button with the `AcceptRole` or `YesRole` is clicked; `rejected`, which is emitted when a button with the `RejectRole` or `NoRole` is clicked; `helpRequested`; or `clicked` when any button is clicked. For this last signal, the callback is passed in the button object.

#### Example 3.2: A yes-no-help set of buttons

We use a dialog button box to hold a standard set of buttons. Figure 3.1 shows the difference in their display for two different operating systems. Below, we just illustrate how to specify callbacks based on the button’s role, but only put in stubs for their commands.

```
db <- Qt$QDialogButtonBox()
ok <- db$addButton(Qt$QDialogButtonBox$Ok)
db$addButton(Qt$QDialogButtonBox$Cancel)
## Or: db$addButton("Cancel", Qt$QDialogButtonBox$RejectRole)
db$addButton(Qt$QDialogButtonBox$Help)
#
ok$setDefault(TRUE)
#
```

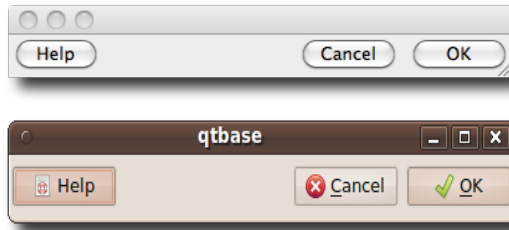


Figure 3.1: Dialog button boxes and their implementation under Mac OS X and Linux.

```
qconnect(db, "accepted", function() print("accepted"))
qconnect(db, "rejected", function() print("rejected"))
qconnect(db, "helpRequested", function() print("help"))
```

## 3.5 Checkboxes

The `QCheckBox` class implements a checkbox. Like the `QPushButton` class, class inherits methods and properties from the `QAbstractButton` class. For example, the checkbox label is associated with the `text` property.

Qt has three states for a checkbox: the obvious `Qt$Qt$Checked`, `Qt$Qt$Unchecked` states, and a third indeterminate state specified by `Qt$Qt$PartiallyChecked`. These are stored in the `checkState` for setting and getting.

In addition to the inherited signals `clicked`, `pressed`, and `released`, The signal `stateChanged` is emitted. Callbacks are passed the state as an integer.

### Groups of checkboxes

The `QButtonGroup` can be used to group together buttons, such as checkboxes, into logical units. The layout of the buttons is not managed by this widget. By default, the buttons are exclusive, like a radio button group, but this can be adjusted by passing `FALSE` to the method `setExclusive`. Buttons are added to the button group one-by-one through the method `addButton`. An optional ID can be passed in to identify the buttons, but it may be more convenient to work with the list of buttons returned by the `buttons` method.

When a button is clicked, pressed or released a signal is emitted, for example `buttonClicked` is when a button is clicked. The callback receives the button object.

#### Example 3.3: Using checkboxes to provide a filter

Filtering a data set by the levels of a factor is a natural desire. This is commonly implemented by using a checkbox group with each level assigned to

a toggle. Below we show how to do so for a cylinders variable, such is in the Cars93 data set of the MASS package.

We begin by defining the levels and a widget to store our buttons.

```
cyls <- c(3,4,5,6,8,"rotary")
w <- Qt$QGroupBox("Cylinders:")
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
```

We want to be able to select more than one button, so set the exclusive property accordingly.

```
bg <- Qt$QButtonGroup()
bg$setExclusive(FALSE)
```

Next we prepare a button for each value of cyls, add it to the layout and then the button group. Finally we initialize the buttons to all be checked.

```
sapply(seq_along(cyls), function(i) {
  button <- Qt$QCheckBox(sprintf("%s Cylinders", cyls[i]))
  lyt$addWidget(button)
  bg$addButton(button, i)
})
sapply(bg$buttons(), function(i) i$setChecked(TRUE))
```

Our simple callback to the buttonClicked signal shows how to see which buttons were checked after the button was pressed. (The buttonPressed is called before the widget state reflects the button press.)

```
qconnect(bg, "buttonPressed", function(button) {
  checked <- sapply(bg$buttons(), function(i) i$checked)
  if(any(checked))
    print(c("x", cyls[checked]))
})
```

### 3.6 Radio groups

Radio buttons are created by the QRadioButton constructor. Radio buttons, like other buttons, have a text and icon property. Radio buttons have a checked or unchecked state which can be queried with isChecked and set with setChecked.

Radio buttons are typically exclusive. To group buttons together, Qt uses all buttons that share the same parent widget.

#### Example 3.4: Using a radio group to filter

Instead of a group of checkboxes, we might also filter through an exclusive selection. The following shows how to construct a radio button group.

### 3. WIDGETS

---

First, we set the radio buttons into a list for easy manipulation. A `QButtonGroup` could also be used here, but we use an R-based approach for variety.

```
l <- list(Qt$QRadioButton("Weight < 3000"),
         Qt$QRadioButton("3000 <= Weight < 4000"),
         Qt$QRadioButton("4000 <= Weight")
        )
```

We now define a layout for our radio button widgets. The buttons are children of `w` below, so are exclusive within that.

```
w <- Qt$QGroupBox("Weight:")
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
```

We now add the widgets to the layout and set the first one to be checked.

```
sapply(l, function(i) lyt$addWidget(i))
l[[1]]$setChecked(TRUE)
```

The toggled signal is emitted twice when a button is clicked: once for when the check button is clicked, and once for the toggle of the previously checked button. Below, we condition on the value of `checked` to restrict to one call.

```
sapply(l, function(i) {
  qconnect(i, "toggled", function(checked) {
    if(checked) {
      ind <- which(sapply(l, function(i) i$isChecked()))
      print(sprintf("You checked %s.", l[[ind]]$text))
    }
  })
})
```

## 3.7 Sliders and spin boxes

Sliders and spin boxes are similar widgets used for selecting from a range of values. Sliders give the illusion of selecting from a continuum, whereas spinboxes offer a discrete choice, but underlying each is a selection from an arithmetic sequence. In Qt the two have similar method names.

### Sliders

Sliders are implemented by `QSlider` a subclass of `QAbstractSlider`, which also provides functionality for scrollbars. Sliders in Qt are for selecting from integer values. To specify the range of possible values to select from the methods `setMinimum` and `setMaximum` are used (assuming integer values).

Movement between the possible values is adjusted by `setSinglePageStep` with broader motion by `setPageStep`. (If we think of the arguments from, to, by of `seq` then these are the minimum, maximum and the single page step.) The actual value stored in the widget is found in the property value.

Sliders can be horizontal or vertical, with a `Qt::Orientation` value being passed to the `setOrientation` method. To adjust the appearance to ticks for a slider, the method `setTickPosition` is used with values drawn from the `QSlider::TickPosition` enumeration (for example, `Qt$QSlider$TicksBelow`, `"TicksLeft"` or the default `"NoTicks"`). The method `setTickInterval` is used to specify an interval between the ticks.

The signal `valueChanged` is emitted when the slider is moved. It passes back the current value to a callback.

## Spin boxes

Spin boxes are derived from `QAbstractSpinBox` which provides the base class for `QSpinBox` (for integers), `QDoubleSpinBox` and `QDateTimeEdit` (not documented here).

The methods have similar names as for sliders: `setMinimum`, `setMaximum`, and `setValue` have similar usages. The step size is provided by `setSingleStep`. The property `wrapping` can be set to `TRUE` to have the values wrap at the ends.

In Qt spinbuttons can have a prefix or suffix with the numbers. These are set by `setPrefix` or `setSuffix`.

As with sliders, the signal `valueChanged` is emitted when the spin button is changed,

### Example 3.5: A range selector

We combine a slider and spinbox to make a range selection widget, offering the speedier movement of the slider, with the finer adjustments of the spin box.

Our slider construction sets values one-by-one.

```
s1 <- Qt$QSlider()
s1$setMinimum(0)
s1$setMaximum(100)
s1$setSingleStep(1)
s1$setPageStep(5)
```

To style our slider we make it horizontal, set the tick position and interval through the following:

```
s1$setOrientation(Qt$Qt$Horizontal)
s1$setTickPosition(Qt$QSlider$TicksBelow)
s1$setTickInterval(10)
```

### 3. WIDGETS

---

The basic spin box construction is relatively straightforward. We add a "%" suffix to make it seem like we are selecting a percent.

```
sp <- Qt$QSpinBox()
sp$setMinimum(0)
sp$setMaximum(100)
sp$setSingleStep(1)
sp$setSuffix("%")
```

To link the two widgets, we define callbacks for their `valueChanged` signal updating the other widget when there is a change. This could possibly cause an infinite loop, so we check if the suggested value is not equal to the current one before updating.

```
qconnect(sp, "valueChanged", function(value) {
  if(! isTRUE(all.equal(value, sl$value)))
    sl$setValue(value)
})
qconnect(sl, "valueChanged", function(value) {
  if(! isTRUE(all.equal(value, sp$value)))
    sp$setValue(value)
})
```

#### 3.8 Single-line text

As seen in previous examples, a widget for entering or displaying a single line of text is provided by the `QLineEdit` class. The `text` property holds the current value. The text may be set as the first argument to the constructor, or through the method `setText` (provided `readOnly` is `FALSE`). Text may be inserted through the `insert` method, replacing the currently selected text or inserting at the cursor. One can programatically position the cursor by index through the `setCursorPosition` method. As is typical, the index is 0 for the left most position, 1 for between the first and second character etc. The right-most index can be found from `nchar(widget$text)`, say. The `setSelection` takes two indices to indicate the left and right bounds of the selection.

When there is a selection, the methods `hasSelectedText`, `selectedText` are applicable. If `dragEnabled` is `TRUE` the selected text may be dragged and dropped on the appropriate targets.

If desired, it is possible to mask the displayed text with asterisks (common with passwords) by setting the `echoMode` property. Value are taken from the `QLineEdit::EchoMode` enumeration, e.g., `Qt$QLineEdit$Password`. If desired, the property `displayText` holds the displayed text.

**Undo, redo** The widget keeps an undo/redo stack. The methods `modified`, `isRedoAvailable`, `isUndoAvailable` are helpful in tracking if the text has changed and undo and redo can go through the changes.



The widget emits several different signals that are of use. The `cursorPositionChanged` signal is emitted as the cursor is moved. The old and new positions are passed along. The `selectionChanged` signal is emitted as the selection is updated. The `textChanged` signal is emitted when the text is changed. Any callback is passed the new text. Similarly for `textEdited`, the difference being that this signal is not emitted when text is set by `setText`. The distinction between the signals `editingFinished` and `returnPressed` is due to the former being called only if a valid entry is given.

## Completion

Using Qt's `QCompleter` framework, a list of possible words can be presented for completion. The word list is generally specified by a model, but may also be specified as a character vector to the constructor. A simple usage is present by example.

### Example 3.6: Using completion on the Qt object

The Qt environment has many components. This example shows how completion can assist in exploring them by name. We use a form layout to arrange our two line edit widgets – one to gather a class name and one for method and property names.

```
w <- Qt$QWidget()
lyt <- Qt$QFormLayout()
w$setLayout(lyt)
lyt$addRow("Class name", c_name <- Qt$QLineEdit())
lyt$addRow("Method name", m_name <- Qt$QLineEdit())
```

The completer for the class is constructed just one. We use `ls` to list the components of the environment. We see that completions are set for a line edit widget through the `setCompleter`

```
c_comp <- Qt$QCompleter(ls(Qt))
c_name$setCompleter(c_comp)
```

The completion for the methods depends on the class. As such, we update the completion when editing is finished for the class name.

```
qconnect(c_name, "editingFinished", function() {
  cl <- c_name$text
  val <- get(cl, envir=Qt)
  if(!is.null(val)) {
    m_comp <- Qt$QCompleter(ls(val))
    m_name$setCompleter(m_comp)
  }
})
```

### 3. WIDGETS

---

Table 3.1: Table of characters and their meanings in an input mask. *Source: Qt documentation for `QLineEdit`.*

| Character | Meaning  |
|-----------|--|
| d         | ASCII alphabetic character required. A-Z, a-z.             |
| f         | ASCII alphabetic character permitted but not required.     |
| N         | ASCII alphanumeric character required. A-Z, a-z, 0-9.      |
| n         | ASCII alphanumeric character permitted but not required.   |
| X         | Any character required.                                    |
| x         | Any character permitted but not required.                  |
| 9         | ASCII digit required. 0-9.                                 |
| 0         | ASCII digit permitted but not required.                    |
| D         | ASCII digit required. 1-9.                                 |
| d         | ASCII digit permitted but not required (1-9).              |
| #         | ASCII digit or plus/minus sign permitted but not required. |
| H         | Hexadecimal character required. A-F, a-f, 0-9.             |
| h         | Hexadecimal character permitted but not required.          |
| B         | Binary character required. 0-1.                            |
| b         | Binary character permitted but not required.               |
| >         | All following alphabetic characters are uppercased.        |
| <         | All following alphabetic characters are lowercased.        |
| !         | Switch off case conversion.                                |

Use a \ to escape the special characters listed above to use them as separators.

### Masks and Validation

`QLineEdit` has various means to restrict and validate user input. The `maxLength` property can be set to restrict the number of allowed characters. To set a pattern for the possible answer, an input mask can be set through `setInputMask`. Input masks are specified through a string indicating a pattern (Table ??). For example, "999-99-9999" is for a U.S. Social Security number.

tab:qt:input-mask A illustrated in Example 1.1, Qt also implements a validation framework where the value in the widget is validated before being committed. Values can have 3 different states of validity "Acceptable" (i.e, `Qt$QValidator$Acceptable`), "Invalid", or the indeterminate "Intermediate". Validation is implemented by implementing a `validate` method for the widget. This requires implementing a subclass, as one can't add a method to one of the classes of Qt. This method gets passed the text to validate and the cursor position and returns one of the validity states.

### 3.9 Multi-line text

Multi-line text is displayed and edited through the `QTextEdit` class. This widget allows for more than plain text. It may show rich text through HTML syntax including images, lists and tables.

As a basic text editor, the widget is a view for an underlying `QTextDocument` instance. The document can be used to retrieve that document, and `setDocument` to replace it. The method `toPlainText` is used to retrieve the text as plain text with a corresponding `setPlainText` for replacing the text. Text can also be added. The `append` will append the text to the end of the document, `insertPlainText` will insert the text at the current cursor position replacing any selection, and `paste` will paste the current clipboard contents into the current cursor position. (Use `copy` and `cut` to replace the clipboards contents.) More complicated insertions can be handled through the `QTextCursor` class.

To illustrate

```
te <- Qt$QTextEdit()
te$setPlainText("The quick brown fox")
te$append("jumped over the lazy dog")
```

```
te$toPlainText()
```

```
[1] "The quick brown fox\njumped over the lazy dog"
```

**Formatting** By default, the widget will wrap text as entered. For use as a code editor, this is not desirable. The `setLineWrapMode` takes values from the enumeration `QTextEdit::LineWrapMode`, to control this. A value `Qt$QTextEdit$NoWrap` will turn off wrapping. When wrapping is enabled, one can control how with the enumeration `QTextOption::WrapMode` and the method `setWordWrap`.

The `setAlignment` method aligns the current paragraph with values from `Qt::Alignment`.

Formatting is managed by the `QTextFormat` with subclasses such as `QTextCharFormat`, For formatting at the character level, and `QTextBlockFormat` to format blocks of text. (A document is comprised of blocks containing paragraphs etc.) There are methods to adjust the alignment, font properties, margins etc.

**The text cursor** The cursor position is returned from the `position` method of the text cursor, which is found through the `textCursor` method. The position is an integer indicating the index of the cursor if the buffer is thought of as a single string. To set the cursor position one first gets the text cursor, sets its position with `setPosition`, then sets the text edits text cursor with `setTextCursor`.

For example, to move the cursor to the end can be done with

### 3. WIDGETS

---

```
n <- nchar(te$toPlainText())
cursor <- te$textCursor()
cursor$setPosition(n)
te$setTextCursor(cursor)
```

**Selections** The text cursor has `selection` method indicates if a selection exists. If it does, the method `selection` returns a `QTextDocumentFragment` which also has a method `toPlainText`. A selection is determined by the position of the cursor and an anchor. The latter is found through `anchor`. However, to move the anchor is a bit more difficult. Basically, as the name suggests, you keep an anchor in place and move the position. The `movePosition` method's second argument is one of `Qt::QTextCursor::KeepAnchor` or `Qt::QTextCursor::MoveAnchor`. When the anchor is kept in place and the cursor moved a selection is defined. The movement of the position can be specified through the `QTextCursor::MoveOperation` with values like "Start", "End", "NextWord", etc.

To set the selection to include the first three words of the text, we have:

```
cursor <- te$textCursor()
cursor$movePosition(Qt::QTextCursor::Start) ## default is move anchor
cursor$movePosition(Qt::QTextCursor::WordRight, Qt::QTextCursor::KeepAnchor, 3)
te$setTextCursor(cursor)
```

```
te$textCursor()$selection()$toPlainText()
```

```
[1] "The quick brown "
```

**Navigation** The widget has a `find` method to move the selection to the next instance of a string. The enumeration `QTextDocument::FindFlag` can modify the search with values "FindBackward", "FindCaseSesnsitively" and "FindWholeWords".

For example, we can search through a standard typesetting string starting at the cursor point for the common word "qui" as follows:

```
te <- Qt::QTextEdit(LoremIpsum) # some text
te$find("qui", Qt::QTextDocument::FindWholeWords)
te$textCursor()$selection()$toPlainText()
```

**Signals** There are several useful signals that are emitted by the widget. Several deal with changes: `cursorPositionChanged`, `selectionChanged`, `textChanged`, and `currentCharFormatChanged`. Others allow one to easily update any actions to reflect the state of the widget. These include `copyAvailable`, which passes in a boolean indicating "yes"; `redoAvailable`, which passes a boolean indicating availability; and similarly `undoAvailable`.

To get keystroke information, one can create a subclass and implement `keyPressEvent`, say.

**Example 3.7: A tabbed text editor**

This example shows how to combine the text edit with a notebook widget to create a widget for editing more than one file at a time.

We begin by defining a sub-class of `QTextEdit` so that we can define a few useful properties and method. In this example, we limit ourselves to just a few customizations of the edit widget.

```
qsetClass("TextEditSheet", Qt$QTextEdit, function(parent=NULL) {
  super(parent)

  this$nb <- NULL                # text edit notebook
  this$filename <- NULL          # name of file , tab

  setLineWrapMode(Qt$QTextEdit$NoWrap)
  setFontFamily("Courier")
})
```

The `nb` property stores the parent notebook widget allowing us to look this up when we have the sheet object.

```
qsetMethod("setNb", TextEditSheet, function(nb) this$nb <- nb)
```

The file name is used for saving the sheet and for labeling the notebook tab. We define a getter and setter. The setter is also tasked with updating the tab label and illustrates how the `nb` property is employed for this.

```
qsetMethod("filename", TextEditSheet, function() this$filename)
qsetMethod("setFilename", TextEditSheet, function(fname) {
  this$filename <- fname
  ## update tab label
  notebook <- this$nb$notebook
  ind <- notebook$indexOf(this)
  notebook$setTabText(ind, basename(fname))
})
```

Next, we define a few methods for the sheet. First, one to save the file. We use the `filename` property for the suggested name to save as. This method uses one of Qt's standard dialogs.

```
qsetMethod("saveSheet", TextEditSheet, function() {
  fname <- Qt$QFileDialog$getSaveFileName(this, "Save file as...", filename())
  if(!is.null(fname)) {
    txt <- this$toPlainText()
    writeLines(strsplit(txt, "\n")[[1]], con=fname)
  }
})
```

We will see soon, that a method to test if a given sheet is the currently visible sheet is useful. This method returns a logical by comparing the current index of the notebook with the index of the sheet.

### 3. WIDGETS

---

```
qsetMethod("isCurrentSheet", TextEditSheet, function() {
  notebook <- this$nb$notebook
  notebook$currentIndex == notebook$indexOf(this)
})
```

Finally we want to initialize a new sheet. This involves defining callbacks that update the actions in the parent notebook as appropriate. Later we define a minimal set of actions for this example and store them in the `tbactions` property of the parent notebook.

```
qsetMethod("initSheet", TextEditSheet, function() {

  qconnect(this, "redoAvailable", function(yes) {
    if(isCurrentSheet())
      this$nb$tbactions$redo$setEnabled(yes)
  })
  qconnect(this, "undoAvailable", function(yes) {
    if(isCurrentSheet())
      this$nb$tbactions$undo$setEnabled(yes)
  })
  qconnect(this, "selectionChanged", function() {
    hasSelection <- this$textCursor()$hasSelection()
    if(isCurrentSheet())
      sapply(c("cut", "copy"), function(i) this$nb$tbactions[[i]]$setEnabled(hasSelection))
  })
  qconnect(this, "textChanged", function() {
    if(isCurrentSheet())
      this$nb$tbactions$save$setEnabled(this$document()$isModified())
  })
})
```

Next, we turn to the task of defining a notebook container. In fact we subclass `QMainWindow` so that we can add a toolbar. The notebook is then a property, as are the actions. Our constructor customizes the notebook, sets up the actions and toolbar, then opens with a blank sheet.

```
qsetClass("TextEditNotebook", Qt$QMainWindow, function(parent=NULL) {
  super(parent)

  ## properties
  this$tbactions <- list()
  this$notebook <- Qt$QTabWidget()

  notebook$setTabsClosable(TRUE)
  notebook$setUsesScrollButtons(TRUE)
  qconnect(notebook, "tabCloseRequested", function(ind) notebook$removeTab(ind))
  qconnect(this$notebook, "currentChanged", function(index) {
    if(index > 0)
```

```

        updateActions()
    })
    setCentralWidget(notebook)

    initActions()
    makeToolBar()
    newSheet()
})

```

For our example, we implement a basic set of methods. This one maps down from the parent notebook widget to a sheet.

```

qsetMethod("currentSheet", TextEditNotebook, function() {
    this$notebook$currentWidget()
})

```

This method is used to open a new sheet, either a blank one or some initial text.

```

qsetMethod("newSheet", TextEditNotebook, function(title="..Scratch..", str=NULL) {

    a <- TextEditSheet()           # a new sheet
    a$setNb(this)                 # set parent notebook
    a$initSheet()                 # initialize the sheet

    if(!is.null(str))             # set text if present
        a$setPlainText(str)

    this$notebook$addTab(a, "")    # add to the notebook
    this$notebook$setCurrentIndex(this$notebook$indexOf(a))
    a$setFilename(title)          # also updates tab

})

```

To open a file in a new sheet, we use a standard dialog to get the filename to open, then call `newSheet`.

```

qsetMethod("openSheet", TextEditNotebook, function() {
    dlg <- Qt$QFileDialog(this, "Select a file...", getwd())
    dlg$setFileMode(Qt$QFileDialog$ExistingFile)
    if(dlg$exec()) {
        fname <- dlg$selectedFiles()
        txt <- paste(readLines(fname), collapse="\n")
        newSheet(fname, txt)
    }
})

```

We have several actions possible in our GUI, such as the standard cut, copy and paste. We define them for the parent notebook, but the actions

### 3. WIDGETS

---

primarily work at the sheet level. The `initActions` is called to set up the actions. We don't show the entire method, as it is repetitive, but is primarily of this type:

```
actions <- list() # hold the actions
## make an action for open. This is QTextEditNotebook instance
actions$open <- Qt$QAction("open", this)
actions$open$setShortcut(Qt$QKeySequence("Open"))
qconnect(actions$open, "triggered", function(obj) obj$openSheet(), user.data=this)
## ... etc. ...
this$tbactions <- actions
```

Whenever a new sheet is shown, the state of the actions should reflect that sheet. The `updateActions` does this task. The constructor has a callback for this method whenever the sheet changes.

```
qsetMethod("updateActions", QTextEditNotebook, function() {
  ## update actions when page changes
  a <- this$tbactions
  cur <- currentSheet()
  if(is.null(cur)) return()
  a$redo$setEnabled(FALSE)
  a$undo$setEnabled(FALSE)
  a$cut$setEnabled(cur$textCursor()$hasSelection())
  a$copy$setEnabled(cur$textCursor()$hasSelection())
  a$paste$setEnabled(cur$canPaste())
})
```

Finally, the `makeToolbar` method, which is not shown, simply maps the actions to a toolbar. Our GUI might also benefit from a menu bar, an exercise left for the reader.

**Context menus** Context menus may be added to the text edit widget, provided you create a subclass. The following example shows how to add to the standard context menu, which is available by the `createStandardContextMenu` method. The key is to redefine the `contextMenuEvent` method.

```
qsetClass("QTextEditWithMenu", Qt$QTextEdit, function(parent=NULL) {
  super(parent)
})
#
qsetMethod("contextMenuEvent", QTextEditWithMenu, function(e) {
  m <- this$createStandardContextMenu()
  if(this$textCursor()$hasSelection()) {
    curVal <- this$textCursor()$selection()$toPlainText()
    comps <- utils::matchAvailableTopics(curVal)
    comps <- setdiff(comps, curVal)
    if(length(comps) > 0 && length(comps) < 10) {
```



```
m$addSeparator() # add actions
sapply(comps, function(i) {
  a <- Qt$QAction(i, this)
  qconnect(a, "triggered", function.Checked) {
    this$insertPlainText(i)
  })
  m$addAction(a)
})
}
}
m$exec(e$globalPos())
})
te <- QTextEditWithMenu()
```



## Widgets using the MVC framework

### 4.1 Model View Controller implementation in Qt

The model-view-controller architecture adds a complexity to widgets that is paid in order to create more flexible and efficient use of resources. Keeping the model separate from the view allows multiple views for the same data. It also allows views to be more responsive when there are large data sets involved. The basic MVC architecture has a controller to act as a go-between for a model and its views. In Qt, this is different, an item in a model has a delegate that allows the user to see and perhaps edit the item's data within a view. Although custom delegates can be defined, we are content to use those provided by Qt.

We discuss in the following various views: comboboxes, list views, table views and tree views. These are all familiar. The primary difficulty with using backend models is the specification and interaction with the model. For the simplest usages, Qt provides a set of convenience classes where the items and views are coupled.

Models in Qt are derived from the `QAbstractItemModel` class, although there are standard sub-classes for list, table and hierarchical models (`QStringListModel`, `QAbstractListModel`, `QAbstractTableModel`, and `QStandardItemModel`). In addition, the `qtbase` package provides the `QDataFrameModel` constructor to provide an mapping from a data frame to an item model.

A model is comprised of items, each having an associated index. Items have associated data which may vary based on the role (Table ??) the item is playing. For example, one role describes the data for display whereas another role describes the data for editing. This allows a numeric item, to be displayed with just a few digits, but edited with all of them. Both descriptions, and others, are stored in the item's data and set through the `setData` method. For the `QDataFrameModel` constructor these roles are encoded as additional, specially named, columns in the data frame.

`tab:qt::enum:itemDataRole`

An item has several properties that are described by the `Qt::ItemFlag`

Table 4.1: Partial list of roles that an item can hold data for and the class of the data.

| Constant          | Description  |
|-------------------|--|
| DisplayRole       | How data is displayed (QString)                    |
| EditRole          | Data for editing (QString)                         |
| ToolTipRole       | Displayed in tooltip ()                            |
| StatusTipRole     | Displayed in status bar ()                         |
| SizeHintRole      | Size hint for views (QSize)                        |
| DecorationRole    | (QColor, QIcon, QPixmap)                           |
| FontRole          | Font for default delegate (QFont)                  |
| TextAlignmentRole | Alignment for default delegate (Qt::AlignmentFlag) |
| BackgroundRole    | Background for default delegate (QBrush)           |
| ForegroundRole    | Foreground for default delegate (QBrush)           |
| CheckStateRole    | Indicates checked state of item (Qt::CheckState)   |

enumeration. These indicate whether an item is selectable (`ItemIsSelectable`), editable (`ItemIsEditable`), enabled (`ItemIsEnabled`), checkable (`ItemIsCheckable`) etc. The `flags` returns the flag recording combinations of these.

## 4.2 The item-view classes

The use of a model separate from a view can be relatively complicated, as such Qt provides classes that couple a standard model and view. Defining the model is much easier. In the case of a combobox and a list view the model can be specified by a character vector.

### Comboboxes

The `QComboBox` class implements the combobox widget. The `QListWidget` class implements a basic list view. Both do similar things – allow selection from a list of possible values, although differently. As such, their underlying methods are mostly similar although their implementations have a different class inheritance. We discuss here the basic usage for comboboxes. Both widgets are views for an underlying model and we restrict ourselves, for now, to the simplest choice of these models.

By basic usage we mean setting a set of values for the user to choose from, and for comboboxes optionally allowing them to type in a new value. In the simplest use, we can define the items in their model through a character vector through the method `addItem`s. (The items must be character, so numeric vectors must be coerced. This works through a conversion into a `QStringList` object.) One can remove items by index with `removeItem`. To allow a user to enter a value not in this list, the property `editable` can be set to `TRUE`.

Once values are entered the property `currentIndex` holds the 0-based index of the selected value. This will be `-1` when no selection is made, and may be meaningless if the user can edit the values. Use `setCurrentIndex` to set the popup value by index (`-1` to leave unselected). The `count` method returns the number of items available.

The property `currentText` returns the current text. This will be a blank string if there is no current index. There is no corresponding `setCurrentText`, rather one can use the `findText` method to get the index of the specified string. There is an option to adjust how finding occurs through the enumeration `Qt::MatchFlags`.

The signal `activated` is emitted when the user chooses an item. The item index is passed to the callback. This signal responds to activation by the finishing editing of the text by the return key. The `highlighted` signal is emitted when the popup is engaged and the user mouses over an entry. For editable comboboxes, the signal `editTextChanged` is emitted after each change to the text.

#### Example 4.1: An example of one combobox updating another

This example shows how one combobox, to select a region in the U.S., is used to update another, which lists states in that region. We will use the following data frame for our data., which we split into a list.

```
df <- data.frame(name=state.name, region=state.region,
                 highlight=state.x77[,1], stringsAsFactors=FALSE)
l <- split(df, df$region)
```

Our region combobox is loaded with the state regions. When one is selected, the callback for `activated` will first remove any items in the state combobox, then add in the appropriate states. The `ind` index is used to determine which region.

```
region <- Qt$QComboBox()
region$addItem(names(l))
region$setCurrentIndex(-1) # no selection
qconnect(region, "activated", function(ind) {
  sapply(rev(seq_len(state$count)), function(i)
    state$removeItem(i-1)) # clear states
  state$addItem(l[[ind+1]]$name) # add
})
```

The state combobox shows how the `highlighted` signal can be employed. In this case, information about the highlighted state is placed in the window's title. Not really a great choice, but sufficient for this example.

```
state <- Qt$QComboBox()
qconnect(state, "highlighted", function(ind) {
  pop <- l[[region$currentText]][ind + 1, "highlight"]
  w$setWindowTitle(sprintf("Population is %s", pop))
})
```

```
    })
```

Finally, we use a form layout to organize the widgets.

```
w <- Qt$QGroupBox("Two comboboxes")
lyt <- Qt$QFormLayout()
w$setLayout(lyt)
lyt$addRow("Region:", region)
lyt$addRow("State:", state)
```

### A list widget

The `QListWidget` provides an easy-to-use widget for displaying a set of items for selection. It uses an item-based model for its data. The `QListView` widget provides a more general framework with different backend models. As with comboboxes, we can populate the items directly from a character vector through the `addItem` method. However, here we mostly focus on interacting with the widget through the item model.

The items in a `QListWidget` instance are of the `QListWidgetItem` class. New items can be constructed directly through the constructor. The first argument is the text and the optional second argument a parent `QListWidget`. If no parent is specified, the item may be added through the methods `addItem`, or `insertItem` where the row to insert is specified by index. The method `takeItem` is used to remove items by their index and `clear` will remove all of them.

`QListWidgetItems` can have their text specified at construction or through the method `setText` and optionally have an icon set through the `setIcon`. There are also methods to set a status bar tip or a tooltip.

Once an item is added to list widget it can be referenced several ways. The currently selected item is returned by `currentItem`, whereas `currentRow` returns the current row by index, and `currentIndex` returns a `QModelIndex` instance (with method `row` to get the index). As well, any item may be referenced by row index through `item` or position (say within an event handler) by `itemAt`. One can search for the items with the `findItems` method, which returns a list of items. An optional second argument uses the `Qt::MatchFlags` enumeration to adjust how matches are made, for example `Qt::MatchRegExp` to match by regular expression.

**Selection** By default, single selection mode is enabled. This can be adjusted through the `setSelectionMode` argument by specifying a value in `QAbstractItemView::SelectionMode`, such as `SingleSelection` or `ExtendedSelection`. The latter changes selection when a new item is chosen, unless the Control or Shift keys are pressed.

To retrieve the selected values, the method `selectedItems` will return the items in a list.

Setting an item to be selected is done through `setCurrentItem`. The first argument is the item, the optional second one the `QItemSelectionModel::SelectionFlag` enumerations. If specified as `Qt::QItemSelectionModel::Select` (the default) the item will be selected, but other choices are possible such as "Deselect" or "Toggle".

**Checkable items** The underlying items may be checkable. This is initiated by setting an initial check state (`setCheckState`) with a value from "Checked" (`Qt::QtChecked`), "Unchecked" or "PartiallyChecked". For example, we can populate a list widget and set the values unchecked with.

```
w <- Qt::QListWidget()
w$addItem(state.name)
sapply(1:w$count, function(i) w$item(i-1)$setCheckState(Qt::QtUnchecked))
```

Then, after checking a few we can get the state along the lines of:

```
sapply(1:8, function(i) as.logical(w$item(i-1)$checkState()))
```

```
[1] TRUE FALSE TRUE TRUE FALSE FALSE TRUE TRUE
```

This uses the fact that the enumeration for "Unchecked" is 0 and "Checked" is *w*.

**Signals** There are several signals that are emitted by the widget. Chief among them are `itemActivated`, which is emitted when a user clicks on an item or presses the activation key. The latter is what distinguishes it from the `itemClicked` signal. For capturing double clicks there is `itemDoubleClicked`. For these three, the underlying item is passed to the callback. The `itemSelectionChanged` signal is emitted when the underlying selection is changed.

#### Example 4.2: Filtering example

We illustrate the widget with a typical filtering example, where a user types in values to narrow down the available choices. We begin by setting up our widgets.

```
w <- Qt::QGroupBox("Filtering example")
lyt <- Qt::QFormLayout()
w$setLayout(lyt)
lyt$addRow("Filter:", f <- Qt::QLineEdit())
lyt$addRow("State:", lw <- Qt::QListWidget())
lyt$addRow("", b <- Qt::QPushButton("Click me"))
```

For convenience, we use a built-in data set for our choices. We populate the list widget and add a tooltip indicating the area.

```
for(i in state.name) {
  item <- Qt::QListWidgetItem(i, lw)           # populate
```

#### 4. WIDGETS USING THE MVC FRAMEWORK

---

```
item$setToolTip(sprintf("%s has %s square miles", i, state.x77[i, "Area"]))
}
```

For this example, we allow multiple selection.

```
lw$setSelectionMode(Qt$QAbstractItemView$ExtendedSelection)
```

The following callback updates the displayed items so that only ones matching the typed in string are displayed. Rather than compare each items to the matched items, we simply hide them all then unhide those that match.

```
qconnect(f, "textChanged", function(str) {
  matching <- lw$findItems(str, Qt$Qt$MatchStartsWith)
  sapply(seq_len(lw$count), function(i) lw$item(i-1)$setHidden(TRUE))
  sapply(matching, function(i) i$setHidden(FALSE))
})
```

The following shows how we can grab the selected values.

```
qconnect(b, "pressed", function() {
  vals <- sapply(lw$selectedItems(), function(i) i$text())
  print(vals)
})
```

##### Example 4.3: Combining a combobox and list widget to select a variable name

This example shows how we can combine a combobox and a list widget to select a variable name from a data frame. Here we select a value by dragging it. As such we need to define a sub-class of `QListWidget` to implement the `mousePressEvent`.

```
## A subclass of QListWidget so that we can drag values
qsetClass("DraggableListWidget", Qt$QListWidget, function(parent=NULL) {
  super(parent)
  this$df <- NULL
})
```

The property `df` holds the name of the dataframe that will be selected through a combobox. Here is a method to set the value.

```
qsetMethod("setDf", DraggableListWidget, function(df) this$df <- df)
```

For drag and drop we show how to serialize an arbitrary R object to pass through to the drop target. We pass in a list of the data frame name and the selected variable name. The method `setData` takes a MIME type (which we arbitrarily define) and a value. This value will be retrieved by the `data` method and we can then call `unserialize`.

```
qsetMethod("mousePressEvent", DraggableListWidget, function(e) {
  item <- itemAt(e$pos())
})
```



```

    val <- list(df=this$df, var=item$text())

    md <- Qt$QMimeData()
    md$setData("R/serialized-data", serialize(val, NULL))

    drag <- Qt$QDrag(this)
    drag$setMimeData(md)

    drag$exec(Qt$Qt$CopyAction | Qt$Qt$MoveAction, Qt$Qt$CopyAction)
  })

```

```
[1] "mousePressEvent"
```

With this, we know create a widget to hold the combobox and the list box. The constructor creates the widgets, lays them out, initializes the data sets then sets a handler to update the variable list when the dataframe selector does.

```

qsetClass("VariableSelector", Qt$QWidget, function(parent=NULL) {
  super(parent)

  this$dfcb <- Qt$QComboBox()
  this$varList <- DraggableListWidget()

  lyt <- Qt$QVBoxLayout()
  lyt$addWidget(dfcb)
  lyt$addWidget(varList)
  varList$setSizePolicy(Qt$QSizePolicy$Expanding, Qt$QSizePolicy$Expanding)
  setLayout(lyt)

  updateDataSets()
  qconnect(dfcb, "activated", function(ind) {
    updateVarList(dfcb$currentText)
  })
})

```

Class 'R::.GlobalEnv::VariableSelector' with 318 public **methods**

Our method to update the data frame choice is a bit convoluted as we try to keep the currently selected data frame, if possible.

```

qsetMethod("updateDataSets", VariableSelector, function() {
  curVal <- this$dfcb$currentText
  this$dfcb$clear()
  x <- ls(envir=.GlobalEnv)
  dfs <- x[sapply(x, function(i) is.data.frame(get(i, inherits=TRUE)))]
  if(length(dfs)) {
    this$dfcb$addItem(dfs)
    if(is.null(curVal) || !curVal %in% dfs) {

```

#### 4. WIDGETS USING THE MVC FRAMEWORK

---

```
        this$dfcb$setCurrentIndex(-1)
        updateVarList(NULL)
      } else {
        this$dfcb$setCurrentIndex(which(curVal == dfs))
        updateVarList(curVal)           # curVal NULL, or a name
      }
    }
  })
```

```
[1] "updateDataSets"
```

Finally, we need to update the list of variables to reflect the state of the combo box selection. Here we define a helper method to display an appropriate icon based on the class of the variable.

```
getIconFile <- function(x) UseMethod("getIconFile")
getIconFile.default <- function(x)
  Qt$QIcon(system.file("images/numeric.gif", package="gWidgets"))
getIconFile.factor <- function(x)
  Qt$QIcon(system.file("images/factor.gif", package="gWidgets"))
getIconFile.character <- function(x)
  Qt$QIcon(system.file("images/character.gif", package="gWidgets"))
```

This method populates the variable list to reflect the indicated data frame. As items are automatically drag enabled, we do not need to add anything more here, as we've implement the mousePressEvent.

```
qsetMethod("updateVarList", VariableSelector, function(df=NULL) {
  this$varList$setDf(df)
  this$varList$clear()
  if(!is.null(df)) {
    d <- get(df)
    sapply(names(d), function(i) {
      item <- Qt$QListWidgetItem(i, this$varList)
      item$setIcon(getIconFile(d[,i]))
    })
  }
})
```

### A table widget

The `QTableWidget` class provides a widget for displaying tabular data in an item-based approach, similar to `QListWidget`. The `QTableView` widget is more flexible, but also more demanding, as it has the ability to have different data models (which can be much faster with large tables). As such, only if your needs are not too complicated this widget will be a good choice.

The dimensions of the table must be set prior to adding items. The methods `setRowCount` and `setColumnCount` are used.

The `QTableWidget` class has a built in model that is built up item by item. Items are of class `QTableWidgetItem` and are created first, then inserted into the widget by row and column through the method `setItem` (These operations are not vectorized and can be slow for large tables.) Items can be removed by index with `takeItem`. The item can be reinserted. The `clear` will remove all items, even headers items, whereas, `clearContents` will leave the headers. Both keep the dimensions.

As with `QListWidget`, items have various properties that can be adjusted. The text can be specified to the constructor, or set through `setText`. Text alignment is specified through `setTextAlignment`. The font may be configured through `setFont`. The methods `setBackground` and `setForeground` are used to adjust the colors.

Items may also have icons (`setIcon`), tooltips (`setToolTip`), and statusbar tips (`setStatusTip`).

Similar to `QListWidget`, `QTableWidgetItem` instances are returned by the method `item`, with a specification of row and column; and by `itemAt`, with a specification of a position. The `findItems` will return a list of items matching a string. Also, there is the method `currentItem`. From an item, the current column and row can be found with the methods `column` and `row`

**Item flags** As mentioned, items may have several different properties: are they editable, draggable, ... To specify, one sets an items flags with values taken from the `Qt::ItemFlag` enumeration. The possible values are: `"NoItemFlags"`, `"ItemIsSelectable"`, `"ItemIsEditable"`, `"ItemIsDragEnabled"`, `"ItemIsDropEnabled"`, `"ItemIsUserCheckable"`, `"ItemIsEnabled"`, and `"ItemIsTriState"` (has three check states). To make an item checkable, one must first set the check state. By default, the widget has all these sets (except no and tri state). To remove a flag, one can specify all the ones they want, or use integer arithmetic and subtract. E.g., to remove the editable attribute one has this possibility:

```
item <- Qt$QTableWidgetItem("Not editable")
item$setFlags(item$flags() - Qt$Qt$ItemIsEditable)
```

**Headers** Columns may have headers. These are set all at once by specifying a character vector to `setHorizontalHeaderLabels`, or can be set with an item by `setHorizontalHeaderItem`. The header is returned by `horizontalHeader` which has a method `setVisible` to toggle its visibility. Similarly, rows can have (vertical) headers.

To specify the width of a column the method `setColumnWidth` is available. One specifies the column, then the width in pixels. The method `horizontalHeader` returns an instance of class `QHeaderView`. To make the last column stretch to fill the available space is specified through the method `setStretchLastSection` with a value of `TRUE`.

**Sorting and Filtering** This widget can have its rows sorted by the values of a column through the method `sortItems`. One specifies the column by index, and an order. The default is "AscendingOrder", the alternative is `Qt::DescendingOrder`. Sorting should be done after the table is populated with items.

Rows and columns can be hidden through the methods `setRowHidden` and `setColumnHidden`. This can be used for filtering purposes without re-drawing the table.

**Selection** For the table widget one can easily select rows, columns, blocks and even combinations thereof. An underlying selection model implements selection, but the `QTableWidget` provides an interface. The currently selected items are returned as a list through the method `selectedItems`.

A given cell may be selected by index (row then column) through the method `setCurrentCell`, or if the item instance is known by `setCurrentItem`. In addition, an option selection flag from the enumeration `QItemSelectionModel::SelectionFlag` can be specified with values among "Select", to add the item to the selection; "Clear", to clear all selection; "Toggle", to toggle the specified item; "Rows", to extend the selection to the enclosing row; and similarly for "Columns". For an item, the method `setSelection` takes a logical value to indicate the selection state.

The current selection can be cleared with by selecting an item with the "Clear" attribute, or by grabbing the underlying selection model and calling its `clearSelection` method. E.g, something like:

```
tbl$selectionModel()$clearSelection()
```

**Signals** The `QTableWidget` class has a number of signals it emits. They mostly come in pairs: "cell" ones passing in the row and column index and "item" ones passing in an item reference. For example, `cellClicked` and `itemClicked`, both called when a cell is clicked. Also there are `cellDoubleClicked`, `cellEntered`, `cellPressed`, `currentCellChanged` and `cellChanged` (with similar "item" ones). Also of interest is the `itemSelectionChanged` which is called when the selection changes.

#### Example 4.4: Selection of variables

This example shows how to use the table widget to select variable names in a data set. The `QListWidget` can also be used for this, but with the table widget we can add more detail in other columns. We begin by placing the table widget into a box layout.

```
tbl <- Qt::QTableWidget()
tbl$setSizePolicy(Qt::QSizePolicy$Expanding, Qt::QSizePolicy$Expanding)
```

```
lyt <- Qt$QVBoxLayout()
lyt$addWidget(tbl)
```

We use a `QDialogButtonBox` to hold our two buttons.

```
db <- Qt$QDialogButtonBox()
lyt$addWidget(db)
db$addButton(Qt$QDialogButtonBox$Ok)
db$addButton(Qt$QDialogButtonBox$Cancel)
qconnect(db, "accepted", function() runCommand())
qconnect(db, "rejected", function() w$hide())
```

We assume we have a data frame, `df`, with two columns – one for the variable name and one for detail, in this case the class of the variable.

We populate our table first by setting its dimensions and then setting the headers. We stretch the last column and set the column width of the first to accomodate the longest variable name in our example.

```
tbl$clear()
tbl$setRowCount(nrow(df))
tbl$setColumnCount(ncol(df))
tbl$setHorizontalHeaderLabels(c("Variable", "Class"))
tbl$horizontalHeader()$setStretchLastSection(TRUE)
tbl$setColumnWidth(0, 200)
tbl$verticalHeader()$setVisible(FALSE)
```

Each row in this example has two items, one for the checkable item holding the variable name and one for detail. Here we add row by row.

```
for(i in 1:nrow(df)) {
  item <- Qt$QTableWidgetItem(df[i,1])
  item$setCheckState(Qt$Qt$Unchecked)
  tbl$setItem(i-1, 0, item)

  item <- Qt$QTableWidgetItem(df[i,2])
  item$setTextAlignment(Qt$Qt$AlignLeft)
  tbl$setItem(i-1, 1, item)
}
```

Finally, this command is called when the "Ok" button is pressed. For simplicity we do two passes through the data to grab the text and the check state (which we must coerce to get a logical value).

```
runCommand <- function() {
  n <- tbl$rowCount
  x <- sapply(seq_len(n), function(i) tbl$item(i-1, 0)$text())
  ind <- sapply(seq_len(n), function(i) tbl$item(i-1, 0)$checkState())
  print(x[as.logical(ind)])
}
```

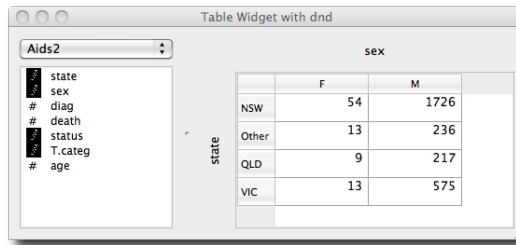


Figure 4.1: A table widget to display contingency tables and a means to specify the variables through drag and drop.

### Example 4.5: A drag and drop interface to xtabs

This examples uses a table widget to display the output from `xtabs`. To specify the variable names, we use the `VariableSelector` class defined in Example 4.3. That provides a means to drag the variable. We next define a drop label, similar to that defined in Example ???. We show the major differences only. Our constructor is similar, but adds three properties: a function to call for the drop event, a value beyond the text, and an angle for text rotation. We define accessors for these properties, but do not show them here.

```
qsetClass("DropLabelRotation", Qt$QLabel, function( parent=NULL) {
  super(parent)

  this$dropFunction <- function(obj, data) {}
  this$value <- list(df=NULL, var=NULL)
  this$angle <- 0L

  setAcceptDrops(TRUE)
  setAlignment(Qt$Qt$AlignHCenter | Qt$Qt$AlignVCenter)
})
```

To handle the drag events we override the methods `dragEnterEvent`, `dragLeaveEvent`, and `dropEvent`. Only the latter is shown. For that we check for data with MIME type `R/serialized-data`. In that case, we call `unserialize` on the data and then the function in the `dropFunction` property.

```
qsetMethod("dropEvent", DropLabelRotation, function(e) {
  setForegroundRole(Qt$QPalette$WindowText)
  md <- e$mimeType()
  if(md$hasFormat("R/serialized-data")) {
    data <- unserialize(md$data("R/serialized-data"))
    dropFunction(this, data)
  }
```

```

    setBackgroundRole(Qt::QPalette::Window)
    e$acceptProposedAction()
  }
})

```

For this application we want to be able to show text in the drop label in a rotated state. The following allows this. We override the `paintEvent` event if the `angle` property is non-zero, otherwise. The chapter on `qtpaint` will cover the techniques used in `drawRotatedText`. In short we translate the origin to the center of the label's rectangle, rotate coordinates by the angle, then place the text within this new rectangle/

```

qsetMethod("paintEvent", DropLabelRotation, function(e) {
  if(!this$angle) {
    super("paintEvent", e)
  } else {
    p <- Qt::QPainter()
    p$begin(this)
    this$drawRotatedText(p)
    p$end()
    update()
  }
})

```

```
[1] "paintEvent"
```

```

qsetMethod("drawRotatedText", DropLabelRotation, function(p) {
  w <- this$width; h <- this$height
  p$save()
  p$translate(w/2, h/2)
  p$rotate(this$angle)
  p$drawText(qrect(-h, -w, h, w), Qt::Qt::AlignCenter, this$text)
  p$restore()
})

```

```
[1] "drawRotatedText"
```

Our main widget consists of three widgets: two drop labels for the contingency table and a table widget to show the output. This could be extended to include a third variable for three-way tables, but we leave that exercise for the interested reader. The constructor simply calls two methods to be defined next.

```

qsetClass("XtabsWidget", Qt::QWidget, function(parent=NULL) {
  super(parent)

  initWidgets()
  initLayout()
})

```

#### 4. WIDGETS USING THE MVC FRAMEWORK

---

Class 'R::.GlobalEnv::XtabsWidget' with 318 public **methods**

We have three widgets to initialize. The xlabel is pretty basic: we define a widget and set a drop function. For the ylabel we also adjust the rotation and constrain the width based on the font size. The latter is necessary, as otherwise the label width reflects the length of the dropped text. The clearLabelsust sets the text and initializes the values for the labels. It is not shown.

```
qsetMethod("initWidgets", XtabsWidget, function() {  
  ## make Widgets  
  this$xlabel <- DropLabelRotation()  
  this$ylabel <- DropLabelRotation()  
  pt <- this$ylabel$font$pointSize()  
  this$ylabel$setMinimumWidth(2*pt); this$ylabel$setMaximumWidth(2*pt)  
  this$ylabel$setRotation(-90L)  
  
  this$tw <- Qt$QTableWidget()  
  
  clearLabels()  
  
  f <- function(obj, data) {  
    obj$setValue(data)  
    obj$setText(data$var)  
    this$makeTable()  
  }  
  this$xlabel$setDropFunction(f)  
  this$ylabel$setDropFunction(f)  
})
```

[1] "initWidgets"

We don't show the layout code for this widget, it is a simple application of QGridLayout, but do show how we create the call to xtabs to create the data after the drop events.

```
qsetMethod("makeTable", XtabsWidget, function() {  
  df <- this$xlabel$getValue()$df  
  if(is.null(xVar <- this$xlabel$getValue()$var)) {  
    out <- NULL  
  } else if(is.null(yVar <- this$ylabel$getValue()$var)) {  
    f <- formula(sprintf("~ %s", xVar))  
    out <- xtabs(f, data=get(df))  
  } else {  
    f <- formula(sprintf("~ %s + %s", yVar, xVar))  
    out <- xtabs(f, data=get(df))  
  }  
  if(!is.null(out))  
    updateTableWidget(out)
```



```
})
```

Finally, for the `XtabsWidget` class we define a method to update the table widget. We have two cases and the code repeats itself a bit. The basic tasks are to set the dimensions of the table, update the header labels, and then populate the table.

```
qsetMethod("updateTableWidget", XtabsWidget, function(out) {
  tw <- this$tw
  tw$clear()
  ndims <- length(dim(out))
  if(ndims == 1) {
    tw$setRowCount(1)
    tw$setColumnCount(dim(out))

    tw$setHorizontalHeaderLabels(names(out))
    tw$horizontalHeader()$setVisible(TRUE)
    tw$verticalHeader()$setVisible(FALSE)
    sapply(seq_along(out), function(i) {
      item <- Qt$QTableWidgetItem(as.character(out[i]))
      item$setTextAlignment(Qt$Qt$AlignRight)
      tw$setItem(0, i-1, item)
    })
  } else if(ndims == 2) {
    tw$setRowCount(dim(out)[1])
    tw$setColumnCount(dim(out)[2])

    tw$setHorizontalHeaderLabels(colnames(out))
    tw$setVerticalHeaderLabels(rownames(out))
    tw$horizontalHeader()$setVisible(TRUE)
    tw$verticalHeader()$setVisible(TRUE)

    for(i in 1:dim(out)[1])
      for(j in 1:dim(out)[2]) {
        item <- Qt$QTableWidgetItem(as.character(out[i,j]))
        item$setTextAlignment(Qt$Qt$AlignRight)
        tw$setItem(i-1, j-1, item)
      }
  }
})
```

```
[1] "updateTableWidget"
```

Figure 4.1 shows the widget placed within a splitter window after a few variables have been dragged.

### A Tree Widget

The `QTreeWidget` provides a tree view based on a coupled tree model. It is used in a manner similar to the table widget, however, one must adjust for the hierarchical nature of the data it can display. The widget shows a specific number of columns, which is specified via `setColumnCount`. The column headers are set through `setHeaderLabels` by specifying a vector of column names.

**Tree Widget Items** The widget organizes itself through items, with each item having 1 or more columns. The items in the tree widget are instances of the `QTreeWidgetItem` class. An item displays a value in each column. The method `setText` takes two arguments, an integer specifying a column, and a value to display. Similarly, other such item methods require one to specify the column. These methods include: `setTextAlignment`, `setFont`, `setToolTip`, `setStatusTip`, and `setIcon`. To retrieve these values, the method `data` takes a column number and a role, such as `Qt::DisplayRole` to get the text.

**Heirarchy** A tree is used to represent a heirarchy. The items have several methods related to this. Each item has a `parent` method pointing to the parent item, or `NULL` if a top-level item. The method `child` returns the child item, specified by index. The index of a given item in the parent is returned by `indexOfChild`. The total number of children for an item is returned by `childCount`.

New child items are added with either with `addChild`, adding at the end; or `insertChild` adding the child item at the specified index. Child items are removed by the `removeChild` method or the `takeChild`, if specifying by index. The latter returns the item to be reparented if desired.

The view can be made to show if a child is present by calling `setChildIndicatorPolicy` with a value of `Qt::TreeWidgetItem::ShowIndicator` (or one of `DontShowIndicator` or `DontShowIndicatorWhenChildless`).

When an item has children and an indicator is shown, the view may be expanded by clicking on this indicator. The method `isExpanded` returns `TRUE` if expanded. This state can be set via `setExpanded`. The methods `setDisabled` and `setHidden` are also available to disable or hide an item.

The tree widget itself also has methods related to the heirarchy. For the tree widget, there is an invisible root item, returned by `invisibleRootItem`. The children of this root item are the “top-level” items, These are added through `addTopLevelItem` or the `insertTopLevelItem` methods, the latter requiring an index to specify where. The basic idea is that a top-level item is prepared, along with its heirarchy, then added.

**Selection** The currently selected item is returned by `currentItem` and may be specified by item view `setCurrentItem`. For more multiple selection, see

the discussion on selection models 4.3.

**Signals** The widget emits several signals, most passing the item and the column to the handler. Some useful signals are `itemClicked`, `itemDoubleClicked`, `itemExpanded`, and `itemActivated` (for clicking or the Enter key).

#### Example 4.6: A workspace browser

This example shows how to use the tree widget item to display a snapshot of the current workspace. Each object in the workspace maps to an item, where recursive objects with names will have their components represented in a hierarchical manner.

When representing objects in a workspace, we need to decide if an object has been changed. To do this, we use the `digest` function from the `digest` package to store a simple means to compare a current object with a past one. To store this information, we create a subclass of `QTreeWidgetItem` with a `digest` property.

```
require(digest)
qsetClass("OurTreeWidgetItem", Qt$QTreeWidgetItem, function() {
  this$digest <- NULL
})
qsetMethod("digest", OurTreeWidgetItem, function() digest)
qsetMethod("setDigest", OurTreeWidgetItem, function(value) {
  this$digest <- value
})
```

This subclass will be used for top-level items in our tree. We define the following to create an item from a variable name. Since we call this function recursively, we have an argument indicating of the item is a top-level item.

We could modify the display of each item to suit more sophisticated tastes, e.g., icons, but for this example stop by adding a column for the class of the object.

As we use the name of an object to display it, when checking for recursive structures, we also check that they have a `names` attribute.

```
makeItem <- function(varname, obj=NULL, toplevel=FALSE) {
  if(is.null(obj))
    obj <- get(varname, envir=.GlobalEnv)

  if(toplevel) {
    item <- OurTreeWidgetItem()
    item$setDigest(digest(obj))
  } else {
    item <- Qt$QTreeWidgetItem()
  }
}
```

#### 4. WIDGETS USING THE MVC FRAMEWORK

---

```
item$setText(0, varname)
item$setText(1, paste(class(obj), collapse=", "))
## icons, fonts children...

if(is.recursive(obj) && !is.null(attr(obj, "names"))) {
  item$setChildIndicatorPolicy(Qt$QTreeWidgetItem$ShowIndicator)
  for(i in names(obj)) {
    newItem <- makeItem(i, obj[[i]])
    item$addChild(newItem)
  }
}
item
}
```

Next we define a few simple function to add, remove and replace a top-level item.

```
addItem <- function(tr, varname) {
  newItem <- makeItem(varname, toplevel=TRUE)
  tr$addTopLevelItem(newItem)
}
```

We can remove an item by index with the `takeTopLevelItem` method, but here it is more convenient to specify the item to remove.

```
removeItem <- function(tr, item) {
  root <- tr$invisibleRootItem()
  root$removeChild(item)
}
```

```
replaceItem <- function(tr, item, varname) {
  removeItem(tr, item)
  addItem(tr, varname)
}
```

Our main function is one that checks the current workspace and updates the values in the tree widget accordingly. This could be set on a timer to be called periodically, or called in response to user input.

As can be seen, we consider three cases: items no longer in the workspace to remove, new items to add, and finally items that may be new. here is where we check the digest value to see if they need to be replaced. Rather than fuss with inserting new items, we simply add them to the end and the sort the values.

```
updateTopLevelItems <- function(tr) {
  objs <- ls(envir=.GlobalEnv)

  curObjs <- lapply(seq_len(tr$topLevelItemCount), function(i) {
    item <- tr$topLevelItem(i-1)
```

```

    list(item=item, value=item$data(0, role=0) )
# better role?
})
cur <- sapply(curObjs, function(i) i$value)
names(curObjs) <- cur

## we have three types here:
removeThese <- setdiff(cur, objs)
maybeSame <- intersect(cur, objs)
addThese <- setdiff(objs, cur)

tr$setUpdatesEnabled(FALSE)
for(i in removeThese) {
  item <- curObjs[[i]]$item
  removeItem(tr, item)
}

for(i in maybeSame) {
  obj <- get(i, envir=.GlobalEnv)
  if(digest(obj) != curObjs[[i]]$item$digest()) {
    replaceItem(tr, curObjs[[i]]$item, i)
  }
}

for(i in addThese)
  addItem(tr, i)

tr$sortItems(0, Qt$Qt$AscendingOrder)
tr$setUpdatesEnabled(TRUE)
}

```

Now we define some functions for traversing the hierarchical tree structure. First we define a function to find the index of an item, be it a top-level item or an ancestor of one.

```

indexFromItem <- function(tr, item) {
  parent <- item$parent()
  if(is.null(parent))
    parent <- tr$invisibleRootItem()
  parent$indexOfChild(item) + 1
}

```

We find it convenient to think in terms of a path rather than an item, either a vector of indices or names. The `index` argument below decides what to return. the basic approach here is to walk backwards by calling the `parent` method until it is null.

```

pathFromItem <- function(tr, item, index=TRUE) {

```

#### 4. WIDGETS USING THE MVC FRAMEWORK

---

```
getVal <- function(item)
  ifelse(index, indexFromItem(tr, item),
        item$data(0, role=0))

path <- getVal(item)
while(!is.null(item <- item$parent())) {
  path <- c(getVal(item), path)
}
path
}
```

This function reverses the last. It takes a path, specified by indices, and finds the item if possible.

```
itemFromPath <- function(tr, path) {
  item <- tr$invisibleRootItem()
  for(i in path) {
    item <- item$child(i)
    if(is.null(item))
      return(NULL)
  }
  item
}
```

As an illustration, we show how one can use `findItems` to return a list of items matching a query. In this case, we pass in a regular expression to match against the class column (column 1.)

```
matchClass <- function(tr, classname) {
  allItems <- sapply(seq_len(tr$topLevelItemCount), function(i) {
    tr$topLevelItem(i-1)
  })
  matched <- tr$findItems(classname, Qt$Qt$MatchRegExp, 1) # column 1
  sapply(allItems, function(i) i$setHidden(TRUE))
  sapply(matched, function(i) i$setHidden(FALSE))
}
```

Finally we show how this can be used. First, we set the column count and the header labels to match our `makeItem` function above.

```
tr <- Qt$QTreeWidget()
tr$setColumnCount(2) # name, class
tr$setHeaderLabels(c("Name", "Class"))
```

This call initializes the display.

```
updateTopLevelItems(tr)
```

An typical signal handler, to be called when an item is clicked, is illustrated next. Here we print both the path and a summary of the object in the workspace the path points to.

```

qconnect(tr, "itemClicked", function(item, column) {
  path <- pathFromItem(tr, item, index=FALSE)
  obj <- get(path[1], envir=.GlobalEnv)
  if(length(path) > 1)
    obj <- obj[[path[-1]]]          # get object
  str(obj)
  print(path)
  updateTopLevelItems(tr)
})

```

### 4.3 Item models and their views

In this section, we consider models and their views decoupled from each other. We focus our discussion to tabular data, although we note that Qt provides various means to provide models for hierarchical data.

#### Using a data frame for a model

For tabular data a model can be made quite easily with the constructor `qdataFrameModel` provided by `qtbases`. This function maps a data frame to the model, returning an instance of class `DataFrameModel` a subclass of `QAbstractTableModel`.

The default role is for display of the data. Other roles can be defined by adding additional columns with a specific naming convention. A column with name `a` can have its role information specified with another column with name `.a.ROLE` where `ROLE` is one of available roles, such as "decoration", "edit", "toolTip", "statusTip", or "textAlignment".

The following illustrates a basic usage. We use a `QTableView` instance for a view. The model is connected to the view through its `setModel` method.

#### Example 4.7: Using `qdataFrameModel` to list variables in a data frame

This example will show how to use `qdataFrameModel` to create a means to select a variable from a data frame. We will use a built-in data set for this.

```

nms <- names(Cars93)
d <- data.frame("Variables"=nms, stringsAsFactors=FALSE)

```

We will add an icon for decoration. To do so, we create a list of icons for each of the classes represented in the variables we have. Here we use some images from the `gWidgets` package.

```

icons <- sapply(c("factor", "numeric"), function(i) {
  Qt$QPixmap(system.file(sprintf("images/%s.gif", i), package="gWidgets"))
})
icons$integer <- icons$numeric

```

#### 4. WIDGETS USING THE MVC FRAMEWORK

---

To add the icons as a decoration, we simply assign to the appropriately named component of our data frame. Data frames can store such data, although the default print method does not work.

```
d$.Variables.decoration <- sapply(nms, function(i) icons[class(Cars93[[i]])])
```

Similarly, we set an informative tooltip for each variable:

```
d$.Variables.toolTip <- sapply(nms, function(i) capture.output(str(Cars93[[i]])))
```

We now define a model and associate it with a view. In this example we use a `QListView` instance, although in most uses of this model we would use the `QTableView` class. We discuss views later in Section 4.3.

```
model <- qdataFrameModel(d)
view <- Qt$QListView()
view$setModel(model)
```

NULL

For the list view, we have a few properties to adjust, in this case we highlight alternating rows as follows:

```
view$setAlternatingRowColors(TRUE)
```

#### Table Models

The `qdataFrameModel` creates a subclass of `QAbstractTableModel`, itself of subclass of `QAbstractItemModel`. One can subclass `QAbstractTableModel` to provide a custom model for a view. This may be useful in different instances, for example, if the cell columns need to be computed dynamically or if editing of cells is desired.

Before discussing methods that must be implemented, we mention some useful inherited methods. For a table model, items are organized by row and column. This allows one to easily refer to an item by index. The inherited `index` method uses 0-based row and column arguments to return a `QModelIndex` instance. In turn, this index instance has methods `row` and `column` for getting back the coordinate information.

The `match` can be used to return items in a column that match a specific string for a given role. The column is specified by a model index, the role by one of the `Qt::ItemDataRole` enumeration. The `Qt::MatchFlag` specifies how the matching is to occur. Matches are returned as a list of indices.

The `sort` method can be used to sort the data by the values in a column. The column is specified by its index, and the order is one of `Qt$Qt$AscendingOrder` or `Qt$Qt$DescendingOrder`.

**Required methods** The basic implementation of a subclass must provide the methods for `rowCount`, `columnCount`, and `data`. The first two describe



the size of the table for any views, the third describes to the view how to display data for a given role. The signature for the data method is (index, role). The index value, an instance of `QModelIndex`, has methods `row` and `column`, whereas role is one of the roles defined in the `Qt::ItemDataRole` enumeration. This method should return the appropriate value for the given role. For example, if one is displaying numeric data, the display role might format the numeric values (showing a fixed number of digits say), yet the edit role might display all the digits so accuracy is not lost. If a role is not implemented, a value of `NULL` should be returned, as this is mapped to a null instance of `QVariant`.

One may also implement the `headerData` method to return the appropriate data to display in the header for a given role. The main one is `display`.

**Editable Models** For editable models, one must implement the `flags` method to return a flag containing `ItemIsEditable` and the `setData` method. This has signature (index, value, role) where value is a character string containing the edited value. When a value is updated, one should call the `dataChanged` method to notify the views that a portion of the model is changed. This method takes two indices, together specifying a rectangle in the table.

To provide for resizable tables, Qt requires one to call some functions so that any connected views can be notified. For example, an implemented `insertColumns` should call `beginInsertColumns` before adding the column to the model and then `endInsertColumns` just after.

## Table views

A table model is typically displayed through the `QTableView` widget, although one can use the model – first column only – with `QListView` or even `QComboBox`. A custom view is also possible, as illustrated later. The table view widget inherits from `QAbstractItemView` which provides the method `setModel` to link the underlying model with a view. This link passes along information about the model being changed back to the view to process and connects the delegates that allow one to edit values in the view and have the changes propagate back to the model.

Once, connected, the table view can have its properties adjusted to control its appearance. For example, columns (and rows) may be hidden or shown through the `setColumnHidden` method. Their widths can be adjusted by the mouse, or through `setColumnWidth`. The grid style can be adjusted through `setGridStyle`.

The view may allow sorting of the underlying model. This is enabled through `setSortingEnabled`. The method `sortByColumn` can be called specifying the column and a sort order (e.g. `Qt$Qt$AscendingOrder`).

**Headers** The table view has headers, horizontal ones for the columns and vertical ones for the rows. The methods `verticalHeader` and `horizontalHeader` return instances of the `QHeaderView` class. This class has many methods. We mention `setHidden`, to suppress the header display; `showSortIndicator`, to display a sort arrow; and `setStretchLastSection` to stretch the last section to fill the space. For tree views, this is `TRUE` for horizontal headers but not for table views.

**Selection Models** The type of selection possible is determined by the `selectionMode` of the view, the options for which are enumerated in `QAbstractItemView::SelectionMode`. Typical values are `SingleSelection` or `ExtendedSelection`, which allows one to extend their selection by pressing the Control or Shift keys while selecting.

For abstract item views, the underlying selection is handled by a selection model, which is found in the `selectionModel` property. The `QItemSelectionModel` can return the current selection by rows through `selectedRows` (with an argument to specify the column to check), columns (`selectedColumns`) or by index with `selectedIndexes`. In each case, a list of model indices is returned. The method `row` and `column` are useful then. The selection model allows one specify if the selection will apply to the entire row or column or if selection is cell by cell. This is done through the selection flags (`QItemSelectionModel::SelectionFlag`)

The selection can be updated programmatically through the `setCurrentIndex` method. The index and a selection flag must be given. The latter enumerated in `QItemSelectionModel::SelectionFlag`, with values such as `Select`, to select the item; `Deselect`, `Clear`; `Toggle`; and `Rows` and `Columns` to select the entire range.

The selection model emits a few signal, notable `selectionChanged` is emitted when a new selection is made.

#### Example 4.8: Using a custom model to edit a data frame

This example shows how to create a custom model to edit a data frame. The performance is much less responsive than that provided by `qdataFrameModel`, as the bulk of the operations are done at the R level. We speed things up a bit, by placing column headers into the first row of the table, and not in the expected place. The numerous calls to a `headerData` method implemented in R proved to be quite slow.

Our basic constructor simply creates a `dataframe` property and sets the data frame.

```
qsetClass("DfModel", Qt$QAbstractTableModel, function(df=data.frame(V1=character(0))
  super(parent)
  ## properties
  this$dataframe <- NULL

  setDf(df)
```

```
})
```

We need an accessor for the dataframe property. When setting a new one, we call the `dataChanged` method to notify any views of a change.

```
qsetMethod("Df", DfModel, function() dataframe)
qsetMethod("setDf", DfModel, function(df) {
  this$dataframe <- df
  dataChanged(index(0, 0), index(nrow(df), ncol(df)))
})
```

All subclass of `QAbstractTableModel` must implement a few methods, here we do `rowCount` and `columnCount` using the dimensions of the current data frame.

```
qsetMethod("rowCount", DfModel, function(index) nrow(this$Df()) + 1)
qsetMethod("columnCount", DfModel, function(index) ncol(this$Df()))
```

The data method is the main method to implement. We wish to customize the data display based on the class of the variable represented in a column. We implement this with S3 methods. We define several. For example, to change the display role we have the following.

```
displayRole <- function(x, row, context) UseMethod("displayRole")
displayRole.default <- function(x, row, context) sprintf("%s", x[row])
displayRole.numeric <- function(x, row, context) sprintf("%.2f", x[row])
displayRole.integer <- function(x, row, context) sprintf("%d", x[row])
```

We see that numeric values are formatted to have 2 decimal points. By setting the display role, we can store numeric data with all its digits, so that we can edit the entire value, but have values formatted along the decimal point. The context argument is shown to indicate how to make the number of digits context sensitive.

The main data has this basic structure (we avoid showing the cases for all the different roles).

```
qsetMethod("data", DfModel, function(index, role) {
  d <- this$Df()
  row <- index$row()
  col <- index$column() + 1

  if(role == Qt$Qt$DisplayRole) {
    if(row > 0)
      displayRole(d[,col], row)
    else
      names(d)[col]
  } else if(role == Qt$Qt$EditRole) {
    if(row > 0)
      as.character(d[row, col])
    else

```

```

        names(d)[col]
    } else {
        NULL
    }
})

```

To allow the user to edit the values we need override the `flags` method to return `ItemIsEditable` in the flag, so that any views are aware of this ability.

```

qsetMethod("flags", DfModel, function(index) {
  if(!index$isValid()) {
    return(Qt$Qt$ItemIsEnabled)
  } else {
    curFlags <- super("flags", index)
    return(curFlags | Qt$Qt$ItemIsEditable)
  }
})

```

To edit cells we need to implement a method to set data once edited. The trick is that value is a character, so we coerce to the right type for the column it is in. We do this with an S3 method. For example we have this (among others):

```

fitIn <- function(x, value) UseMethod("fitIn")
fitIn.default <- function(x, value) value
fitIn.numeric <- function(x, value) as.numeric(value)

```

The `setData` is responsible for taking the value from the delegate and assigning it into them model, we use the `fitIn` function unless it is a column header.

```

qsetMethod("setData", DfModel, function(index, value, role) {

  if(index$isValid() && role == Qt$Qt$EditRole) {
    d <- this$Df()
    row <- index$row()
    col <- index$column() + 1

    if(row > 0) {
      x <- d[, col]
      d[row, col] <- fitIn(x, value)
    } else {
      names(d)[col] <- value
    }
    this$dataframe <- d
    dataChanged(index, index)

    return(TRUE)
  } else {
    super("setData", index, value, role)
  }
})

```

```
}
})
```

For a data frame editor, we may wish to extend the API for table of items to be R specific. This method allows one to replace a column of values.

```
qsetMethod("setColumn", DfModel, function(col, value) {
  ## pad with NA if needed
  n <- nrow(this$Df())
  if(length(value) < n)
    value <- c(value, rep(NA, n - length(value)))
  value <- value[1:n]
  d <- this$Df()
  d[,col] <- value
  this$dataframe <- d # only notify about this column
  dataChanged(index(0, col-1), index(rowCount()-1, col-1))
  return(TRUE)
})
```

We implement a method similar to the `insertColumn` method, but specific to our task. Since we may add a new column, we call the "begin" and "end" methods to notify any views.

```
qsetMethod("addColumn", DfModel, function(name, value) {
  d <- this$Df()
  if(name %in% names(d)) {
    return(setColumn(min(which(name == names(d))), value))
  }

  beginInsertColumns(Qt$QModelIndex(), columnCount(), columnCount())
  d <- this$Df()
  d[[name]] <- value
  this$dataframe <- d
  endInsertColumns()
  return(TRUE)
})
```

To test this out, we define a model and set it in a view.

```
model <- DfModel(mtcars)
view <- Qt$QTableView()
view$setModel(model)
```

Finally, we customize the view by defining what triggers the delegate for editing and hide the row and column headers.

```
triggerFlag <- Qt$QAbstractItemView$DoubleClicked |
               Qt$QAbstractItemView$SelectedClicked |
               Qt$QAbstractItemView$EditKeyPressed
view$setEditTriggers(triggerFlag)
view$verticalHeader()$setHidden(TRUE)
```

```
view$horizontalHeader()$setHidden(TRUE)
```

### Custom Views

One can write a custom view for a model. In the example, we show how to connect a label's text to the values in a column of a model. The base class for a view of an item model is `QAbstractItemView`. This class has many methods – that can be overridden – to cover keyboard, mouse, scrolling, selection etc. Here we focus on just enough for our example.

#### Example 4.9: A label that updates as a model is updated

This example shows how to create a simple custom view for a table model. We use the `DfModel` class developed in Example 4.8 to provide the model.

Our custom view class will be a sub-class of `QAbstractItemView`, although we don't consider the individual items below. Our goal is a simple illustration, where we provide a label with text summarizing the mean of the values in the first column of the model.

In the constructor we call the inherited `setModel` method and the `setLabelValue` method we define later.

```
qsetClass("CustomView", Qt$QAbstractItemView, function(parent=NULL) {
  super(parent)
  this$label <- Qt$QLabel()
  label$setMinimumSize(400,400)
  setModel(NULL)
})
```

The label has an accessor function which we use in our private `setLabelValue` method defined below. Here we get the model from the custom view.

```
qsetMethod("label", CustomView, function() label)
qsetMethod("setLabelValue", CustomView, function() {
  m <- this$model()
  if(is.null(m)) {
    txt <- "No model"
  } else {
    df <- m$Df()
    xbar <- mean(df[,1])
    txt <- sprintf("The mean is %s", xbar)
  }
  label$setText(txt)
},
  access="private")
```

The main method call is when the data is changed. Here we update the label text when so notified.

```
qsetMethod("dataChanged", CustomView, function(upperIndex,lowerIndex) {
```

```
    setLabelValue()  
  })
```

To initialize the label text, we add a call to `setLabelValue` when the model is set.

```
qsetMethod("setModel", CustomView, function(model) {  
  super("setModel", model)  
  setLabelValue()  
})
```

Finally, to test this out, we put the view in a simple GUI along with an instance of an editable data frame. When we modify the model through that, the label's text updates accordingly.

```
model <- DfModel()  
model$setDf(mtcars)  
view <- Qt$QTableView()  
view$setModel(model)  
view$setEditTriggers(Qt$QAbstractItemView$DoubleClicked)  
cv <- CustomView()  
cv$setModel(model)  
w <- Qt$QWidget()  
lyt <- Qt$QHBoxLayout()  
w$setLayout(lyt)  
lyt$addWidget(view)  
lyt$addWidget(cv$label())
```





Qt paint