1

Qt: Overview

1.1 The Qt library

Qt is an open-sourced, cross-platform application framework that is perhaps best known for its widget toolkit. The features of Qt are divided into about a dozen modules. We highlight some of the more important and interesting ones:
Core Basic utilties, collections, threads, I/O, ...
Gui Widgets, models, etc for graphical user interfaces
OpenGL Convenience layer (e.g., 2D drawing API) over OpenGL
Webkit Embeddable HTML renderer (shared with Safari, Chrome)
Other modules include functionality for networking, XML, SQL databases, SVG, and multimedia. However, R packages already provide many of those features.

The history of Qt begins with Haavard Nord and Eirik Chambe-Eng in 1991 and follows with the Trolltech company, until 2008. It is now owned by Nokia, a major cell-phone producer. While it was originally unavailable as open-source on every platform, version 4 was released universally under the GPL. With the release of Qt 4.5, Nokia additionally placed Qt under the LGPL, so it is available for use in proprietary software, as well. Popular software developed with Qt include the communication application Skype and the KDE desktop for Linux.

Qt is developed in C++ with extensions that require a special preprocessor called the *Meta Object Compiler* (MOC). The MOC allows for convenient syntax in the definition of signals, slots (signal handlers), and properties, which behave very similarly to those of GTK+.

There are many languages with bindings to Qt, and R is one such language. The qtbase package interfaces with every module of the library. As its name suggests, qtbase forms the base for a number of R packages that provide high-level special-purpose interfaces to Qt. The qtpaint package extends the QGraphicsView canvas to better support interactive statistical graphics. Features include: a layered buffering strategy, efficient spatial queries for mapping user actions to the data, and an OpenGL renderer optimized for statistical plots. An interface resembling that of the lattice package is provided for qtpaint by the mosaiq package. The cranvas package builds on qtpaint to provide a collection of high-level interactive plots in the conceptual vein of GGobi. A number of general utilities are implemented by qtutils, including an object browser widget, an R console widget, and a conventional R graphics device based on QGraphicsView.

While qtbase is not yet as mature as tcltk and RGtk2, we include it in this book, as Qt compares favorably to GTK+ in terms of GUI features and excels in several other areas, including its fast graphics canvas and integration of the WebKit web browser. [1] In addition, Qt, as a commercially supported package, has thorough documentation of its API, including many C++ examples. However, the complexity of C++ and Qt may present some challenges to the R user. In particular, the developer should have a strong grounding in object-oriented programming and have a basic understanding of memory management.

The development of qtbase package is hosted on Github (http://github.com/ggobi/qtbase). It depends on the Qt framework, available as a binary install from http://qt.nokia.com/.

1.2 An introductory example

As a synopsis for how one programs a GUI using qtbase, we present a simple dialog that allows the user to input a date. A detailed introduction to these concepts will follow this example.

After ensuring that the underlying libraries are installed, the package may be loaded like any other R package:

```
require(qtbase)
```

Constructors As with all other toolkits, Qt widgets are objects, and the objects are created with constructors. 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.

```
window <- Qt$QWidget()
label <- Qt$QLabel("Date:")
edit <- Qt$QLineEdit()
button <- Qt$QPushButton("Ok")
```

The constructors are not found in the global environment, but rather in the Qt environment, an object exported from the qtbase namespace. As such, the $ lookup operator is used.

[1]There is a GTK+ WebKit port, but it is not included with GTK+ itself.

Widgets in Qt have various properties that specify the state of the object. For example, the `windowTitle` property controls the title of a top-level widget:

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

Qt objects are represented as extended R environments, and every property is a member of the environment. The `$` function called above is simply that for environments.

Method calls tell an object to perform some behavior. Like properties, methods are accessible from the instance environment. For example, the `QLineEdit` widget supports an input mask that constrains user input to a particular syntax. 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 set an input mask we have:

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

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 Chapter 2, 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(label, row=0, column=0, rowSpan=1, columnSpan=1)
lyt$addWidget(edit, 0, 1, 1, 1)
lyt$addWidget(button, 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`:

```
window$setLayout(lyt)
```

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

```
window$show()
```

Callbacks As with other GUI toolkits, we add interactivity to our GUI by binding callbacks to certain signals. To react to the clicking of the button, the programmer attaches a handler to the `clicked` signal using the `qconnect` function. The function requires the object, the signal name and the handler. Here we print the value stored in the "Date" field.

```
handler <- function()  print(edit$text)
qconnect(button, "pressed", handler)
```

We will discuss callbacks more completely in Section 1.6.

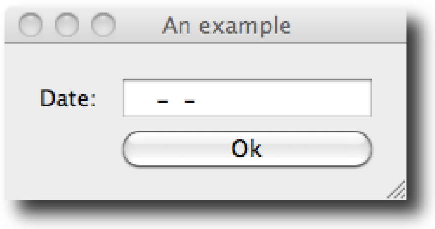


Figure 1.1: Screenshot of our sample GUI to collect a date from the user.

Object-oriented support `QLineEdit` can validate text input, and we would like to validate the entered date. There are a few built-in validators, and for this purpose the regular expression validator could be used, but it would be difficult to write a sufficiently robust expression. Instead we attempt to coerce the string value to a date via R's `as.Date` function with a format of `"%Y-%m-%d"`. In GTK+, validation would be implemented by a signal handler or other callback. However, as C++ is object-oriented, Qt expects the programmer to derive a new class from `QValidator` and pass an instance to the `setValidator` method on `QLineEdit`.

It is possible to define R subclasses of C++ classes with `qtbase`. More details on working with classes and methods are provided in Section 1.8. For this task, we need to extend `QValidator` and override its `validate` virtual method. The `qsetClass` function defines a new class:

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

To override `validate`, we call `qsetMethod`:

```
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)
})
```

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 indicates a state of "Acceptable", "Invalid", or, if neither can be determined, "Intermediate." These values are listed in an enumeration in the `Qt$QValidator` class (cf. Section 1.7 for more on enumerations).

The class object, which doubles as the constructor, is defined in the current top-level environment as a side effect of `qsetMethod`. We call it to construct an instance, which is passed to the edit widget:

```
validator <- DateValidator()
edit$setValidator(validator)
```

1.3 Classes and objects

The `qtbase` package exports very few objects. The central one is an environment, `Qt`, that represents the Qt library in R. [2] The components of this environment are `RQtClass` objects that represent an actual C++ class or namespace. For example, the `QWidget` class is represented by `Qt$QWidget`:

```
Qt$QWidget
```

```
Class 'QWidget' with 318 public methods
```

An `RQtClass` object contains methods in the class scope (*static* methods in C++), enumerations defined by the class, and additional `RQtClass` objects representing nested classes or namespaces. Here we list some of the components of `QWidget` and access one of the enumeration values:

```
head(names(Qt$QWidget), n = 3)
```

```
[1] "connect"              "DrawChildren"         "DrawWindowBackground"
```

```
Qt$QWidget$DrawChildren
```

```
Enum value: DrawChildren (2)
```

Most importantly, however, an instance of `RQtClass` is in fact an R `function` object, and serves as the constructor of instances of the class. For example, we could construct an instance of `QWidget`:

```
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"
```

[2] The `Qt` object is an instance of `RQtLibrary`. The `qtbase` package provides infrastructure for binding any conventional C++ library, even those independent of Qt. Third party packages can define their own `RQtLibrary` object for some other library.

The base class, `RQtObject`, is an environment containing the properties and methods of the instance. For `w`, we list the first few using `ls`:

```
head(ls(w), n=3)
```

```
[1] "dumpObjectTree" "childAt"        "setFixedSize"
```

Properties and methods are accessed from the environment in the usual manner. The most convenient extractor is the `$` operator, but `[[` and `get` will also work. (With the `$` operator at the command line, completion works.) For example, a `QWidget` has a `windowTitle` property which is used when the widget draws itself with a window:

```
w$windowTitle                                   # initially NULL
```

```
NULL
```

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

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

Although Qt defines methods for accessing properties, the R user will normally invoke methods that perform some action. For example, we could show our widget:

```
w$show()
```

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()
```

`QPushButton` extends `QWidget` and thus inherits the properties like `windowTitle`:

```
is(b, "QWidget")
```

```
[1] TRUE
```

```
b$windowTitle
```

```
NULL
```

It is important to realize this distinction when referencing the documentation. As with GTK+, the methods are documented with the class that declares the method.

1.4 Methods and dispatch

In C++, it is possible to have multiple methods and constructors with the same name, but different signatures. This is called *overloading*. An overloaded method is roughly similar to an S4 generic, save the obvious difference that an S4 generic does not belong to any class. The selected overload is that with the signature that best matches the types of the arguments. The exact rules of overload resolution are beyond our scope.

It is particularly common to overload constructors. For example, a simple push button can be constructed in several different ways. Here again is the invocation of the `QPushButton` constructor with no arguments:

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

By convention, all classes derived from `QObject`, including `QWidget`, provide a constructor that accepts a parent `QObject`. This has important consequences that are discussed later. We demonstrate this for `QPushButton`:

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

An alternative constructor for `QPushButton` accepts the text for the label on the button:

```
b <- Qt$QPushButton("Button text")
```

Buttons may also have icons, for example

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

We have passed three different types of object as the first argument to `Qt$QPushButton`: a `QWidget`, a string, and finally a `QIcon`. The dispatch depends only on the type of argument, unlike the constructors in `RGtk2`, which dispatch based on which arguments are specified. (In particular, dispatch is based on position of argument, but not on names given to arguments. We use names only for clarity in our examples.)

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, to learn the methods for a simple button, we would call:

```
out <- qmethods(Qt$QPushButton)
dim(out)
```

```
[1] 435   5
```

```
head(out, n=3)
```

```
         name       return                     signature protected static
1 QPushButton QPushButton*                 QPushButton()     FALSE
FALSE
2 QPushButton QPushButton*        QPushButton(QWidget*)     FALSE
FALSE
3 QPushButton QPushButton* QPushButton(QIcon, QString)     FALSE
FALSE
```

1.5 Properties

Every `QObject`, which includes every widget, may declare a set of properties that represent its state. We list some of the available properties for our button:

```
head(qproperties(b))
```

```
                               type readable writable
objectName                  QString     TRUE     TRUE
modal                          bool     TRUE    FALSE
windowModality Qt::WindowModality     TRUE     TRUE
enabled                        bool     TRUE     TRUE
geometry                      QRect     TRUE     TRUE
frameGeometry                 QRect     TRUE    FALSE
```

As shown in the table, every property has a type and logical settings for whether the property is readable and/or writeable. Virtually every property value may be read, and it is common for properties to be read-only. For example, we can fully manipulate the `objectName` property, but our attempt to modify the `modal` property fails:

```
b$objectName <- "My button"
b$objectName
```

```
[1] "My button"
```

```
b$modal
```

```
[1] FALSE
```

```
try(b$modal <- TRUE)                    # fails
```

Qt provides accessor methods for getting and setting properties. The getter methods have the same name as the property, so they are masked at the R level. Setter methods are available and are typically named with the word "set" followed by the property name:

```
b$setObjectName("My button")
```

However, it is recommended to use the replacement syntax shown in the previous example, for the sake of symmetry.

1.6 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 define a special type of method known as a slot in another component (or the same) that can be connected to the signal as the handler. The two components are decoupled as the emitter does not need to know about the receiver except through the signal connection. In R, any function can be treated as a slot and connected as a signal handler. This is similar to the signal handling in RGtk2. The function qconnect establishes the connection of an R function to a signal. For example

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

Signals are defined by a class and are inherited by subclasses. Here, we list some of the available signals for the QPushButton class:

```
tail(qsignals(Qt$QPushButton), n=5)
```

```
      name     signature
4  pressed     pressed()
5 released    released()
6  clicked clicked(bool)
7  clicked     clicked()
8  toggled toggled(bool)
```

The signal definition specifies the callback signature, given in the signature column. Like other methods, signals can be overloaded so that there are multiple signatures for a given signal name. Signals can also have default arguments, and arguments with a default value are optional in the signal handler. We see this for the clicked signal, where the bool (logical) argument, indicating whether the button is checked, has a default value of FALSE. The clicked signal is automatically overloaded with a signature without any arguments.

The qconnect function attempts to pick the correct signature by considering the number of formal arguments in the callback. Rarely, two signatures will have the same number of arguments, in which case one will be chosen arbitrarily. To connect to a specific signature, the full signature, rather than only the name, should be passed to qconnect. For example, the following two calls are equivalent:

```
qconnect(b, "clicked", function(checked) print(checked))
qconnect(b, "clicked(bool)", function(checked) print(checked))
```

Any object passed to the optional argument user.data is passed as the last argument to the signal handler. The user data serves to parameterize the callback. In particular, it can be used to pass in a reference to the sender object itself, although we encourage the use of closures for this purpose.

The `qconnect` function returns a dummy `QObject` instance that provides the slot that wraps the R function. This dummy object can be used with the `disconnect` method on the sender to break the signal connection:

```
proxy <- qconnect(b, "clicked", function() print("ouch"))
b$disconnect(proxy)
```

```
[1] TRUE
```

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.

Unlike GTK+, Qt widgets generally do not emit hardware events, such as a mouse press, via signals. Instead, a method in the widget is invoked upon receipt of an event. The developer is expected to extend the widget and override the method to catch the event. The apparent philosophy of Qt is that hardware events are low-level and thus should be handled by the widget, not some other instance. We will discuss extending classes in Section **??**. Example **??** demonstrates handling widget events.

1.7 Enumerations and flags

Often, it is useful to have discrete variables with more than two states, in which case a logical value is no longer sufficient. For example, the label widget has a property for how its text is aligned. It supports the alignment styles left, right, center, top, bottom, etc. These styles are enumerated by integer values and Qt defines these by name within the relevant class or, for global enumerations, in the `Qt` namespace. Here are examples of both:

```
Qt$Qt$AlignRight
```

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

```
Qt$QSizePolicy$Expanding
```

```
Enum value: Expanding (7)
```

The first is the value for right alignment from the `Alignment` enumeration in the `Qt` namespace, while the second is from the `Policy` enumeration in the `QSizePolicy` class.

Although these enumerations can be specifed directly as integers, they are given the class `QtEnum` and have the overloaded operators | and & to combine values bitwise. This makes the most sense when the values correspond to bit flags, as is the case for the alignment style. For example, aligning the text in a label in the upper right can be done through

```
l <- Qt$QLabel("Our text")
l$alignment <- Qt$Qt$AlignRight | Qt$Qt$AlignTop
```

To check if the alignment is to the right, we could query by:

```
as.logical(l$alignment & Qt$Qt$AlignRight)
```

```
[1] FALSE
```

1.8 Extending Qt Classes from R

As Qt is implemented in an object-oriented language, C++, the designers of the API expect the developer to extend Qt classes, like `QWidget`, during the normal course of GUI development. This is a significant difference from GTK+, where it is only necessary to extend classes when one needs to fundamentally alter the behavior of a widget. The `qtbase` package allows the R user to extend C++ classes in order to enhance the features of Qt. The `qtbase` package includes functions `qsetClass` and `qsetMethod` to create subclasses and their methods. Methods may override virtual methods in an ancestor C++ class, and C++ code will invoke the R implementation when calling the overridden virtual. Properties may be defined with a getter and setter function. If a type is specified, and the class derives from `QObject`, the property will be exposed by Qt. It is also possible to store arbitrary objects in an instance of an R class; we will refer to these as dynamic fields. They are private to the class but are otherwise similar to attributes on any R object. Their type is not checked, and they are useful as a storage mechanism for implementing properties.

Defining a Class

Here, we show a generic example, and follow with a specific one.

```
qsetClass("SubClass", Qt$QWidget)
```

This creates a variable named `SubClass` in the workspace:

```
SubClass
```

```
Class 'R::.GlobalEnv::SubClass' with 319 public methods
```

Its value is an `RQtClass` object that behaves like the `RQtClass` for the built-in classes, such as `Qt$QWidget`. There are no static methods or enumerations in an R class, so the class object is essentially the constructor:

```
instance <- SubClass()
```

By default, the constructor delegates directly to the constructor in the parent class. A custom constructor is often useful, for example, to initialize fields or to make a compound widget. The function implementing the constructor should be passed as the constructor argument. By convention, QObject subclasses should provide a parent constructor argument, for specifying the parent object. A typical usage would be

```
qsetClass("SubClass2", Qt$QWidget,
          function(title, prop, parent=NULL) {
            super(parent)
            this$property <- prop
            setWindowTitle(title)
          })
```

Within the body of a constructor, the super variable refers to the constructor of the parent class, often called the "super" class. In the above, we call super to delegate the registration of the parent to the QWidget constructor. Another special symbol in the body of a constructor is this, which refers to the instance being constructed. We can set and implicitly create fields in the instance by using the same syntax as setting properties.

Defining Methods

One may define new methods, or override methods from a base class through the qsetMethod function. For example, accessors for a field may be defined with

```
qsetMethod("field", SubClass, function() field)
qsetMethod("setField", SubClass, function(value) {
  this$field <- value
})
```

For an override of an existing method to be visible from C++, the method must be declared virtual in C++. The access argument specifies the scope of the method: "public", "protected", or "private". These have the same meaning as in C++.

As with a constructor, the symbol this in a method definition refers to the instance. There is also a super function that behaves similarly to the super found in a constructor: it searches for an inherited method of a given name and invokes it with the passed arguments:

```
qsetMethod("setVisible", SubClass, function(value) {
  message("Visible: ", value)
  super("setVisible", value)
})
```

In the above, we intercept the setting of the visibility of our widget. If we hide or show the widget, we will receive a notification to the console:

```
instance$show()
```

This is somewhat similar to the behavior of `callNextMethod`, except `super` is not restricted to calling the same method.

Defining Signals and Slots

Two special types of methods are slots and signals, introduced earlier in the chapter. These exist only for `QObject` derivatives. Most useful are signals. Here we define a signal:

```
qsetSignal("somethingHappened", SubClass)
```

```
[1] "somethingHappened"
```

If the signal takes an argument, we need to indicate that in the signature:

```
qsetSignal("somethingHappenedAtIndex(int)", SubClass)
```

```
[1] "somethingHappenedAtIndex"
```

Writing a signature requires some familiarity with C/C++ types and syntax, but this is concise and consistent with how Qt describes its methods. Although almost always public, it is possible to make a signal protected or private, via the `access` argument.

Defining a slot is very similar to defining a signal, except a method implementation must be provided as an R function:

```
qsetSlot("doSomethingToIndex(int)", SubClass, function(index) {
  # ....
})
```

```
[1] "doSomethingToIndex"
```

The advantage of a slot compared to a method is that a slot is exposed to the Qt metaobject system. This means that a slot could be called from another dynamic environment, like from Javascript running in the `QScript` module or via the D-Bus through the `QDBus` module. It is also necessary to use slots as signal handlers for a GUI built with `QtDesigner`, if one is using the automated connection feature, see Section **??**.

Defining Properties

A property, introduced earlier, is a self-describing field that is encapsulated by a getter and a setter. We can define a property on any class using the `qsetProperty` function. Here is the simplest usage:

```
qsetProperty("property", SubClass)
```

```
[1] "property"
```

We can now access `property` like any other property; for example:

```
instance <- SubClass()
instance$property
```

```
NULL
```

```
instance$property <- "value"
instance$property
```

```
[1] "value"
```

However, the property is not actually exposed by the Qt meta object system. That requires specifying the `type` argument, which we will cover later.

By default, the property value is actually stored as a (private) field in the object, called `".property"`. One can override the default behavior by specifying a function for the getter and/or the setter:

```
qsetProperty("checkedProperty", SubClass, write = function(x) {
  if (!is(x, "character"))
    stop("'checkedProperty' must be a character vector")
  this$.checkedProperty <- x
})
```

```
[1] "checkedProperty"
```

We have taken advantage of the setter override to check the validity of the incoming value. If `"NULL"` is passed as the `write` argument, the property is read-only. One might also want to override the `read` function, for cases where a property depends only on other properties or on some external resource.

To automatically emit a signal whenever a property is set, one can pass the name of the signal as the `notify` of `qsetProperty`:

```
qsetSignal("propertyChanged", SubClass)
```

```
[1] "propertyChanged"
```

```
qsetProperty("property", SubClass, notify = "propertyChanged")
```

```
[1] "property"
```

If a class derives from `QObject`, as does any widget, we can specify the C++ type of the property to expose it to the Qt meta object system:

```
qsetProperty("typedProperty", SubClass, type = "QString")
```

```
tail(qproperties(SubClass()), 1)
```

```
                  type readable writable
typedProperty QString     TRUE     TRUE
```

We see that the type is now exposed via the general `qproperties` function. Specifying the type enables all of the features of a Qt property.

Example 1.1: A model for workspace objects
In this example, we revisit creating a backend storage model for a workspace browser (cf. Example **??**). With `gWidgets`, we needed to define an Observable class for our model. With Qt, we can leverage the existing signal framework to notify views of changes to the model. This example demonstrates only the model; implementing a view is left to the reader.

Our basic model subclasses `QObject`, not `QWidget`, as it has no graphical representation – a job left for its views:

```
qsetClass("WSModel", Qt$QObject, function(parent=NULL) {
  super(parent)
  updateVariables()
})
```

```
Class 'R::.GlobalEnv::WSModel' with 50 public methods
```

We have two main properties: a list of workspace objects and a digest hash for each, which we use for comparison purposes. The digest is generated by the `digest` package, which we load:

```
library(digest)
```

We store the digest in a property:

```
qsetProperty("digest", WSModel)
```

```
[1] "digest"
```

When a new object is added, an object is deleted, or an object is changed, we wish to signal that occurence to any views of the model. For that purpose, we define a new signal below:

```
qsetSignal("objectsChanged", WSModel)
```

We then specify this signal name to the `notify` argument when defining the `objects` property, so that assignment will emit the signal:

```
qsetProperty("objects", WSModel, notify="objectsChanged")
```

Our class has a method to update the variable list. This simply compares the digest of the current workspace objects with a cached list. If there are differences, we update the objects, which in turn signals a change.

```
qsetMethod("updateVariables", WSModel, function() {
  x <- sort(ls(envir=.GlobalEnv))
  objs <- sapply(x, function(i)
                 digest(get(i, envir=.GlobalEnv)))

  if((length(objs) != length(digest)) ||
     length(digest) == 0 ||
     any(objs != digest)) {
    this$digest <- objs       # update cache
    this$objects <- x         # emit signal
  }
  invisible()
})
```

To illustrate, we create a notifier that the obejcts were changed by assigning a callback:

```
m <- WSModel()
qconnect(m, "objectsChanged", function()
         message("workspace objects were updated"))
```

Finally, we arrange to call the update function as needed. If the workspace size is modest, using a task callback is a reasonable strategy:

```
addTaskCallback(function(expr, value, ok, visible) {
  m$updateVariables()
  TRUE
})
```

1.9 `QWidget` Basics

The widgets we discuss in the next section 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. It implements the event processing and property systems. The `QWidget` class is the base class for all widgets and implements their shared functionality.

Upon construction, widgets are invisible, so that they may be configured behind the scenes. The `visible` property controls whether a widget is visible.

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

```
[1] FALSE
```

```
w$visible <- TRUE
w$visible
```

```
[1] TRUE
```

The `show` and `hide` methods are the corresponding convenience functions for making a widget visible and invisible, respectively.

```
w$show()
```

```
w$visible
```

```
[1] TRUE
```

```
w$hide()
```

```
w$visible
```

```
[1] FALSE
```

There is an S3 method for `print` on `QWidget` that invokes `show`. Whenever a widget is shown, all of its children are also made visible. The method `raise` will raise the window to the top of the stack of windows.

Similary, the property `enabled` controls whether a widget is sensitive to user input, including mouse events.

```
b <- Qt$QPushButton("button")
b$enabled <- FALSE
b$enabled
```

```
[1] FALSE
```

Only one widget can have the keyboard focus at once. The user shifts the focus by tab-navigation or mouse clicks (unless customized, see `focusPolicy`). When a widget has the focus, its `focus` property is `TRUE`. The property is read-only; the focus may be shifted to a widget by calling its `setFocus` method.

Qt has a number of mechanisms for the user to query a widget for some description of its purpose and usage. Tooltips, stored as a string in the `toolTip` property, may be shown when the user hovers the mouse over the widget. Similarly, the `statusTip` property holds a string to be shown in the statusbar instead of a popup window. Finally, Qt provides a "What's This?" tool that will show the text in the `whatsThis` property in response to query, such as pressing SHIFT+F1 when the widget has focus.

Except for top-level windows, the position and size of a widget are determined automatically by a layout algorithm; see Chapter 2. To specify the size of a top-level window, manipulate the `size` property, which holds a `QSize` object:

```
w$size <- qsize(400, 400)
## or
w$resize(400, 400)
w$show()
```

We create the QSize object with the qsize convenience function implemented by the qtbase package. The resize method is another convenient shortcut. One should generally configure the size of a window before showing it. This helps the window manager optimally place the window.

Fonts

Fonts in Qt are represented by the QFont class. The qtbase package defines a convenience constructor for QFont called qfont. The constructor accepts 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 font should be italicized, as a logical. Defaults are obtained from the application font, returned by Qt$QApplication$font().

For example, we could create a 12 point, bold, italicized font from the helvetica family:

```
f <- qfont(family="helvetica", pointsize=12,
           weight=Qt$QFont$Bold, italic=TRUE)
```

The font for a widget is stored in the font property. For example, we change the font for a label:

```
l <- Qt$QLabel("Text for the label")
l$font$toString()
l$font <- f
l$font$toString()
```

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

Palette Every platform has its own distinct look and feel, and an application should conform to platform conventions. Qt hides these details from the application. Every widget has a palette, stored in its palette property and represented by a QPalette object. A QPalette maps the state of a widget to a group of colors that is used for painting the widget. The possible states are active, inactive and disabled. Each color within a group has specific role, as enumerated in QPalette::ColorRole. Examples include the color for background (Window), the foreground (WindowText) and the selected state (Highlight). Qt chooses the correct default palette depending on the platform and the type of widget. One can change the colors used in rendering a widget by manipulating the palette, as illustrated in Example **??**.

Style Sheets Cascading style sheets (CSS) are used by web deisgners to decouple the layout and look and feel of a web page from the content of the page. In Qt it is also possible to customize the rendering of a widget using CSS syntax. The supported syntax is described in the overview on stylesheets provided with Qt documentation and is not summarized here, as it is quite readable.

The style sheet for a widget is stored in its `styleSheet` property, as a string. For example, for a button, we could set the background to white and the foreground to red:

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

The CSS syntax may be unfamiliar to R programmers, so the `qtbase` package provides an alternative interface that is reminiscent of the `par` function. We specify the above stylesheet in this syntax:

```
qsetStyleSheet(color = "red", background = "white",
               what = "QPushButton", widget = b)
```

The `widget` argument defaults to `NULL`, which applies the stylesheet application-wide through the `QApplication` instance. The default for `what` is `"*"`, meaning that the stylesheet applies to any widget class. The following would cause all widgets in the application to have the same colors as the button:

```
qsetStyleSheet(color = "red", background = "white")
```

Example 1.2: A "error label"

This example extends the line edit widget to display an error state via an icon embedded within the entry box. Such a widget might prove useful when one is validating entered values. Our implementation uses a stylesheet to place the icon in the background and to prevent the text from overlapping the icon.

To indicate an error, we will add an icon and set the tooltip to display an informative message. The constructor will be the default, so our class is defined with:

```
qsetClass("LineEditWithError", Qt$QLineEdit)
```

The main method sets the error state. We use stye sheets to place an image to the left of the entry message and set the tooltip.

```
qsetMethod("setError", LineEditWithError, function(msg) {
  f <- system.file("images/cancel.gif", package="gWidgets")
  qsetStyleSheet("background-image" = sprintf("url(%s)", f),
                 "background-repeat" = "no-repeat",
                 "background-position" = "left top",
                 "padding-left" = "20px",
                 widget = this)
  setToolTip(msg)
})
```

```
[1] "setError"
```

We can clear the error by resetting the properties to `NULL`.

```
qsetMethod("clearError", LineEditWithError, function() {
  setStyleSheet(NULL)
  setToolTip(NULL)
})
```

1.10 Importing a GUI from QtDesigner

`QtDesigner` is a tool for graphical, drag-and-drop design of GUI forms. Although this book focuses on constructing a GUI by programming in R, we recognize that a graphical approach may be preferable in some circumstances. `QtDesigner` outputs a GUI definition as an XML file in the `"UI"` format. The `QUiLoader` class loads a `"UI"` definition through its `load` method:

```
loader <- Qt$QUiLoader()
widget <- loader$load(Qt$QFile("designer.ui"))
```

The `widget` object could be shown directly; however, we first need to implement the behavior of the GUI by connecting to signals. Through the `QtDesigner` GUI, the user can connect signals to slots on built-in widgets. This works for some trivial cases, but in general one needs to handle signals with R code. There are two ways to accomplish this: manual or automatic.

To manually connect an R handler to a signal, we first need to obtain the widget with the signal. Every widget in a `UI` file is named, so we can call the `qfindChild` utility function to find a specific widget. Assume we have a button named "findButton" and corresponding text entry "findEntry" in our UI file, then we retrieve them with

```
findButton <- qfindChild(widget, "findButton")
findEntry <- qfindChild(widget, "findEntry")
```

Then we connect to the `clicked`QPushButton signal:

```
qconnect(findButton, "clicked", function() {
  findText(findEntry$text)
})
```

Alternatively, we could establish the signal connections automatically. This requires defining each signal handler to be a slot in the parent object, which will need to be of a custom class:

```
qsetClass("MyMainWindow", Qt$QWidget, function() {
  Qt$QMetaObject$connectSlotsByName(this)
})
```

```
Class 'R::.GlobalEnv::MyMainWindow' with 318 public methods
```

The constructor calls `connectSlotsByName` to automatically establish the connections. For a slot to be connected to the correct signal, it must be named according to the convention `"on_[objectName]_[signalName]"`. For example,

```
qsetSlot("on_findButton_clicked", MyMainWindow, function() {
  findText(findEntry$text)
})
```

In the case of a large, complex GUI, this automatic approach is probably more convenient than manually establishing the connections.

2

Qt: Layout Managers and Containers

Qt provides a set of classes to faciliate the layout of child widgets of a component. These layout managers, derived from the QLayout class, are tasked with determining the geometry of child widgets, according to a specific layout algorithm. Layout managers will generally update the layout whenever a parameter is modified, a child widget is added or removed, or the size of the parent changes. Unlike GTK+, where this management is tied to a container object, Qt decouples the layout from the widget.

This chapter will introduce the available layout managers, of which there are three types: the box (QBoxLayout), grid (QGridLayout) and form (QFormLayout). Widgets that function primarily as containers, such as the frame and notebook, are also described here.

Example 2.1: Synopsis of Layouts in Qt

This example uses a combination of different layout managers to organize a reasonably complex GUI. It serves as a synopsis of the layout functionality in Qt. A more gradual and detailed introduction will follow this example. Figure 2.1 shows a screenshot of the finished layout.

First, we need a widget as the top-level container. We assign a grid layout to the window for arranging the main components of the application:

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

There are three objects managed by the grid layout: a table (we use a label as a placeholder), a notebook, and a horizontal box layout for some buttons. We construct them

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

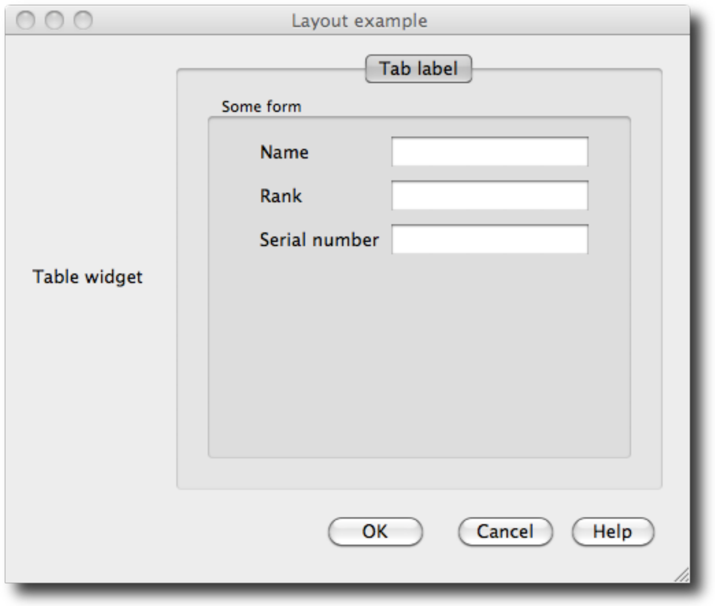


Figure 2.1: A mock GUI illustrating various layout managers provided by Qt.

and add them to the grid layout:

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

Next, we construct our buttons and add them to the box putting 12 pixels of space between the last two.

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

We added a stretch, which acts much like a spring, to pack our buttons against the right side of the box. A fixed space of 12 pixels is inserted between the “Cancel” and ”help” buttons.

The notebook widget is constructed next, with a single page:

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

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

The form layout allows us to create standardized forms where each row contains a label and a widget. Although this could be done with a grid layout, using the form layout is more convenient, and allows Qt to style the page as appropriate for the underlying operating system. We place a form layout in the notebook page and populate it:

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

Each `addRow` call adds a label and an adjacent input widget, in this case a text entry.

This includes our cursory demonstration of layout in Qt. We have constructed a mock-up of a typical application layout using the box, grid and form layout managers.

2.1 Layout Basics

Adding and Manipulating Children

We will demonstrate the basics of layout in Qt with a horizontal box layout, `QHBoxLayout`:

```
layout <- Qt$QHBoxLayout()
```

`QHBoxLayout`, like all other layouts, is derived from the `QLayout` base class. Details specific to box layouts are presented in Section 2.2.

A layout is not a widget. Instead, a layout is set on a widget, and the widget delegates the layout of its children to the layout object:

```
wid <- Qt$QWidget()
wid$setLayout(layout)
```

Child widgets are added to a container through the `addWidget` method:

```
layout$addWidget(Qt$QPushButton("Push Me"))
```

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

Internally, layouts store child components as items of class `QLayoutItem`. The item at a given zero-based index is returned by `itemAt`. We get the first item in our layout:

```
item <- layout$itemAt(0)
```

The actual child widget is retrieved by calling the `widget` method on the item:

```
button <- item$widget()
```

Qt provides the methods `removeItem` and `removeWidget` to remove an item or widget from a layout:

```
layout$removeWidget(button)
```

Although the widget is no longer managed by a layout, its parent widget is unchanged. The widget will not be destroyed (removed from memory) as long as it has a parent. Thus, to destroy a widget, one should set the parent of the widget `NULL` using `setParent`:

```
button$setParent(NULL)
```

Size and Space Negotiation

The allocation of space to child widgets depends on several factors. The Qt documentation for layouts spells out well the steps: [1]

1. All the widgets will initially be allocated an amount of space in accordance with their `sizePolicy` and `sizeHint`.
2. If any of the widgets have stretch factors set, with a value greater than zero, then they are allocated space in proportion to their stretch factor.
3. If any of the widgets have stretch factors set to zero they will only get more space if no other widgets want the space. Of these, space is allocated to widgets with an expanding size policy first.
4. Any widgets that are allocated less space than their minimum size (or minimum size hint if no minimum size is specified) are allocated this minimum size they require. (Widgets don't have to have a minimum size or minimum size hint in which case the strech factor is their determining factor.)
5. Any widgets that are allocated more space than their maximum size are allocated the maximum size space they require. (Widgets do not have to have a maximum size in which case the strech factor is their determining factor.)

Every widget returns a size hint to the layout from the the `size-Hint` method/property. The interpretation of the size hint depends on the `sizePolicy` property. The size policy is an object of class `QSizePolicy`. It contains a separate policy value, taken from the `QSizePolicy` enumeration, for the vertical and horizontal directions. If a layout is set on a widget, then the widget inherits its size policy from the layout. The possible size policies are listed in Table **??**.

[1]`http://doc.qt.nokia.com/4.6/layout.html`

Table 2.1: Possible size policies

Policy	Meaning
Fixed	to require the size hint exactly
Minimum	to treat the size hint as the minimum, allowing expansion
Maximum	to treat the size hint as the maximum, allowing shrinkage
Preferred	to request the size hint, but allow for either expansion or shrinkage
Expanding	to treat like `Preferred`, except the widget desires as much space as possible
MinimumExpanding	to treat like `Minimum`, except the widget desires as much space as possible
Ignored	to ignore the size hint and request as much space as possible

tab:qt:size-policies As an example, consider `QPushButton`. It is the convention that a button will only allow horizontal, but not vertical, expansion. It also requires enough space to display its entire label. Thus a `QPushButton` instance returns a size hint depending on the label dimensions and has the policies `Fixed` and `Minimum` as its vertical and horizontal policies respectively. We could prevent a button from expanding at all:

```
b <- Qt$QPushButton("No expansion")
b$setSizePolicy(vertical=Qt$QSizePolicy$Fixed,
                horizontal=Qt$QSizePolicy$Fixed)
```

Thus, the sizing behavior is largely inherent to the widget or its layout, if any, rather than any parent layout parameters. This is a major difference from GTK+, where a widget can only request a minimum size and all else depends on the parent container widget. The Qt approach seems better at encouraging consistency in the layout behavior of widgets of a particular type.

Most widgets attempt to fill the allocated space; however, this is not always appropriate, as in the case of labels. In such cases, the widget will not expand and needs to be aligned within its space. By default, the widget is centered. We can control the alignment of a widget via the `setAlignment` method. For example, we align the label to the left side of the layout:

```
label <- Qt$QLabel("Label")
layout$addWidget(label)
layout$setAlignment(label, Qt$Qt$AlignLeft)
```

Alignment is also possible to the top, bottom and right. The alignment values are flags andmay be combined with | to specify both vertical and horizontal alignment.

The spacing between every cell of the layout is in the `spacing` property, the following requests 5 pixels of space:

```
layout$spacing <- 5L
```

2.2 Box Layouts

Box layouts arrange child widgets as if they were packed into a box in either the horizontal or vertical orientation. The `QHBoxLayout` class implements a horizontal layout whereas `QVBoxLayout` provides a vertical one. Both of these classes extend the `QBoxLayout` class, where most of the functionality is documented. We create a horizontal layout:

```
hb <- Qt$QHBoxLayout()
```

Child widgets are added to a box container through the `addWidget` method:

```
buttons <- sapply(letters[1:3], Qt$QPushButton)
sapply(buttons, hb$addWidget)
```

The `direction` property specifies the direction in which the widgets are added to the layout. By default, this is left to right (top to bottom for a vertical box).

The `addWidget` method for a box layout takes two optional parameters: the stretch factor and the alignment. Stretch factors proportionally allocate space to widgets when they expand. For those familiar with GTK+, the difference between a stretch factor of 0 and 1 is roughly equivalent to the difference between `"FALSE"` and `"TRUE"` for the value of the `expand` parameter to `gtkBoxPackStart`. However, recall that the widget size policy and hint can alter the effect of a stretch factor. After the child has been added, the stretch factor may be modified with `setStretchFactor`:

```
hb$setStretchFactor(wid, 2.0)
```

```
[1] FALSE
```

Spacing There are two types of spacing between two children: fixed and expanding. Fixed spacing between any two children was already described. To add a fixed amount of space between two specific children, call the `addSpacing` method while populating the container. The following line is from Example 2.1:

```
buttonLayout$addSpacing(12L)
```

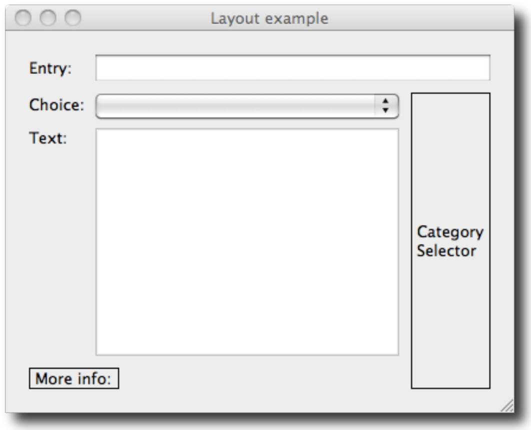


Figure 2.2: A mocked up layout using the `QGridLayout` class.

An expanding, spring-like spacer between two widgets is known as a *stretch*. We add a stretch with a factor of 2.0 and subsequently add a child button that will be pressed against the right side of the box:

```
g$addStretch(2)
g$addWidget(Qt$QPushButton("Help..."))
```

This is just a convenience for adding an invisible widget with some stretch factor.

Struts It is sometimes desirable to restrict the minimum size of a layout in the perpendicular direction. For a horizontal box, this is the height. The box layout provides the *strut* for this purpose:

```
g$addStrut(10)                    # atleast 10 pixels high
```

```
NULL
```

2.3 Grid Layouts

The `QGridLayout` class provides a grid layout for aligning its child widgets into rows and columns. To illustrate grid layouts we mock up a GUI centered around a text area widget (Figure 2.2). To begin, we create the window with the grid layout:

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

When we add a child to the grid layout, we need to specify the zero-based row and column indices:

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

In the second call to `addWidget`, we pass values to the optional arguments for the row and column span. These are the numbers of rows and columns, respectively, that are spanned by the child. For our second row, we add a labeled combo box:

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

The bottom three cells in the third column are 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:

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

Since this widget will expand, we align the label to the top of its cell. Otherwise, it will be centered in the vertical direction.

Finally we add a space for information on the fourth row:

```
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)
```

Again we draw a frame around the label. By default the box would expand to fill the space of the two columns, but we prevent this through a `"Fixed"` size policy.

There are a number of parameters controlling the sizing and spacing of the rows and columns. The concepts apply equivalently to both rows and columns, so we will limit our discussion to columns, without loss of generality. A minimum width is set through `setColumnMinimumWidth`. The actual minimum width will be increased, if necessary, to satisfy the minimal width requirements of the widgets in the column. If more space is available to a column than requested, the extra space is apportioned according to the `stretch` factors. A column stretch factor is set by calling the `setColumnStretch` method.

Since there are no stretch factors set in our example, the space allocated to each row and column would be identical when resized. To allocate extra space to the text area, we set a positive stretch factor for the third row and second column:

```
lyt$setRowStretch(2, 1)                 # third row
lyt$setColumnStretch(1,1)               # second column
```

As it is the only item with a positive stretch factor, it will be the only widget to expand when the parent widget is resized.

The spacing between widgets can be set in both directions via the `spacing` property, or set for a particular direction with `setHorizontalSpacing` or `setVerticalSpacing`. The default values are derived from the style.

The method `itemAtPosition` returns the `QLayoutItem` instance corresponding to the specified row and column:

```
lineEdit <- lyt$itemAtPosition(0, 1)$widget()
```

The item method `widget` returns the corresponding widget. Removing a widget is similar to a box layout, using `removeItem` or `removeWidget`. The methods `rowCount` and `columnCount` return the dimensions of the grid.

2.4 Form layouts

Forms can easily be arranged with the grid layout, but Qt provides a convenient high-level form layout (`QFormLayout`) that conforms to platform-specific conventions. A form consists of a number of rows, where each row has a label and an input widget. We create a form and add some rows for gathering parameters to the `dnorm` function:

```
w <- Qt$QWidget()
w$setWindowTitle("Wrapper for 'dnorm' function")
w$setLayout(flyt <- Qt$QFormLayout())
flyt$addRow("quantile", Qt$QLineEdit())
flyt$addRow("mean", Qt$QLineEdit())
flyt$addRow("sd", Qt$QLineEdit())
flyt$addRow(Qt$QCheckBox("log"))
```

The first three calls to `addRow` take a string for the label and a text entry for entering a numeric value. Any widget could serve as the label. A field may be any widget or layout. The final call to `addRow` places only a single widget in the row. As with other layouts, we could call `setSpacing` to adjust the spacing between rows.

To retrieve a widget from the layout, call the `itemAt` method, passing the zero-based row index and the role of the desired widget. Here, we obtain the edit box for the quantile parameter:

```
quantileEdit <- flyt$itemAt(0, Qt$QFormLayout$FieldRole)
```

2.5 Frames

The frame widget, QGroupBox, groups conceptually related widgets by drawing a border around them and displaying a title. QGroupBox is often used to group radio buttons, see Section 3.5 for an example. The title, stored in the title property, may be aligned to left, right or center, depending on the alignment property, see Figure **??**. If the checkable property is "TRUE", the frame contents can be enabled or disabled by clicking a check button.

2.6 Separators

Like frames, a horizontal or vertical line is also useful for visually separating widgets into conceptual groups. There is no explicit line or separator widget in Qt. Rather, one configures the more general widget QFrame, which draws a frame around its children. Somewhat against intuition, a frame can take the shape of a line:

```
separator <- Qt$QFrame()
separator$frameShape <- Qt$QFrame$HLine
```

This yields a horizontal separator. A frame shape of Qt$QFrame$VLine would produce a vertical separator.

2.7 Notebooks

A notebook container is provided by the class QTabWidget:

```
nb <- Qt$QTabWidget()
```

To create a page, one needs to specify the label for the tab and the widget to display when the page is active:

```
nb$addTab(Qt$QPushButton("page 1"), "page 1")
icon <- Qt$QIcon("small-R-logo.jpg")
nb$addTab(Qt$QPushButton("page 2"), icon,  "page 2")
```

As shown in the second call to addTab, one can provide an icon to display next to the tab label. We can also add a tooltip for a specific tab, using zero-based indexing:

```
nb$setTabToolTip(0, "This is the first page")
```

The currentIndexroperty holds the zero-based index of the active tab. We make the second tab active:

```
nb$currentIndex <- 1
```

The tabs can be positioned on any of the four sides of the notebook; this depends on the tabPositionroperty. By default, the tabs are on top, or "North". We move them to the bottom:

```
nb$tabPosition <- Qt$QTabWidget$South
```

Other features include close buttons, movable pages by drag and drop, and scroll buttons for when the number of tabs exceeds the available space. We enable all of these:

```
nb$tabsClosable <- TRUE
qconnect(nb, "tabCloseRequested", function(index) {
  nb$removeTab(index)
})
nb$movable <- TRUE
nb$usesScrollButtons <- TRUE
```

We need to connect to the `tabCloseRequested` signal to actually close the tab when the close button is clicked.

Example 2.2: A help page browser
This example shows how to create a help browser using the `QWebView` class to show web pages. The only method from this class we use is `setUrl`. The key to this is informing `browseURL` to open web pages using an R function, as opposed to the default system browser.

```
qsetClass("HelpBrowser", Qt$QTabWidget, function(parent=NULL) {
  super(parent)
  #
  this$tabsClosable <- TRUE
  qconnect(this, "tabCloseRequested", function(index) {
    this$removeTab(index)
  })
  this$movable <- TRUE; this$usesScrollButtons <- TRUE
  #
  this$browser <- getOption("browser")
  f <- function(url) openPage(url)
  options("browser" = f)
})
```

```
Class 'R::.GlobalEnv::HelpBrowser' with 367 public methods
```

The lone method is that called to open page.

```
qsetMethod("openPage", HelpBrowser, function(url) {
  nm <- strsplit(url, "/")[[1]]
  nm <- sprintf("%s: %s", nm[length(nm)-2], nm[length(nm)])
  w <- Qt$QWebView()
  w$setUrl(Qt$QUrl(url))
  i <- addTab(w, nm)
})
```

```
[1] "openPage"
```

General Widget Stacking It is sometimes useful to have a widget that only shows one of its widgets at once, like a QTabWidget, except without the tabs. There is no way to hide the tabs of QTabWidget. Instead, one should use QStackedWidget, which stacks its children so that only the widget on top of the stack is visible. There is no way for the user to switch between children; it must be done programmatically. The actual layout is managed by QStackedLayout, which should be used directly if only a layout is needed, e.g., as a sub-layout.

2.8 Scroll Areas

Sometimes a widget is too large to fit in a layout and thus must be displayed partially. Scroll bars then allow the user to adjust the visible portion of the widget. Widgets that often become too large include tables, lists and text edit panes. These inherit from QAbstractScrolledArea and thus natively provide scroll bars without any special attention from the user. Occasionally, we are dealing with a widget that lacks such support and thus need to explicitly embed the widget in a QScrollArea. This often arises when displaying graphics and images. To demonstrate, we will create a simple zoomable image viewer. The user can zoom in and out and use the scroll bars to pan around the image. First, we place an image in a label and add it to a scroll area:

```
image <- Qt$QLabel()
image$pixmap <- Qt$QPixmap("someimage.png")
sa <- Qt$QScrollArea()
sa$setWidget(image)
```

Next, we define a function for zooming the image:

```
zoomImage <- function(x = 2.0) {
  image$resize(x * image$pixmap$size())
  updateScrollBar <- function(sb) {
    sb$value <- x * sb$value + (x - 1) * sb$pageStep / 2
  }
  updateScrollBar(sa$horizontalScrollBar())
  updateScrollBar(sa$verticalScrollBar())
}
```

Of note here is that we are scaling the size of the pixmap using the * function, which qtbase is forwarding to the corresponding method on the QSize object. Updating the scroll bars is also somewhat tricky, since their value corresponds to the top-left, while we want to preserve the center point. We leave the interface for calling the zoomImage function as an exercise for the interested reader.

The geometry of a scroll area is such that there is an empty space in the corner between the ends of the scroll bars. To place a widget in the corner, pass it to the `setCornerWidget` method.

2.9 Paned Windows

`QSplitter` is a split pane widget, a container that splits its space between its children, with draggable separators that adjust the balance of the space allocation.

Unlike `GtkPaned` in GTK+, there is no limit on the number of child panes. We add three and change the orientation from horizontal to vertical:

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

We can adjust the sizes programmatically:

```
sp$setSizes(c(100L, 200L, 300L))
```

3

Qt: Widgets

This chapter covers some of the basic dialogs and widgets provided by Qt. Together with layouts, these form the basis for most user interfaces. The next chapter will introduce the more complex widgets that typically act as a view for a separate data model.

3.1 Dialogs

Qt implements the conventional high-level dialogs, including those for printing, selecting files, selecting colors, and, most usefully, sending simple messages and input requests to the user. We first introduce message and input dialogs. This is followed by a discussion of the infrastructure in Qt for implementing custom dialogs and wizards. Finally, we briefly introduce some of the remaining high-level dialogs, such as the file selector.

Message Dialogs

All dialogs in Qt are derived from `QDialog`. The message dialog, `QMessageBox`, communicates a textual message to the user. At the bottom of the dialog are a set of buttons, each representing a possible response. Normally, the type of message is indicated by an icon. If extra details are available, the dialog provides the option for the user to view them.

Qt provide two ways to create a message box. The simplest approach is to call a static convenience method for issuing common types of messages, including warnings and simple questions. The alternative, described later, involves several steps and offers more control at a cost of convenience. Here we take the simple path for presenting a warning dialog:

```
response <- Qt$QMessageBox$warning(parent = NULL,
              title = "Warning!", text = "Warning message...")
```

This call will block the flow of the program until the user responds and returns the standard identifier for the button that was clicked. Each type of button corresponds to a fixed type of response. The standard button/response codes are listed in the `QMessageBox::StandardButton` enumeration. In this case, there is only a single button, `"QMessageBox$Ok"`. The dialog is *modal*, meaning that the user cannot interact with the `"parent"` window until responding. If the `"parent"` is `"NULL"`, as in this case, input to all windows is blocked. The dialog is automatically positioned near its parent, and if the parent is destroyed, the dialog is destroyed, as well. Additional arguments specify the available buttons/responses and the default response. We have relied on the default values for these.

For more control over the appearance and behavior of the dialog, we will take a more gradual path. First, we construct an instance of `QMessageBox`. It is possible to specify several properties at construction. Here is how one might construct a warning dialog:

```
dlg <- Qt$QMessageBox(icon = Qt$QMessageBox$Warning,
                      title = "Warning!",
                      text = "Warning text...",
                      buttons = Qt$QMessageBox$Ok,
                      parent = NULL)
```

This call introduces the `icon` property, which is a code from the `QMessageBox::Icon` enumeration and identifies a standard, themeable icon. The icon also implies the message type, just as a button implies a response type. We also need to specify the possible responses with the `"buttons"` argument.

Our dialog is already sufficiently complete to be displayed. However, we have the opportunity to specify further properties. Two of the most useful are `informativeText` and `detailedText`:

```
dlg$informativeText <- "Less important warning information"
dlg$detailedText <- "Extra details most do not care to see"
```

Both provide additional textual information at an increasing level of detail. The `informativeText`QMessageBox will be rendered as secondary to the actual message text. To display the `detailedText`, the user will need to interact with a control in the dialog. An example is a stack trace for the warning.

After specifying the desired properties, the dialog is shown. The approach to showing the dialog depends on whether the dialog should be modal. A modal dialog is displayed with the `exec` method.

```
dlg$exec()
```

```
[1] 1024
```

As its name implies, `exec` executes a loop that will block until the user responds. As with the static convenience methods, the return value indicates the button/response.

To present a non-modal dialog, we first need to register a response listener, as the response will arrive asynchronously:

```
qconnect(dlg, "finished", function(response) {
  ## handle response
  dlg$close()
})
```

```
QObject instance
```

There are several signals that indicate user response, including "finished", "accepted", and "rejected". The most general is "finished", which passes the button/response code as its only argument.

Then we show, raise and activate the dialog:

```
dlg$show()
dlg$raise()
dlg$activateWindow()
```

Modal dialogs may be window modal (`Qt$Qt$WindowModal`), where the dialog blocks all access to its ancestor windows, or application modal (`Qt$Qt$ApplicationModal`) (the default) where all windows are blocked. To specify the type of modality, call `setWindowModality`.

To summarize, we present a general message box, supporting multiple responses:

```
dlg <- Qt$QMessageBox()
dlg$windowTitle <- "[This space for rent]"
dlg$text <- "This is the main text"
dlg$informativeText <- "This should give extra info"
dlg$detailedText <- "And this provides\neven more detail"
dlg$icon <- Qt$QMessageBox$Critical
dlg$standardButtons <- Qt$QMessageBox$Cancel | Qt$QMessageBox$Ok
## 'Cancel' instead of 'Ok' is the default
dlg$setDefaultButton(Qt$QMessageBox$Cancel)
if(dlg$exec() == Qt$QMessageBox$Ok)
  print("A Ok")
```

Input dialogs

The `QInputDialog` class provides a convenient means to gather information from the user and is in a sense the inverse of `QMessageBox`. Possible input modes include selecting a value from a list or entering text or numbers. By default, input dialogs consist of an input control, an icon, and two buttons: "Ok" and "Cancel".

Like `QMessageBox`, one can display a `QInputDialog` either by calling a static convenience method or by constructing an instance and configuring it before showing it. We demonstrate the former approach for a dialog that requests textual input:

```
text <- Qt$QInputDialog$getText(parent = NULL,
                        title = "Gather text",
                        label = "Enter some text")
```

The return value is the entered string, or `NULL` if the user cancelled the dialog. Additional parameters allow one to specify the initial text and to override the input mode, e.g., for password-style input.

We can also display a dialog for integer input. Here, we ask the user for an even integer between 1 and 10:

```
num <- Qt$QInputDialog$getInt(parent = NULL,
                       title = "Gather integer",
                       label = "Enter an integer from 1 to 10",
                       value = 0, min = 2, max = 10, step = 2)
```

The number is chosen using a bounded spin box. To request a real value, call `Qt$QInputDialog$getDouble` instead.

The final type of input is selecting an option from a list of choices:

```
item <- Qt$QInputDialog$getItem(parent = NULL,
                        title = "Select item",
                        label = "Select a letter",
                        items = LETTERS, current = 17)
```

The dialog contains a combo box filled with the capital letters. The initial choice is 0-based index 17, or the letter "R". The chosen string is returned.

`QInputDialog` has a number of options that cannot be specified via one of the static convenience methods. These option flags are listed in the `QInputDialog$InputDialogOption` enumeration and include hiding the "Ok" and "Cancel" buttons and selecting an item with a list widget instead of a combo box. If such control is necessary, we must explicitly construct a dialog instance, configure it, execute it and retrieve the selected item.

```
dlg <- Qt$QInputDialog()
dlg$setWindowTitle("Select item")
dlg$setLabelText("Select a letter")
dlg$setComboBoxItems(LETTERS)
dlg$setOptions(Qt$QInputDialog$UseListViewForComboBoxItems)
```

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

```
[1] "A"
```

Button boxes

Before discussing custom dialogs, we first introduce the `QDialogButton-Box` utility for arranging dialog buttons in a consistent and cross-platform

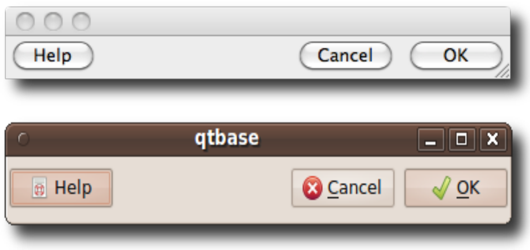


Figure 3.1: Dialog button boxes and their implementation under Mac OS X and Linux.

manner. Dialogs often have a standard button placement that varies among desktop environments. `QDialogButtonBox` is a container of buttons that arranges its children according to the convention of the platform. We place some standard buttons into a button box:

```
db <- Qt$QDialogButtonBox(Qt$QDialogButtonBox$Ok |
                          Qt$QDialogButtonBox$Cancel |
                          Qt$QDialogButtonBox$Help)
```

Figure 3.1 shows how the buttons are displayed on two different operating systems. To indicate the desired buttons, we pass a combination of flags from the `QDialogButtonBox$StandardButton` enumeration. Each standard button code implies a default label and role, taken from the `QDialog-ButtonBox$ButtonRole` enumeration. In the above example, we added a standard `OK` button, with the label "OK" (depending on the language) and the role `AcceptRole`. The `Cancel` button has the appropriate label and `Can-celRole` as its role. Icons are also displayed, depending on the platform and theme. The benefits of using standard buttons include convenience, standardization, platform consistency, and automatic translation of labels.

To respond to user input, one can connect directly to the `clicked` signal on a given button. It is often more convenient, however, to connect to one of the high-level button box signals, which include: `accepted`, which is emitted when a button with the `AcceptRole` or `YesRole` is clicked; `re-jected`, 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 the button object.

```
qconnect(db, "accepted", function() message("accepted"))
qconnect(db, "rejected", function() message("rejected"))
qconnect(db, "helpRequested", function() message("help"))
qconnect(db, "clicked", function(button) message(button$text))
```

The first button added with the `AcceptRole` role is made the default. Overriding this requires adding the default button with `addButton` and setting the `defaultroperty` on the returned button object.

Custom Dialogs

Every dialog in Qt inherits from `QDialog`, which we can leverage for our own custom dialogs. One approach is to construct an instance of `QDialog` and add arbitrary widgets to its layout. However, we suggest an alternative approach: extend `QDialog` or one of its derivates and implement the custom functionality in a subclass. This more formally encapsulates the state and behavior of the custom dialog. We demonstrate the subclass approach by constructing a dialog that requests a date from the user.

We begin by defining our class and its constructor:

```
qsetClass("DateDialog", Qt$QDialog,
          function(parent = NULL) {
            super(parent=parent)
            setWindowTitle("Choose a date")
            this$calendar <- Qt$QCalendarWidget()
            #
            buttonBox <- Qt$QDialogButtonBox(Qt$QMessageBox$Cancel |
                                      Qt$QMessageBox$Ok)
            qconnect(buttonBox, "accepted", function() {
              this$close()
              this$setResult(Qt$QMessageBox$Ok)
            })
            qconnect(buttonBox, "rejected", function() this$close())
            #
            layout <- Qt$QVBoxLayout()
            sapply(list(calendar, buttonBox), layout$addWidget)
            setLayout(layout)
          })
```

```
Class 'R::.GlobalEnv::DateDialog' with 336 public methods
```

Our dialog consists of a calendar, implemented by the `QCalendarWidget`, and a set of response buttons, organized by a `QDialogButtonBox`. The calendar is stored as a field on the instance, so that we can retrieve the selected date upon request.

We define a method that gets the currently selected date:

```
qsetMethod("selectedDate", DateDialog,
           function(x) calendar$selectedDate$toString())
```

```
[1] "selectedDate"
```

`DateDialog` can be executed like any other `QDialog`:

```
dateDialog <- DateDialog()
if (dateDialog$exec())
  message(dateDialog$selectedDate())
```

Wizards

`QWizard` implements a wizard – a multipage dialog that guides the user through a sequential, possibly branching process. Wizards are composed of pages, and each page has a consistent interface, usually including buttons for moving backwards and forwards through the pages. The look and feel of a `QWizard` is consistent with platform conventions.

We create a wizard object and set its title:

```
wizard <- Qt$QWizard()
wizard$setWindowTitle("A wizard")
```

Each page is represented by a `QWizardPage`. We create one for requesting the age of the user and add the page to the wizard:

```
getAgePage <- Qt$QWizardPage(wizard)
getAgePage$setTitle("What is your age?")
lyt <- Qt$QFormLayout()
getAgePage$setLayout(lyt)
lyt$addRow("Age", (age <- Qt$QLineEdit()))
wizard$addPage(getAgePage)
```

Two more pages are added:

```
getToysPage <- Qt$QWizardPage(wizard)
getToysPage$setTitle("What toys do you like?")
lyt <- Qt$QFormLayout()
getToysPage$setLayout(lyt)
lyt$addRow("Toys", (toys <- Qt$QLineEdit()))
wizard$addPage(getToysPage)
getGamesPage <- Qt$QWizardPage(wizard)
getGamesPage$setTitle("What games do you like?")
lyt <- Qt$QFormLayout()
getGamesPage$setLayout(lyt)
lyt$addRow("Games", (games <- Qt$QLineEdit()))
wizard$addPage(getGamesPage)
```

Finally, we run the wizard by calling its `exec` method:

```
ret <- wizard$exec()
```

File and Directory Choosing Dialogs

`QFileDialog` allows the user to select files and directories, by default using the platform native file dialog. As with other dialogs there are static methods to create dialogs with standard options. These are `"getOpenFileName"`, "getOpenFileNames", "getExistingDirectory", and "getSaveFileName". To select a file name to open we would have:

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

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

To allow multiple selection, call the plural form of the method:

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

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())
```

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:

```
nameFilter <- paste("R files (*.R .RData)",
                  "Sweave files (*.Rnw)",
                  "All files (*.*)",
                  sep=";;")
#
fnames <- Qt$QFileDialog$getOpenFileNames(NULL,
             "Open file(s)...", getwd(), nameFilter)
```

Although the static functions provide most of the functionality, to fully configure a dialog, it may be necessary to explicitly construct and manipulate a dialog instance. Examples of options not available from the static methods are history (previously selected file names), sidebar shortcut URLs, and filters based on low-level file attributes like permissions.

Example 3.1: File dialogs
We construct a dialog for opening an R-related file, using the file names selected above as the history:

```
dlg <- Qt$QFileDialog(NULL, "Choose an R file", getwd(), nameFilter)
dlg$fileMode <- Qt$QFileDialog$ExistingFiles
dlg$setHistory(fnames)
```

The dialog is executed like any other. To get the specified files, call selectedFiles:

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

Other Choosers

Qt provides several additional dialog types for choosing a particular type of item. These include QColorDialog for picking a color, and QFontDialog for selecting a font. These special case dialogs will not be discussed further here.

3.2 Labels

As seen in previous example, basic labels in Qt are instances of the QLabel class. Labels in Qt are the primary means for displaying static text and images. Textual labels are the most common, and the constructor accepts a string for the text, which can be plain text or, for rich text, HTML. Here we use HTML to display red text:

```
l <- Qt$QLabel("<font color='red'>Red</font>")
```

By default, QLabel guesses whether the string is rich or plain text. In the above, the rich text format is identified from the markup. The textFormat property overrides this.

The label text is stored in the text property. Properties relevant to text layout include: alignment, indent (in pixels), margin, and wordWrap.

3.3 Buttons

As we have seen, the ordinary button in Qt is created by QPushButton, which inherits most of its functionality from QAbstractButton, the common base class for buttons. We create a simple “Ok” button:

```
button <- Qt$QPushButton("Ok")
```

Like any other widget, a button may be disabled, so that the user cannot press it:

```
button$enabled <- FALSE
```

This is useful for preventing the user from attempting to execute commands that do not apply to the current state of the application. Qt changes the rendering widget, including that of the icon, to indicate the disabled state.

A push button usually executes some command when clicked, i.e., when the clicked signal is emitted:

```
qconnect(button, "clicked", function() message("Ok clicked") )
```

Icons and pixmaps

A button is often decorated with an icon, which serves as a visual indicator of the purpose of the button. The `QIcon` class represents an icon. Icons may be defined for different sizes and display modes (normal, disabled, active, selected); however, this is often not necessary, as Qt will automatically adapt an icon as necessary. As we have seen, Qt automatically adds the appropriate icon to a standard button in a dialog. When using `QPushButton` directly, there are no such conveniences. For our "Ok" button, we could add an icon from a file:

```
iconFile <- system.file("images/ok.gif", package="gWidgets")
button$icon <- Qt$QIcon(iconFile)
```

However, in general, this will not be consistent with the current style. Instead, we need to get the icon from the `QStyle`:

```
style <- Qt$QApplication$style()
button$icon <- style$standardIcon(Qt$QStyle$SP_DialogOkButton)
```

The `"QStyle::StandardPixmap"` enumeration lists all of the possible icons that a style should provide. In the above, we passed the key for an "Ok" button in a dialog.

We can also create a `QIcon` from image data in a `QPixmap` object. `QPixmap` stores an image in a manner that is efficient for display on the screen [1]. One can load a pixmap from a file or create a blank image and draw on it using the Qt painting API (not discussed in this book). Also, using the `qtutils` package, we can draw a pixmap using the R graphics engine. For example, the following uses `ggplot2` to generate an icon representing a histogram. First, we create the Qt graphics device and plot the icon with `grid`:

```
require(qtutils)
device <- QT()
grid.newpage()
grid.draw(GeomHistogram$icon())
```

Next, we create the blank pixmap and render the device to a paint context attached to the pixmap:

```
pixmap <- Qt$QPixmap(device$size$toSize())
pixmap$fill()
painter <- Qt$QPainter()
painter$begin(pixmap)
```

[1] `QPixmap` is not to be confused with `QImage`, which is optimized for image manipulation, or the vector-based `QPicture`

```
device$render(painter)
painter$end()
```

Finally, we use the icon in a button:

```
b <- Qt$QPushButton("Histogram")
b$setIcon(Qt$QIcon(pixmap))
```

3.4 Checkboxes

The `QCheckBox` class implements a checkbox. Like the `QPushButton` class, `QCheckBox` extends `QAbstractButton`. Thus, `QCheckBox` inherits the signals `clicked`, `pressed`, and `released`. We create a check box for our demonstration:

```
checkBox <- Qt$QCheckBox("Option")
```

The `checked` property indicates whether the button is checked:

```
checkBox$checked
```

```
[1] FALSE
```

Sometimes, it is useful for a checkbox to have an indeterminate state that is neither checked nor unchecked. To enable this, set the `tristate` property to `"TRUE"`. In that case, one needs to call the `checkState` method to determine the state, as it is no longer boolean but from the `"Qt::CheckState"` enumeration.

The `stateChanged` signal is emitted whenever the checked state of the button changes:

```
qconnect(checkBox, "stateChanged", function(state) {
  if (state == Qt$Qt$Checked)
    message("checked")
})
```

```
QObject instance
```

The argument is from the `"Qt::CheckState"` enumeration; it is not a logical vector.

Groups of checkboxes

Checkboxes and other types of buttons are often naturally grouped into logical units. The frame widget, `QGroupBox`, is appropriate for visually representing this grouping. However, `QGroupBox` holds any type of widget, so it has no high-level notion of a group of buttons. The `QButtonGroup` object, which is *not* a widget, fills this gap, by formalizing the grouping of buttons behind the scenes.

To demonstrate, we will construct an interface for filtering a data set by the levels of a factor. A common design is to have each factor level correspond to a check button in a group. For our example, we take the cylinders variable from the Cars93 data set of the MASS package. First, we create our QGroupBox as the container for our buttons:

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

Next, we create the button group:

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

By default, the buttons are exclusive, like a radio button group. We disable that by setting the exclusive property to "FALSE".

We add a button for each level of the "Cylinders" variable to both the button group and the layout of the group box widget:

```
data(Cars93, package="MASS")
cyls <- levels(Cars93$Cylinders)
sapply(seq_along(cyls), function(i) {
  button <- Qt$QCheckBox(sprintf("%s Cylinders", cyls[i]))
  lyt$addWidget(button)
  bg$addButton(button, i)
})
sapply(bg$buttons(), function(i) i$checked <- TRUE)
```

Every button is initially checked.

We can retrieve a list of the buttons in the group and query their checked state:

```
checked <- sapply(bg$buttons(), function(i) i$checked)
if(any(checked)) {
  ind <- Cars93$Cylinders %in% cyls[checked]
  print(sprintf("You've selected %d cases", sum(ind)))
}
```

By attaching a callback to the buttonClicked signal, we will be informed when any of the buttons in the group are clicked:

```
qconnect(bg, "buttonClicked", function(button) {
  message(paste("Level '", button$text, "': ", button$checked, sep = ""))
})
```

3.5 Radio groups

Another type of checkable button is the radio button, QRadioButton. Radio buttons always belong to a group, and only one radio button in a group may be checked at once. Continuing our filtering example, we create several radio buttons for choosing a range for the "Weight" variable in the "Cars93" dataset:

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

The simplest way to group the radio boxes is to place them into the same layout:

```
w <- Qt$QGroupBox("Weight:")
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
sapply(l, function(i) lyt$addWidget(i))
l[[1]]$setChecked(TRUE)
```

As with any other derivative of QAbstractButton, the checked state is stored in the checked property:

```
l[[1]]$checked
```

```
[1] TRUE
```

The toggled signal is emitted twice when a button is checked or unchecked:

```
sapply(l, function(i) {
  qconnect(i, "toggled", function(checked) {
    if(checked) {
      message(sprintf("You checked %s.", i$text))
    }
  })
})
```

QButtonGroup is a useful utility for grouping radio buttons:

```
buttonGroup <- Qt$QButtonGroup()
lapply(l, buttonGroup$addButton)
```

Since our button group is exclusive, we can query for the currently checked button:

```
buttonGroup$checkedButton()
```

```
QRadioButton instance
```

3.6 Combo Boxes

A combo box allows a single selection from a drop-down list of options. In this section, we describe the basic usage of QComboBox. This includes populating the menu with a list of strings and optionally allowing arbitrary input through an associated text entry. For the more complex approach of deriving the menu from a separate data model, see Section 4.8.

This example shows how one combobox, listing regions in the U.S., updates another, which lists states in that region. First, we prepare a `data.frame` with the name, region and population of each state and split that `data.frame` by the regions:

```
df <- data.frame(name=state.name, region=state.region,
                 population=state.x77[,'Population'], stringsAsFactors=FALSE)
statesByRegion <- split(df, df$region)
```

We create our combo boxes, loading the `region` combobox with the regions:

```
state <- Qt$QComboBox()
region <- Qt$QComboBox()
region$addItems(names(statesByRegion))
```

The `addItems` accepts a character vector of options and is the most convenient way to populate a combo box with a simple list of strings. The `currentIndex` property indicates the index of the currently selected item:

```
region$currentIndex
```

```
[1] 0
```

```
region$currentIndex <- -1
```

By setting it to -1, we make the selection to be empty.

To respond to a change in the current index, we connect to the `activated` signal:

```
qconnect(region, "activated", function(ind) {
  state$clear()
  state$addItems(statesByRegion[[ind+1]]$name)
})
```

```
QObject instance
```

Our handler resets the state combo box to correspond to the selected region, indicated by `"ind"`.

Finally, we place the widgets in a form layout:

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

To allow a user to enter a value not in the menu, the property `editable` can be set to `TRUE`. This would not be sensible for our example.

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. However, underlying each is an arithmetic sequence. Our example will include both widgets and synchronize them for specifying a single range. The slider allows for quick movement across the range, while the spin box is best suited for fine adjustments.

Sliders

Sliders are implemented by `QSlider`, a subclass of `QAbstractSlider`. `QSlider` selects only from integer values. We create an instance and specify the bounds of the range:

```
sl <- Qt$QSlider()
sl$minimum <- 0
sl$maximum <- 100
```

We can also customize the step size:

```
sl$singleStep <- 1
sl$pageStep <- 5
```

Single step refers to the effect of pressing one of the arrow keys, while pressing `"Page Up/Down"` adjusts the slider by `pageStep`.

The current cursor position is give by the property `value`; we set it to the middle of the range:

```
sl$value
```

```
[1] 0
```

```
sl$value <- 50
```

A slider has several aesthetic properties. We set our slider to be oriented horizontally (vertical is the default), and place the tick marks below the slider, with a mark every 10 values:

```
sl$orientation <- Qt$Qt$Horizontal
sl$tickPosition <- Qt$QSlider$TicksBelow
sl$tickInterval <- 10
```

The `valueChanged` signal is emitted whenever the `value` property is modified. An example is given below, after the introduction of the spin box.

Spin boxes

There are several spin box classes: `QSpinBox` (for integers), `QDoubleSpinBox` and `QDateTimeEdit`. All of these derive from a common base, `QAbstract-SpinBox`. As our slider is integer-valued, we will introduce `QSpinBox` here. Configuring a `QSpinBox` proceeds much as it does for `QSlider`:

```
sp <- Qt$QSpinBox()
sp$minimum <- sl$minimum
sp$maximum <- sl$maximum
sp$singleStep <- sl$singleStep
```

There is no `"pageStep"` for a spin box. Since we are communicating a percentage, we specify "%" as the suffix for the text of the spin box:

```
sp$suffix <- "%"
```

It is also possible to set a prefix.

Both `QSlider` and `QSpinBox` emit the `valueChanged` signal whenever the value changes. We connect to the signal on both widgets to keep them synchronized:

```
f <- function(value, obj) obj$value <- value
qconnect(sp, "valueChanged", f, user.data=sl)
qconnect(sl, "valueChanged", f, user.data=sp)
```

We pass the other widget as the user data, so that state changes in one are forwarded to the other.

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:

```
le <- Qt$QLineEdit("Initial Contents")
```

The `text`roperty holds the current value:

```
le$text
```

```
[1] "Initial Contents"
```

We wish to select the text, so that the initial contents are overwritten when the user begins typing:

```
le$setSelection(start = 0, length = nchar(le$text))
```

```
NULL
```

```
le$selectedText
```

```
[1] "Initial Contents"
```

If `dragEnabled` is `TRUE` the selected text may be dragged and dropped on the appropriate targets. The `selectionChanged` signal reports selection changes.

By default, the line edit displays the typed characters. Other echo modes are available, as specified by the `echoMode` property. For example, the `Qt$QLineEdit$Password` mode will behave as a password entry, echoing only asterisks.

In Qt versions 4.7 and above, one can specify place holder text that fills the entry it is empty and unfocused. Typically, this text indicates the expected contents of the entry:

```
le$text <- ""
le$placeholderText <- "Enter some text"
```

The `editingFinished` signal is emitted when the user has committed the edit, typically by pressing the return key, and the input has been validated:

```
qconnect(le, "editingFinished", function() {
  message("Entered text: '", le$text, "'")
})
```

```
QObject instance
```

To respond to any editing, without waiting for it to be committed, connect to the `textEdited` signal.

Completion

Using the `QCompleter` framework, a list of possible words can be presented for completion when text is entered into a `QLineEdit`.

Example 3.2: Completing on `Qt` classes and methods

This example shows how completion can assist in exploring the classes and namespaces of the `Qt` library. A form layout arranges 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())
```

Next, we construct the completer for the class entry, listing the components of the `"Qt"` environment with `ls`:

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

Masks and Validation

QLineEdit has various means to restrict and validate user input. The maxLength property restricts the number of allowed characters. Beyond that, there are two mechanisms for validating input: masks and QValidator. An input mask is convenient for restricting input to a simple pattern. We could, for example, force the input to conform to the pattern of a U.S. Social Security Number:

```
le$inputMask <- "999-99-9999"
```

Please see the API documentation of QLineEdit for a full description of the format of an input mask.

As illustrated in Example 1.2, QValidator is a much more general validation mechanism, where the value in the widget is checked by the validator before being committed.

3.9 Web View Widget

The QtWebKit module provides a Qt-based implementation of the cross-platform WebKit API. The standards support is comparable to that of other WebKit implementations like Safari and Chrome. This includes HTML version 5, Javascript and SVG. The Javascript engine, provided by the QtScript module, allows bridging Javascript and R, which will not be discussed. The widget QWebView uses QtWebKit to render web pages in a GUI.

This is the basic usage:

```
webview <- Qt$QWebView()
webview$load(Qt$QUrl("http://www.r-project.org"))
```

```
NULL
```

A web browser typically provides feedback on the URL loading process. The signals `loadStarted`QWebView, `loadProgress`QWebView and `loadFinished`QWebView are provided for this purpose. History information is stored in a `QWebHistory` object, retrieved by calling `history` on the web view. This could be used for implementing a "Back" button.

3.10 Embedding R Graphics

The `qtutils` package includes a Qt-based graphics device, written by Deepayan Sarkar. We make a simple scatterplot:

```
library(qtutils)
qtDevice <- QT()
plot(mpg ~ hp, data = mtcars)
```

The `"qtDevice"` object may be shown directly or embedded within a GUI. For example, we might place it in a notebook of multiple plots:

```
notebook <- Qt$QTabWidget()
notebook$addTab(qtDevice, "Plot 1")
```

```
[1] 0
```

```
print(notebook)
```

```
QTabWidget instance
```

The device provides a context menu with actions for zooming, exporting and printing the plot. One could execute an action programmatically by extracting the action from `"qtDevice"` and activating it.

To increase performance at a slight cost of quality, we could direct the device to leverage hardware acceleration through OpenGL. This requires passing `"opengl = TRUE"` to the `QT` constructor:

```
qtOpenGLDevice <- QT(opengl = TRUE)
```

Even without the help of OpenGL, the device is faster than most other graphics devices, in particular `cairoDevice`, due to the general efficiency of Qt graphics.

Internally, the device renders to a `QGraphicsScene`. Every primitive drawn by R becomes an object in the scene. Nothing is rasterized to pixels until the scene is displayed on the screen. This presents the interesting possibility of programmatically manipulating the graphical primitives after they have been plotted; however, this is beyond our scope. See Example 3.3 for a way to render the scene to an off-screen `QPixmap` for use as an icon.

3.11 Drag and drop

Some Qt widgets, such as those for editing text, natively support basic drag and drop activities. For other situations, it is necessary to program against the low-level drag and drop API, presented here. A drag and drop event consists of several stages: the user selects the object that initiates the drag event, drags the object to a target, and finally drops the object on the target. For our example, we will enable the dragging of text from one label to another, following the Qt tutorial.

Initiating a Drag

We begin by setting up a label to be a drag target:

```
qsetClass("DragLabel", Qt$QLabel, function(text="", parent=NULL) {
  super(parent)
  setText(text)

  setAlignment(Qt$Qt$AlignCenter)
  setMinimumSize(200, 200)
})
```

When a drag and drop sequence is initiated, the source, i.e., the widget receiving the mouse press event, needs to encode chosen graphical object as mime data. This might be as an image, text or other data type. This occurs in the `mouseEventHandler` of the source:

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

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

  drag$exec()
})
```

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

We store the text in a `QMimeData` and pass it to the `QDrag` object, which represents the drag operation. The `"drag"` object is given `"this"` as its parent, so that `"drag"` is not garbage collected when the handler returns. Finally, calling the `exec` method is necessary to initiate the drag. It is also possible to call `setPixmap` to set a pixmap to represent the object as it is being dragged to its target.

Handling a Drop

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=NULL) {
  super(parent)

  setText(text)
  this$acceptDrops <- TRUE

  this$bgrole <- backgroundRole()
  this$alignment <- Qt$Qt$AlignCenter
  setMinimumSize(200, 200)
  this$autoFillBackground <- TRUE
  clear()
})
```

The important step is to allow the widget to receive drops by setting `acceptDrops` to `"TRUE"`. The other settings ensure that the label fills a minimal amount of space and draws its background. The background role is preserved so that we can restore it later after applying highlighting.

First, we define a couple of utility methods:

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

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

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

```
[1] "setText"
```

The `clear` method is used to restore the label to an initial state. The background role is remembered in the constructor, and the `setText` override saves the original text.

When the user drags an object over our target, we need to verify that the data is of an acceptable type. This is implemented by the `dragEnterEvent` handler:

```
qsetMethod("dragEnterEvent", DropLabel, function(e) {
  md <- e$mimeData()
  if (e$hasImage() || e$hasHtml() | e$hasText()) {
    super("setText", "<Drop Text Here>")
    setBackgroundRole(Qt$QPalette$Highlight)
    e$acceptProposedAction()
  }
})
```

If the data type is acceptable, we accept the event. This changes the mouse cursor, indicating that a drop is possible. A secondary role of this handler is to indicate that the target is receptive to drops; we highlight the background of the label and change the text. To undo the highlighting, we override the `dragLeaveEvent` method:

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

Finally, we have the important drop event handler. The following code implements this more generally than is needed for this example, as we only have text in our mime data:

```
qsetMethod("dropEvent", DropLabel, function(e) {
  md <- e$mimeData()

  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()
})
```

We are passed a `QDropEvent` object, which contains the `QMimeData` set on the `QDrag` by the source. The data is extracted and translated to one or more properties of the target. The final step is to accept the drop event, so that the DnD operation is completed.

4

Qt: Widgets Using Data Models

The model, view, controller (MVC) pattern is fundamental to the design of widgets that display and manipulate data. Keeping the model separate from the view allows multiple views for the same data. Generally, the model is an abstract interface. Thus, the same view and controller components are able to operate on any data source (e.g., a database) for which a model implementation exists.

Qt provides `QAbstractItemModel` as the base for all of its data models. Like `GtkTreeModel`, `QAbstractItemModel` represents tables, optionally with a hierarchy. The precise implementation depends on the subclass. Widgets that view item models extend `QAbstractItemView` and include tables, lists, trees and combo boxes. This section will outline the available model and view implementations in Qt and `qtbase`.

4.1 Display of tabular data

Displaying an R data frame

As mentioned, Qt expects data to be stored in a `QAbstractItemModel`. In R, the canonical structure for tabular data is `data.frame`. The `DataFrameModel` class bridges these structures by wrapping `data.frame` in an implementation of `QAbstractItemModel`. This essentially allows a `data.frame` object to be passed to any part of Qt that expects tabular data. It also offers significant performance benefits: there is no need to copy the data frame into a C++ data structure, which would be especially slow if the looping occurred in R. Displaying a simple table of data with `DataFrameModel` is much simpler than with GTK+ and `RGtkDataFrame`. Here we show a `data.frame` in a table view:

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

```
NULL
```

We could also pass our model to any other view expecting a QAbstractItemModel. For example, the first column could be displayed in a list or combo box.

The DataFrameModel object is a reference, so any changes are reflected in all of its views. The R data frame of a DataFrameModel may be accessed using qdataFrame:

```
head(qdataFrame(model), 3)
```

```
                mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21.0   6  160 110 3.90 2.620 16.46  0  1    4
4
Mazda RX4 Wag  21.0   6  160 110 3.90 2.875 17.02  0  1    4
4
Datsun 710     22.8   4  108  93 3.85 2.320 18.61  1  1    4
1
```

```
qdataFrame(model)$hpToMpg <- with(qdataFrame(model), hp / mpg)
```

Our table view now contains a new column, holding the horsepower to miles per gallon ratio. It is important to notice that the view has been updated implicitly, through manipulation of the underlying model.

A view has a horizontal and vertical header. The horizontal header displays the column names, while the vertical header displays the row names, if any. QHeaderView is the widget responsible for displaying headers. It has a number of parameters, such as whether the column may be moved (setMovable) and the defaultAlignment of the labels, which, as we will see later, can be overridden by the model for specific columns. By default, the labels are centered. Here, we specify left alignment for the column labels:

```
header <- view$horizontalHeader()
header$defaultAlignment <- Qt$AlignLeft
```

QTableView provides a number of aesthetic features. By default, a grid is drawn that delineates the cells. One can set showGrid to "FALSE" to disable this. If a table has more than a few columns, it may be a good idea to fill the row backgrounds with alternating colors:

```
view$alternatingRowColors <- TRUE
```

Memory management

A view keeps a reference to its model, and the model method returns the model object. However, we offer a word of caution: since multiple views can refer to a single model, a view does not own its model. This means that if a model becomes inaccessible to R, i.e., it goes out of scope, the model will be garbage collected, from lack of an owner. For example, this does not work:

```
brokenView <- Qt$QTableView()
brokenView$setModel(qdataFrameModel(mtcars))
```

```
NULL
```

```
gc()
```

```
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 321929 17.2     467875 25.0   407500 21.8
Vcells 299430  2.3    2047927 15.7  2009723 15.4
```

```
brokenView$model()
```

```
NULL
```

To prevent this, one should either (1) maintain a reference to the model in R, which we typically do in this text, or (2) explicitly give the view ownership of the model by setting the view as the parent of the model, like this:

```
parentalView <- Qt$QTableView()
brokenView$setModel(qdataFrameModel(mtcars, parent = parentalView))
```

```
NULL
```

```
gc()
```

```
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 321871 17.2     531268 28.4   407500 21.8
Vcells 299352  2.3    1638341 12.5  2009723 15.4
```

```
brokenView$model()
```

```
DataFrameModel instance
```

Formatting cells

Let us now assume that a missing value (`NA`) has been introduced into our dataset:

```
qdataFrame(model)$mpg[1] <- NA
```

The table view will display this as "nan" or "inf", which is inconsitent with the notation of R. The conversion of the numeric data to text is carried out by an *item delegate*. Similar to a GTK+ cell renderer, an item delegate is responsible for the rendering and editing of items (cells) in a view. Every type of item delegate is derived from the `QAbstractItemDelegate` class. By default, views in Qt will use an instance of `QStyledItemDelegate`, which renders items according to the current style. As Qt is unaware of the notion and encoding of missing values in R, we need to give Qt extra guidance. The `qtbase` package provides the `RTextFormattingDelegate` class for this purpose. To use it, one creates an instance and sets it as the item delegate for the view:

```
delegate <- qrTextFormattingDelegate()
view$setItemDelegate(delegate)
```

```
NULL
```

Delegates may also be assigned on a per column or per row basis. `RTextFormattingDelegate` will handle missing values in numeric vectors, as well as adhere to the numeric formatting settings in `options()`, namely `"digits"` and `"scipen"`.

Column sizing

Managing the column widths of a table view is a challenge. This section will describe some of the strategies and suggest some best practices. The appropriate strategy depends, in part, on whether the table is expanding in its container.

When the table view is expanding, it will not necessarily fill its available space. To demonstrate,

```
model <- qdataFrameModel(mtcars[,1:5])
view <- Qt$QTableView()
view$setModel(model)
wid <- Qt$QWidget()
wid$resize(1000, 500)
vbox <- Qt$QVBoxLayout()
vbox$addWidget(view)
wid$setLayout(vbox)
```

There is a gap between the last column and the right side of the window. It is difficult to appropriately size the columns of an expanding table. The simplest solution is to expand the last column:

```
header <- view$horizontalHeader()
header$stretchLastSection <- TRUE
```

To avoid the last column from being too large, we can set pixel widths on the other columns. The simplest approach is to set the `defaultSection-Size` property, which gives all of the columns the same initial size (except for the last):

```
header$defaultSectionSize <- 150
header$stretchLastSection <- TRUE
```

This usually yields an appropriate initial sizing. To resize specific columns, we could call `resizeSection`. Although specifying exact pixel sizes is inherently inflexible, the user is still free to adjust the column widths.

If, instead, one wishes to pack a table, so that it is not expanding, it may be desirable to initialize the column widths so that the columns optimally fit their contents:

```
view$resizeColumnsToContents()
```

```
NULL
```

This will need to be called each time the contents change.

By default, the size is always under control of the user (and the programmer). This depends on the resize mode. The `resizeMode` property represents the default resize mode for all columns, and it defaults to `"Interactive"`. The other modes are `"Fixed"`, `"Stretch"` (expanding), and `"ResizeToContents"` (constrained to width needed to fit contents). The `setResizeMode`ethod changes the resize mode of a specific column. Below, we make all of our columns expand:

```
header$resizeMode(Qt$QHeaderView$Stretch)
```

```
Enum value:  (0)
```

The drawback to any of these modes is that the resizing is no longer interactive; the user cannot tweak the column widths.

When the size of a column is reduced such that it can no longer naturally display its contents, special logic is necessary. By default, `QTableView` will wrap text at word boundaries. This is controlled by the `wordWrap` property. When a single word is too long, the text will be ellipsized, i.e., truncated and appended with "...". This can be disabled with

```
view$textElideMode <- Qt$Qt$ElideNone
```

When the user attempts to reduce the size of a column to the point where ellipisizing would be necessary, it may be preferable to instead reduce the widths of the other columns. This mode is enabled with

```
header$cascadingSectionResizes <- TRUE
```

4.2 Displaying Lists

It is often desirable to display a list of items, usually as text. A single column QTableView approximates this but also includes row and column headers, by default. Also, the two dimensional API of QTableView is more complicated than needed for a one dimensional list. For these and other reasons, Qt provides QListView for displaying a single column from a QAbstractItemModel as a list. We can use DataFrameModel to quickly display the first column from a data frame (or anything coercible into a data frame):

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

```
NULL
```

By default, QListView displays the first column from the model, although the column index can be customized.

Using a data model allows us to share data between multiple views. For example, we could view a data frame as a table using a QTableView and also display the row identifiers in a separate list:

```
mtcars.id <- cbind(makeAndModel = rownames(mtcars), mtcars)
model <- qdataFrameModel(mtcars.id)
tableView <- Qt$QTableView()
tableView$setModel(model)
```

```
NULL
```

```
listView <- Qt$QListView()
listView$setModel(model)
```

```
NULL
```

Now, when we resort the model, both views will be updated:

```
df <- qdataFrame(model)
qdataFrame(model) <- df[order(df$mpg),]
```

When the list items are not associated with a data frame, they may be conveniently represented as a character vector. In this case, DataFrameModel is not very appropriate, as the character vector will be coerced to a data frame. Instead, consider QStringListModel from Qt. In qtbase, QStringList refers to a character vector. We demonstrate the use of QStringListModel to populate a list view from a character vector:

```
model <- Qt$QStringListModel(rownames(mtcars))
listView <- Qt$QListView()
listView$setModel(model)
```

```
NULL
```

Now we can retrieve the values as a character vector, rather than as a data frame:

```
head(model$stringList)
```

```
1 function (...) 
2 qinvoke(<environment>, "stringList", ...)
```

QListView supports features beyond those of a simple list, including features often found in file browsers and desktops. For example, items may be wrapped into additional columns, and an icon mode, supporting unrestricted layout and drag and drop, is also available.

4.3 Accessing Item Models

We have shown how DataFrameModel and QStringListModel allow the storage and retrieval of data in familiar data structures. However, this is not true of all data models, including most of those in Qt. Alternative models are required, for example, in the case of hierarchical data. In such cases, or when interpreting user input, such as selection, it is necessary to interact with the low-level, generic API of the item/view framework.

An item model refers to its rows, columns and cells with QModelIndex objects, which are created by the model:

```
index <- model$index(0, 0)
index$row()
```

```
[1] 0
```

```
index$column()
```

```
[1] 0
```

Our "index" refers to the first row of the QStringListModel, using 0-based indices. The index points to a cell in the model, and we can retrieve the data in the cell using only the index:

```
firstCar <- index$data()
```

This can be extended to retrieve all of the items in the list:

```
sapply(seq(model$rowCount()), function(i) model$index(i - 1, 0)$data())
```

```
 [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
 [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
 [7] "Duster 360"          "Merc 240D"           "Merc 230"
[10] "Merc 280"            "Merc 280C"           "Merc 450SE"
```

```
[13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
[16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
[19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
[22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
[25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
[28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
[31] "Maserati Bora"       "Volvo 142E"
```

Setting the data is also possible, yet requires calling `setData` on the model, not the index:

```
model$setData(index, toupper(firstCar))
```

```
[1] TRUE
```

We will leave the population of a model with the low-level API as an exercise for the reader. Recall that `DataFrameModel` and `QStringListModel` provide an interface that is much faster and more convenient. When using such models, it is usually only necessary to directly manipulate a `QModelIndex` when handling user input, as we describe in the next section.

4.4 Item Selection

Selection is likely the most common type of user interaction with lists and tables. The selection state is stored in its own data model, `QItemSelectionModel`:

```
selModel <- listView$selectionModel()
```

This design allows views to synchronize selection. It also supports views on the selection state, such as a label indicating how many items are selected, independent of the particular type of item view.

There are five selection modes for item views: single, extended, contiguous, multi, and none. These values are defined by the `QAbstractItemView::SelectionMode` enumeration. `"SingleSelection"` mode allows only a single item to be selected at once. `"ExtendedSelection"` mode, the default, supports canonical multiple selection, where a range of items is selected by clicking the end points while holding the Shift key, and clicking with the Ctrl key pressed adds arbitrary items to the selection. The `"ContiguousSelection"` mode disallows the Ctrl key behavior. To allow selection on mouse-over, with range selection by clicking and dragging, choose `"MultiSelection"`. We configure our list view for single selection:

```
listView$selectionMode <- Qt$QAbstractItemView$SingleSelection
```

```
NULL
```

We can query the selection model for the selected items in our list. Let us assume that we have selected the third row. We retrieve the data (label) in that row:

```
indices <- selModel$selectedIndexes()
indices[[1]]$data()
```

```
[1] "Datsun 710"
```

When multiple selection is allowed, we must take care to interpret the selection efficiently, especially if a table has many rows. In the above, we obtained the selected indices. A selection is more formally represented by a `QItemSelection` object, which is a list of `QItemSelectionRange` objects. Under the assumption that the user has selected three separate ranges of items from the list view, we retrieve the selection from the selection model:

```
selection <- selModel$selection()
```

Next, we coerce the `QItemSelection` to an explicit list of `QItemSelection-Range` objects and generate a vector of the selected indices:

```
indicesForSelection <- function(selection) {
  selRanges <- as.list(selection)
  unlist(lapply(selRanges, function(range) seq(range$top(), range$bottom())))
}
indicesForSelection(selection)
```

```
 [1]  2  3  4  5 10 11 12 13 14 15 16 20 21 22 23 24
```

Coercion with `as.list` is possible for any class extending `QList`; `QItemS-election` is the only such class the reader is likely to encounter. Usually, the user selects a relatively small number of ranges, although the ranges may be wide. Looping over the ranges, but not the individual indices, will be significantly more efficient for large selections.

It is also possible to programmatically change the selection. For example, we may wish to select the first list item:

```
listView$setCurrentIndex(model$index(0, 0))
```

```
NULL
```

This approach is simple but only supports selecting a single item. The selection is most generally modified by calling the `select` method on the selection model:

```
selModel$select(model$index(0, 0), Qt$QItemSelectionModel$Select)
```

The second argument describes how the selection is to be changed with regard to the index. It is a flag value and thus can specify several options at once, all listed in QItemSelectionModel::SelectionFlags. In the above, we issued the "Select" command. Other commands include "Deselect" and "Toggle". Thus, we could deselect the item in similar fashion:

```
selModel$select(model$index(0, 0), Qt$QItemSelectionModel$Deselect)
```

To efficiently select a range of items, we construct a QItemSelection object and set it on the model: We have selected items 3 to 10. Multiple ranges may be added to the QItemSelection object by calling its select method.

For tabular views, selection may be row-wise, column-wise or item-wise (GTK+ supports only row-wise selection). By default, selection is by item. While this is common in spreadsheets, one usually desires row-wise selection in a table, so we will override the default:

```
tableView$selectionBehavior <- Qt$QAbstractItemView$SelectRows
```

Querying a selection is essentially the same as for the list view, except we can request indices representing entire rows or columns. In this example, we are interested in the rows, where the user has selected the third row (2):

```
selModel <- tableView$selectionModel()
sapply(selModel$selectedRows(), qinvoke, "row")
```

```
[1] 2
```

We invoke the row method on each returned QModelIndex object to get the row indices. When setting the selection, there are conveniences for selecting an entire row or column. We select the first row of the table:

```
tableView$selectRow(0)
```

Selecting a range of rows is very similar to selecting a range of list items, except we need to add the "Rows" selection flag:

```
selModel$select(sel, Qt$QItemSelectionModel$Select |
                Qt$QItemSelectionModel$Rows)
```

To respond to a change in selection, connect to the selectionChanged signal on the selection model:

```
selectedIndices <- rep(FALSE, nrow(mtcars))
selectionChangedHandler <- function(selected, deselected) {
  selectedIndices[indicesForSelection(selected)] <<- TRUE
  selectedIndices[indicesForSelection(deselected)] <<- FALSE
}
qconnect(selModel, "selectionChanged", selectionChangedHandler)
```

The change in selection is communicated as two QItemSelection objects: one for the selected items, the other for the deselected items. We update a vector of the selected indices according to the change.

4.5 Sorting and Filtering

One of the benefits of the MVC design is that models can serve as proxies for other models. Two common applications of proxy models are sorting and filtering. Decoupling the sorting and filtering from the source model avoids modifying the original data. The filtering and sorting is dynamic, in the sense that no data is actually stored in the proxy. The proxy delegates to the child model, while mapping indices between the filtered and unfiltered (or sorted and unsorted) coordinate space. Thus, there is little cost in memory.

Qt implements both sorting and filtering in a single class: `QSortFilterProxyModel`. After constructing an instance and specifying the child model, the proxy model may be handed to a view like any other model:

```
proxy <- Qt$QSortFilterProxyModel()
proxy$setSourceModel(model)
tableView$setModel(proxy)
listView$setModel(proxy)
```

Our views will now draw data through the proxy, rather than from the original model.

Both table and tree views provide an interface for the user to sort the underlying model. The user clicks on a column header to sort by the corresponding column. Clicking multiple times toggles the sort order. This behavior is enabled by setting the `sortingEnabled` property:

```
tableView$sortingEnabled <- TRUE
```

Since the sort occurs in the model, both the table view and list view display the sorted data. The sort has been applied to both the table and list view. It is also possible to sort programmatically by calling the `sort` method, passing the index of the sort column. We sort our data by the `"mpg"` variable:

```
proxy$sort(1)
```

The built-in sorting logic understands basic data types like strings and numbers. Customizing the sorting requires overriding the `lessThan` virtual method in a new class.

`QSortFilterProxyModel` supports filtering by row. The column indicated by the `filterKeyColumn` property is matched against a string pattern. Only rows with a matching value in the key column are allowed past the filter. The pattern is a `QRegExp`, which supports several different syntax forms, including: fixed strings, wildcards (globs), and regular expressions. For example, we can filter for cars made by Mercedes:

```
proxy$filterKeyColumn <- 0
proxy$filterRegExp <- Qt$QRegExp("^Merc")
```

This approach should satisfy the majority of use cases. To achieve more complex filtering, including filtering of columns, subclassing is necessary.

It is also possible to hide rows and columns at the view by calling `setColumnHidden` or `setRowHidden`. For example, we hide the `"Price"` column:

```
tableView$setColumnHidden(4, TRUE)
```

It is common for different views to display different types of information, which translates to different sets of columns. For row filtering, the proxy model approach is usually preferable to hiding view rows, as the filtering will apply to all views of the data.

4.6 Decorating Items

Thus far, we have only considered the display of plain text in item views. To move beyond this, the model needs to communicate extra rendering information to the view. With GTK+, this information is stored in extra columns, which are mapped to visual properties. Unlike GTK+, however, Qt does not require every cell in a column to have the same rendering strategy or even the same type of data. Thus, Qt stores rendering information at the item level. An item is actually a collection of data elements, each with a unique *role* identifier. The mapping of roles to visual properties depends on the `QItemDelegate` associated with the item. The default item delegate, `QStyledItemDelegate`, understands most of the standard roles listed in the `Qt::ItemDataRole` enumeration.

For example, when we create a `DataFrameModel`, the default behavior is to associate the data frame values with the `Qt$DisplayRole`. `QStyledItemDel-egate` (and its extension `RTextFormattingDelegate`) convert the value to a string for display. Other roles control aspects like the background and foreground colors, the font, and the decorative icon, if any.

`DataFrameModel` supports role-specific values for each item, `"useRoles = TRUE"` is passed to the constructor. It is then up to the programmer to indicate the mapping from a data frame column to a column and role in the model. The mapping is encoded in the column names. Each column name should have the syntax `"[.NAME][.ROLE]"`, where `"NAME"` indicates the column name in the model and `"ROLE"` refers to a value in `Qt::ItemDataRole`, without the `"Role"` suffix. If the column name does not contain a period (i.e., there is no `"ROLE"`), the display role is assumed. For example, we could shade the background of the first column, the makes and models, in gray:

```
mtcars.id <- cbind(makeAndModel = rownames(mtcars), mtcars)
model <- qdataFrameModel(mtcars.id)
qdataFrame(model)$.makeAndModel.background <- list(qcolor("gray"))
```

In the above, we store a list of QColor instances in our data frame. As a side note, if we had added that column in a call to "data.frame" or cbind, it would have been necessary to wrap the list with "I()" in order to prevent coercion of the list to a data frame.

The set of supported data types for each role depends on the delegate. For delegates derived from QStyledItemDelegate, see "qhelp(QStyledItemDelegate)". Due to implicit conversion in the internals of Qt, the number of possible inputs is much greater than those explicitly documented. For example, the "background" role demonstrated above formally accepts a QBrush object, while implicit conversion allows types such as QColor and QGradient.

It is possible for a single data frame column to specify the values for a particular role across multiple model columns. This is useful, for example, when modifying the font uniformly across several columns of interest. Here, we bold the "mpg" and "hp" columns:

```
qdataFrame(model)$.mpg.hp.font <- list(qfont(weight = Qt$QFont$Bold))
```

As shown, periods separate the data frame column names in the "NAME" component. To apply a column to all columns in the model, omit the column name:

```
qdataFrame(model)$.font <- list(qfont(pointsize = 14))
```

For models other than DataFrameModel, one sets data for a specific role by passing the optional role argument to setData. The value of role defaults to "EditRole", meaning that the data is in an editable form. We create a list view and set the background of the first item to yellow:

```
listModel <- Qt$QStringListModel(rownames(mtcars))
listModel$setData(listModel$index(0, 0), "yellow", Qt$Qt$BackgroundRole)
listView <- Qt$QListView()
listView$setModel(listModel)
```

4.7 Displaying Hierarchical Data

Hierarchical data is generally stored in QStandardItemModel, the primary implementation of QAbstractItemModel built into Qt. Hierarchical data often arises when splitting a tabular dataset by some combination of factors. For our demonstration, we will display in a tree the result of splitting the Cars93 dataset by manufacturer. The first step of our demonstration is to create the model, with a single column:

```
treeModel <- Qt$QStandardItemModel(rows = 0, columns = 1)
```

We need to create an item for each manufacturer, and store the corresponding records as its children:

```
by(Cars93, Cars93$Manufacturer, function(df) {
```

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 (`QString`)
`StatusTipRole`	Displayed in status bar (`QString`)
`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`)

```
  treeModel$insertRow(treeModel$rowCount())
  manufacturer <- treeModel$index(treeModel$rowCount()-1L, 0)
  treeModel$setData(manufacturer, df$Manufacturer[1])
  treeModel$insertRows(0, nrow(df), manufacturer)
  treeModel$insertColumn(0, manufacturer)
  for (i in seq_along(df$Model)) {
    record <- treeModel$index(i-1L, 0, manufacturer)
    treeModel$setData(record, df$Model[i])
  }
})
```

As before, we need to create a `QModelIndex` object for accessing each cell of the model. We need to add rows and columns to each manufacturer node before creating its children. This nested loop approach to populating a model is much less efficient than converting a `data.frame` to a `DataFrameModel`, but it is necessary to communicate the hierarchical information.

In addition to implementing the `QAbstractItemModel` interface, `QStandardItemModel` also represents an item as a `QStandardItem` object. Many operations, including inserting, removing and manipulating children, may be performed on a `QStandardItem`, instead of directly on the model. This may be convenient in some circumstances. For example, the code listed above for populating the model becomes:

```
by(Cars93, Cars93$Manufacturer, function(df) {
  manufacturer <- Qt$QStandardItem(as.character(df$Manufacturer[1]))
  treeModel$appendRow(manufacturer)
  children <- lapply(as.character(df$Model), Qt$QStandardItem)
  lapply(children, manufacturer$appendRow)
})
```

The QTreeView widget displays the data in a table, with the conventional buttons on the left for expanding and collapsing nodes. We create an instance and set the model:

```
treeView <- Qt$QTreeView()
treeView$setModel(treeModel)
```

Often, as in our case, a tree view only has a single column. It may be desirable to hide that column header with

```
treeView$headerHidden <- TRUE
```

Columns in a QStandardItemModel may be named by calling setHorizontalHeaderNames, as shown in the next example.

Example 4.1: 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 compare a current object with a past one. To store this information, we will use a custom "digest" role in the item model.

```
library(digest)
.DIGEST_ROLE <- Qt$Qt$UserRole + 1L
```

Using a custom role in this manner is convenient but dangerous: third-party code could attempt to store a differenet type of data using the same role ID. It is thus important to document any reserved roles.

The addItem function creates an item from a named component of a parent object and adds the new item under the given parent index:

```
addItem <- function(varname, parentObj, parentItem) {
  obj <- parentObj[[varname]]

  item <- Qt$QStandardItem(varname)
  item$setData(digest(obj), .DIGEST_ROLE)
  classItem <- Qt$QStandardItem(paste(class(obj), collapse = ", "))
  print(class(item))
  print(class(classItem))

  parentItem$appendRow(list(item, classItem))

  nms <- NULL
  if (is.recursive(obj)) {
    if (is.environment(obj))
```

```
      nms <- ls(obj)
    else if (!is.null(names(obj)))
      nms <- names(obj)
  }

  sapply(nms, addItem, parentItem = item, parentObj = obj)
}
```

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. We consider three cases: items no longer in the workspace to remove, new items to add, and finally items that may have changed and need to be replaced.

```
updateTopLevelItems <- function(view, env = .GlobalEnv) {
  envNames <- ls(envir=env)

  model <- view$model()
  items <- lapply(seq_len(model$rowCount()), model$item, column = 0)
  curNames <- as.character(sapply(items, qinvoke, "text"))



  maybeSame <- curNames %in% envNames

  curDigests <- sapply(items[maybeSame], qinvoke, "data", .DIGEST_ROLE)
  envDigests <- sapply(mget(curNames[maybeSame], env), digest)
  same <- as.character(curDigests) == as.character(envDigests)


  view$updatesEnabled <- FALSE

  remove <- !maybeSame
  remove[maybeSame] <- !same
  sapply(sort(which(remove)-1L, decreasing=TRUE), model$removeRow)

  replaceNames <- curNames[maybeSame][!same]
  newNames <- setdiff(envNames, curNames)

  sapply(c(replaceNames, newNames), addItem, parentObj = env,
         parentItem = model$invisibleRootItem())

  model$sort(0, Qt$Qt$AscendingOrder)
  view$updatesEnabled <- TRUE
}
```

First, we obtain the names and digests for each top-level row. Then, the digests are compared. Names that no longer exist in the environment or have mismatching digests are removed. We need to sort the indices in decreasing order so as not invalidate any indices. The names for the changed objects are then readded, before we add the new names. Finally, we sort the model. While the model is being modified, we freeze the view.

Finally, we construct the model and view:

```
model <- Qt$QStandardItemModel(rows = 0, columns = 2)
model$setHorizontalHeaderLabels(c("Name", "Class"))
view <- Qt$QTreeView()
view$headerHidden <- FALSE
view$setModel(model)
```

This last call initializes the display:

```
updateTopLevelItems(view)
```

4.8 Model-based combo boxes

Combo boxes were previously introduced as containers of string items and accompanying icons. The high-level API is sufficient for most use cases; however, it is beneficial to understand that a combo box displays its popup menu with a `QListView`, which is based on a `QStandardItemModel` by default. It is possible to provide a custom data model for the list view. Explicitly leveraging the MVC pattern with a combo box affords greater aesthetic control and facilitates synchronizing the items with other views.

For example, we can create a combo box that lists the same cars that are present in our table and list views:

```
comboBox <- Qt$QComboBox()
comboBox$setModel(model)
```

```
NULL
```

By default, the first column from the model is displayed; this is controlled by the `modelColumn` property.

4.9 User Editing of Data Models

Some data models, including `DataFrameModel`, `QStringListModel` and `QStandardItemModel` support modification of their data. To determine whether an item may be edited, call the `flags` method on the model, passing the index of the item, and check for the `ItemIsEditable` flag:

```
(treeModel$index(0, 0)$flags() & Qt$Qt$ItemIsEditable) > 0
```

```
[1] TRUE
```

To enable editing on a column in a `DataFrameModel`, it is necessary to specify the `edit` role for the column. For example, we might add a logical column named `Analyze` to the `mtcars` data frame for indicating whether a record should be included in an analysis. We prefix `edit` to the column name, so that the user can change its value between `TRUE` and `FALSE`:

```
df <- mtcars
df$Analyze.edit <- TRUE
model <- qdataFrameModel(df)
```

If a view is assigned an editable model, it will enter its editing mode upon a certain trigger. By default, derivatives of `QAbstractItemView` will initiate editing of an editable column upon double mouse button click or a key press. This is controlled by the `editTriggers` property, which accepts a combination of `QAbstractItemView::EditTrigger` flags. For example, we could disable editing through a view:

```
view$editTriggers <- Qt$QAbstractItemView$NoEditTriggers
```

When editing is requested, the view will pass the request to the delegate for the item. The standard item delegate, `QStyledItemDelegate`, will present an editing widget created by its instance of `QItemEditorFactory`. The default item editor factory will create a combo box for logical data, a spin box for numeric data, and a text edit box for character data. Other types of data, like times and dates, are also supported. To specify a custom editor widget for some data type, it is necessary to subclass `QItemEditorCreatorBase` and register an instance with the item editor factory.

4.10 Drag and Drop in Item Views

The item views have native support for drag and drop. All of the built-in models, as well as `DataFrameModel`, communicate data in a common format so that drag and drop works automatically between views. `DataFrameModel` also provides its data in the R serialization format, corresponding to the `"application/x-rlang-transport"` MIME type. This facilitates implementing custom drop targets for items in R.

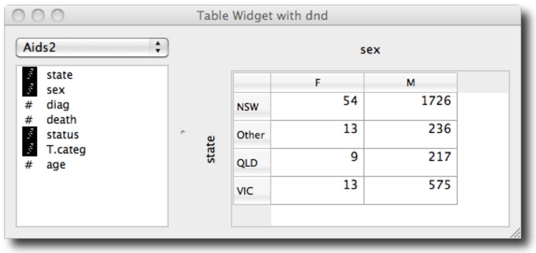
Dragging is enabled by setting the `dragEnabled` property to `"TRUE"`:

```
view$dragEnabled <- TRUE
```

Enabling drops is the same as for any other widget, with one addition:

```
view$acceptDrops <- TRUE
view$showDropIndicator <- TRUE
```

The second line tells the view to visually indicate where the item will be dropped. The following enables moving items within a view, i.e., reordering:



	F	M
NSW	54	1726
Other	13	236
QLD	9	217
VIC	13	575

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

```
view$dragDropMode <- Qt$QAbstractItemView$InternalMove
```

However, that will prevent receiving drops from other views, and dragging to other views will always be a move, not a copy.

Although we have enabled drag and drop on the view, the level of support actually depends on the model. The supported actions may be queried with `supportedDragActions` and `supportedDropActions`. The item flags determine whether an individual item may be dragged or dropped upon. Most of the built-in models will support both copy and move actions, when dragging or dropping. `DataFrameModel` only supports copy actions when dragging; dropping is not supported.

Example 4.2: A drag and drop interface to `xtabs`

This examples uses a table view to display the output from `xtabs`. To specify the variables, the user drags variable names from a list to one of two labels, representing terms in the formula.

First, we define the `VariableSelector` widget, which contains a combo box for choosing a data frame and a list view for the variable names. When a data frame is chosen in the combo box, its variables are shown in the list:

```
qsetClass("VariableSelector", Qt$QWidget, function(parent=NULL) {
  super(parent)
  ## widgets
  this$dfcb <- Qt$QComboBox()
  this$varList <- Qt$QListView()
  this$varList$setModel(qdataFrameModel(data.frame(), this))
  this$varList$dragEnabled <- TRUE

  ## layout
  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) {
    this$dataFrame <- dfcb$currentText
  })
})
```

This utility populates the combo box with a list of data frames:

```
qsetMethod("updateDataSets", VariableSelector, function() {
  curVal <- this$dfcb$currentText
  this$dfcb$clear()
  dfs <- names(which(unlist(eapply(.GlobalEnv, is.data.frame))))
  if(length(dfs)) {
    this$dfcb$addItems(dfs)
    if(is.null(curVal) || !curVal %in% dfs) {
      this$dfcb$currentIndex <- -1
      this$dataFrame <- NULL
    } else {
      this$dfcb$currentIndex <- which(curVal == dfs)
      this$dataFrame <- curVal
    }
  }
})
```

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

The data frame is stored in the following property:

```
qsetProperty("dataFrame", VariableSelector, write = function(df) {
  if (is.null(df))
    df <- data.frame()
  else if (is.character(df))
    df <- get(dfname, .GlobalEnv)
  model <- this$varList$model()
  qdataFrame(model) <- data.frame(variable = names(df),
                                  variable.decoration = I(lapply(df, getIcon)))
  this$.dataFrame <- df
  dataFrameChanged()
})
```

When the property is written, the variable list is updated and it emits this signal:

```
qsetSignal("dataFrameChanged", VariableSelector)
```

The `getIcon` generic resolves an icon from the class of a column:

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

Next, a derivative of QLabel is defined that accepts drops from the variable list and is capable of rotating text for displaying the Y component:

```
qsetClass("VariableLabel", Qt$QLabel, function(parent=NULL) {
  super(parent)
  this$rotation <- 0L
  setAcceptDrops(TRUE)
  setAlignment(Qt$Qt$AlignHCenter | Qt$Qt$AlignVCenter)
})
```

We define two properties, one for the rotation and the other for the variable name, which is not always the same as the label text:

```
qsetProperty("rotation", VariableLabel)
qsetProperty("variableName", VariableLabel)
```

To enable client code to respond to a drop, we define a signal:

```
qsetSignal("variableNameDropped", VariableLabel)
```

This utility tries to extract a variable name from the MIME data, which DataFrameModel should have serialized appropriately:

```
variableNameFromMimeData <- function(md) {
  name <- NULL
  RDA_MIME_TYPE <- "application/x-rlang-transport"
  if(md$hasFormat(RDA_MIME_TYPE)) {
    list <- unserialize(md$data(RDA_MIME_TYPE))
    if (length(list) && is.character(list[[1]]))
      name <- list[[1]]
  }
  name
}
```

To handle the drag events we override the methods dragEnterEvent, dragLeaveEvent, and dropEvent. The first two simply change the background of the label to indicate a valid drop:

```
qsetMethod("dragEnterEvent", VariableLabel, function(e) {
  md <- e$mimeData()
  if(!is.null(variableNameFromMimeData(md))) {
    setForegroundRole(Qt$QPalette$Dark)
    e$acceptProposedAction()
  }
})
qsetMethod("dragLeaveEvent", VariableLabel, function(e) {
  setForegroundRole(Qt$QPalette$WindowText)
  e$accept()
})
```

To handle the drop, we get the variable name, set the text of the label and emit the variableNameDroppedVariableLabel signal:

```
qsetMethod("dropEvent", VariableLabel, function(e) {
  setForegroundRole(Qt$QPalette$WindowText)
  md <- e$mimeData()
  if(!is.null(this$variableName <- variableNameFromMimeData(md))) {
    this$text <- variableName
    variableNameDropped()
    setBackgroundRole(Qt$QPalette$Window)
    e$acceptProposedAction()
  }
})
```

To complete the `VariableLabel` class, we override the `paintEvent` event to respect the `rotation` class. Drawing low-level graphics is beyond our scope. In short we translate the origin to the center of the label rectangle, rotate the coordinate system by the angle, then draw the text:

```
qsetMethod("paintEvent", VariableLabel, function(e) {
  p <- Qt$QPainter()
  p$begin(this)
  w <- this$width; h <- this$height
  p$save()
  p$translate(w/2, h/2)
  p$rotate(-(this$rotation))
  rect <- p$boundingRect(0, 0, 0, 0, Qt$Qt$AlignCenter, this$text)
  p$drawText(rect, Qt$Qt$AlignCenter, this$text)
  p$restore()
  p$end()
})
```

Our main widget consists of three child widgets: two drop labels for the formula 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:

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

  initWidgets()
  initLayout()
})
```

We do not list the `initLayout` method, as it simply adds the widgets to a grid layout. The `initWidgets` method initializes three widgets:

```
qsetMethod("initWidgets", XtabsWidget, function() {
  ## make Widgets
  this$xlabel <- VariableLabel()
  qconnect(xlabel, "variableNameDropped", invokeXtabs)

  this$ylabel <- VariableLabel()
  pt <- this$ylabel$font$pointSize()
  this$ylabel$minimumWidth <- 2*pt; this$ylabel$maximumWidth <- 2*pt
  this$ylabel$rotation <- 90L
  qconnect(ylabel, "variableNameDropped", invokeXtabs)

  this$tableView <- Qt$QTableView()
  this$tableView$setModel(qdataFrameModel(data.frame(), this))
  clearLabels()
})
```

The `xlabel` is straight-forward: we construct it connect to the drop signal. For the `ylabel` we also adjust the rotation and constrain the width based on the font size (otherwise the label width reflects the length of the dropped text). The `clearLabels` method, not shown, just initializes the labels.

This function builds the formula, invokes `xtabs` and updates the table view:

```
qsetMethod("invokeXtabs", XtabsWidget, function() {
  if (is.null(dataFrame))
    return()
  if(is.null(xVar <- this$xlabel$variableName)) {
    out <- NULL
  } else if(is.null(yVar <- this$ylabel$variableName)) {
    f <- formula(sprintf("~ %s", xVar))
    out <- xtabs(f, data=get(dataFrame))
  } else {
    f <- formula(sprintf("~ %s + %s", yVar, xVar))
    out <- xtabs(f, data=get(dataFrame))
  }
  if(!is.null(out))
    updateTableView(out)
})
```

We define a method to update the table view:

```
qsetMethod("updateTableView", XtabsWidget, function(table) {
  model <- this$tableView$model()
  if (length(dim(table)) == 1)
    qdataFrame(model) <- data.frame(count = unclass(table))
  else qdataFrame(model) <- data.frame(unclass(table))
})
```

Finally, we define a property for the data frame on `XtabsWidget`:

```
qsetProperty("dataFrame", XtabsWidget, write = function(df) {
  clearLabels()
  this$.dataFrame <- df
})
```

All that remains is to place the `VariableSelector` and `XtabsWidget` together in a split pane and then connect a handler that keeps the datasets synchronized:

```
w <- Qt$QSplitter()
w$setWindowTitle("GUI for xtabs()")
w$addWidget(vs <- VariableSelector())
w$addWidget(tw <- XtabsWidget())
w$setStretchFactor(1, 1)
qconnect(vs, "dataFrameChanged", function() {
  tw$dataFrame <- vs$dataFrame
})
w$show()
```

Figure 4.1 shows the result, after the user has dragged two variables onto the labels.

4.11 Widgets With Internal Models

While separating the model from the view provides substantial flexibility, in practice it is often sufficient and slightly more convenient to manipulate a view with a built-in data model. Qt provides a set of view widgets with internal models:

`QListWidget` for simple lists of items,
`QTableWidget` for a flat table and
`QTreeWidget` for a tree table.

In our experience, the convenience of these classes is not worth the loss in flexibility and other advantages of the model/view design pattern. `QTableWidget`, in particular, precludes the use of `DataFrameModel`, so `QTableWidget` is usually not nearly as convenient or performant as the model-based `QTableView`. Thus, we are inclined to omit a detailed description of these widgets. However, we will describe `QListWidget`, out of an acknowledgement that displaying a short simple list of items is a common task in a GUI.

Displaying Short, Simple Lists

`QListWidget` is an easy-to-use widget for displaying a set of items for selection. As with combo boxes, we can populate the items directly from a character vector through the `addItems` method:

```
listWidget <- Qt$QListWidget()
listWidget$addItems(state.name)
```

This saves one line of code compared to populating a `QListView` via a `QStringListModel`. To clear a list of its items, call the `clear` method. Passing an item to `takeItem` will remove that specific item from the widget.

The items in a `QListWidget` instance are of the `QListWidgetItem` class. New items can be constructed directly through the constructor:

```
item <- Qt$QListWidgetItem("Puerto Rico", listWidget)
```

The first argument is the text and the optional second argument a parent `QListWidget`. If no parent is specified, the the item may be added through the methods `addItem`, or `insertItem` for inserting to a specific instance.

To retrieve an item given its index, we call the `item` method:

```
first <- listWidget$item(0)
first$text()
```

```
[1] "Alabama"
```

Many aspects of an item may be manipulated. These roughly correspond to the built-in roles of items in `QAbstractItemModel`. One may specify the text, font, icon, status and tool tips, and foreground and background colors.

By default, `QListWidget` allows only a single item to be selected simultaneously. As with other `QAbstractItemView` derivatives, this may be adjusted to allow multiple selection through the `selectionMode` property:

```
listWidget$selectionMode <- Qt$QListWidget$ExtendedSelection
```

We can programmatically select the states that begin with "A":

```
sapply(grep("^A", state.name),
       function(i) listWidget$item(i - 1)$setSelected(TRUE))
```

The method `selectedItems` will return the selected items in a list:

```
selected <- listWidget$selectedItems()
sapply(selected, qinvoke, "text")
```

```
[1] "Alabama"  "Alaska"   "Arizona"  "Arkansas"
```

To handle changes in the selection, connect to `itemSelectionChanged`:

```
qconnect(listWidget, "itemSelectionChanged", function() {
  selected <- listWidget$selectedItems()
  selectedText <- sapply(selected, qinvoke, "text")
  message("Selected: ", paste(selectedText, collapse = ", "))
})
```

```
QObject instance
```

It is often easier for the user to select multiple items by clicking a check button next to the desired items. The check box is only shown if we explicitly set the check state of item. The possible values are "Checked", "Unchecked" or "PartiallyChecked". Here, we set all of the items to unchecked to show the check buttons and then check the selected items:

```
items <- sapply(seq(listWidget$count), function(i) {
  listWidget$item(i - 1)$setCheckState(Qt$Qt$Unchecked)
})
sapply(selected, function(x) x$setCheckState(Qt$Qt$CheckedState))
```

For long lists, this looping will be time consuming. In such cases, it is likely preferable use QListView, DataFrameModel and the "CheckedStateRole".

4.12 Implementing Custom Models

Normally, the DataFrameModel and the models in Qt are sufficient. One can imagine other cases, however. For example, one might need to view an instance of a formal reference class that conforms to a tabular or hierarchical structure. In such case, it may be appropriate to implement a custom model in R. We warn the reader that this is a significant undertaking and, unfortunately, custom models do not scale well, due to frequent callbacks into R.

Required methods The basic implementation of a model must provide the methods rowCount, columnCount, and data. The first two describe the size of the table for any views, and the third describes provides data to the view for a particular cell and role. We have already demonstrated the use of the data method in the previous sections. For example, if one is displaying numeric data, the DisplayRole might format the numeric values (showing a fixed number of digits say), yet the EditRole role might display all the digits so accuracy is not lost. If a role is not implemented, a value of NULL should be returned. One may also implement the headerData method to populate the view headers.

Editable Models For editable models, one must implement the flags method to return a flag containing ItemIsEditable and the setData method.

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 notify the views about dimension changes. For example, an implemented `insertColumns` should call `beginInsertColumns` before adding the column to the model and then `endInsertColumns` just after.

Example 4.3: Using a custom model to edit a data frame
This example shows how to create a custom model to edit a data frame. Given that `DataFrameModel` supports editing, there is no reason to actually use this model. The purpose is to illustrate the steps in model implementation. The performance is poor compared to that of `DataFrameModel`, 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, instead of overriding the `headerData` method, which the Qtviews call far too often.

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

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

Next, we define the `dataframe` property. When a new data frame is set, we call the `dataChanged` method to notify any views of a change:

```
qsetProperty("dataframe", DfModel, write = function(df) {
  this$.dataframe <- df
  dataChanged(index(0, 0), index(nrow(df), ncol(df)))
})
```

There are three virtual methods that we are required to implement: `rowCount`, `columnCount` and `data`. The first two simply access the dimensions of the data frame:

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

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, and several are defined below:

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

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

We see that numeric values are formatted to have 2 decimal points. The data is still stored in its native form; a string is returned only for display. An alternative approach would be to provide the raw data and rely on `RTextFormattingDelegate` to display the numeric values according to the current R configuration. However, the above approach generalizes basic numeric formatting.

Our `data` method has this basic structure (we avoid showing the cases for all the different roles):

```
qsetMethod("data", DfModel, function(index, role) {
  d <- this$dataframe
  row <- index$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 to 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. Since the `data` method provides a string for the edit role, `setData` will be passed one, as well. We define some methods on the S3 generic `fitIn`, which will coerce the string to the original type. For example:

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

The `setData` method is responsible for taking the value from the delegate and assigning it into the model:

```
qsetMethod("setData", DfModel, function(index, value, role) {
  if(index$isValid() && role == Qt$Qt$EditRole) {
    d <- this$dataframe
    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 our table of items to be R specific. For example, this method allows one to replace a column of values:

```
qsetMethod("setColumn", DfModel, function(col, value) {
  ## pad with NA if needed
  n <- nrow(this$dataframe)
  if(length(value) < n)
    value <- c(value, rep(NA, n - length(value)))
  value <- value[1:n]
  d <- this$dataframe
  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$dataframe
  if(name %in% names(d)) {
    return(setColumn(min(which(name == names(d))), value))
  }
  beginInsertColumns(Qt$QModelIndex(), columnCount(), columnCount())
  d[[name]] <- value
  this$dataframe <- d
  endInsertColumns()
  return(TRUE)
})
```

To demonstrate our model, we construct an instance and set it on a view:

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

Finally, we customize the view by defining the edit triggers and hiding 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)
```

4.13 Alternative Views of Data Models

Thus far, we have discussed the application of `QAbstractItemView` for viewing items in a `QAbstractItemModel`. This is the canonical model/view approach in Qt. The role of a `QAbstractItemView` is to display each item in a model, more or less simultaneously. Sometimes it is useful to view an individual item from a model in a simple widget like a label or even an editing widget, such as a line edit or spin box. For example, a GUI for entering records into a database might want to associate each of its widgets with a column in the model, one row at a time.

The `QDataWidgetMapper` class facilitates this by associating a column (or row) in a model with a property on a widget. By default, the *user* property is selected. The user property is marked as the primary user-facing property of a widget; there is only one per class. An example is the `text` property on a `QLineEdit`.

Example 4.4: Mapping selected model items to a text entry
We will demonstrate QDataWidgetMapper by displaying a table view of the "Cars93" dataset, along with a label. When a row is selected, the model name of the record will be displayed in the label. First, we establish the mapping:

```
data(Cars93, package="MASS")
model <- qdataFrameModel(Cars93, editable=TRUE)
mapper <- Qt$QDataWidgetMapper()
mapper$setModel(model)
label <- Qt$QLabel()
mapper$addMapping(label, 1)
```

The addMapping establishes a mapping between the view widget and the 0-based column index in the model.

Next, we construct a table view and establish a handler that changes the current row of the data mapper upon selection:

```
tableView <- Qt$QTableView()
tableView$setModel(model)
qconnect(tableView$selectionModel(), "currentRowChanged",
         mapper$setCurrentIndex)
```

Finally, we layout our GUI:

```
w <- Qt$QWidget()
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
lyt$addWidget(tableView)
lyt$addWidget(label)
```

Let us consider a different problem: summarizing or aggregating multiple model items, such as an entire column, and displaying the result in a widget. For example, a label might show the mean of a column, and the label would be updated as the model changed. The QDataWidgetMapper is not appropriate for this class, as it is limited to a one-to-one mapping between a model item and a widget, at any given time. The next example proposes an ad-hoc solution to this.

Example 4.5: A label that updates as a model is updated
This example shows how to create an aggregating view for a table model. We will subclass QLabel to display a mean value for a given column. Our custom view class will be a sub-class of QLabel. This 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 define a label property and call our setModel method:

```
qsetClass("MeanLabel", Qt$QLabel, function(model, column = 0, parent=NULL) {
  super(parent)
  this$model <- model
  this$column <- column
  qconnect(model, "dataChanged", function(topLeft, bottomRight) {
    if (topLeft$column() <= column && bottomRight$column() >= column)
      updateMean()
  })
  updateMean()
})
```

Whenever the data in the model changes, we need to update the display of the mean value. This private method performs the update:

```
qsetMethod("updateMean", MeanLabel, function() {
  if(is.null(model)) {
    txt <- "No model"
  } else {
    df <- qdataFrame(model)
    cname <- colnames(df)[column+1L]
    xbar <- mean(df[,cname])
    txt <- sprintf("Mean for '%s': %s", cname, xbar)
  }
  this$text <- txt
}, access="private")
```

To demonstrate the use of our custom view, we put it in a simple GUI along with an editable data frame view. When we edit the data, the text in our label is updated accordingly.

```
model <- qdataFrameModel(mtcars, editable=colnames(mtcars))
tableView <- Qt$QTableView()
tableView$setModel(model)
tableView$setEditTriggers(Qt$QAbstractItemView$DoubleClicked)
meanLabel <- MeanLabel(model)
w <- Qt$QWidget()
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
lyt$addWidget(tableView)
lyt$addWidget(meanLabel)
```

4.14 Viewing and Editing Text Documents

Multi-line text is displayed and edited by the `QTextEdit` widget, which is the view and controller for the `QTextDocument` model. `QTextEdit` supports both plain and rich text in HTML format, including images, lists and tables. Applications that display only plain text may be better served by `QPlainTextEdit`, which is faster due to a simpler layout algorithm. `QPlainTextEdit` is otherwise equivalent to `QTextEdit` in terms of API and functionality, so we will focus our discussion on QTextEdit, with little loss of generality.

Here, we create a QTextEdit and populate it with some text. Although the text is actually stored in a QTextDocument, it is usually sufficient to interact with the QTextEdit directly:

```
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"
```

The textChanged signal is emitted when the text is changed.

The text cursor To manage selections, insert special objects like tables and images, or apply the full range of formatting options, it is necessary to interact with a text cursor object, of class QTextCursor. We obtain the user-visible cursor and move it to the end of the document:

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

Manipulating the cursor object does not actually modify the location and parameters of the cursor on the screen. We need to explicitly set the modified cursor object on the QTextEdit. This behavior is often convenient, because it allows us to modify arbitrary parts of the document, without affecting the user cursor. For example, we could insert an image at the beginning:

```
cursor$setPosition(0)
cursor$insertImage(system.file("images/ok.gif", package="gWidgets"))
```

To listen to changes in the cursor position, connect to the cursorPosition-Changed signal on the QTextEdit.

Selections Selection is a component of the QTextCursor state. For plain text, the selected text is returned by the selectedText method:

```
te$textCursor()$selectedText()
```

```
NULL
```

The NULL value indicates that the user has not selected any text. The selection spans from the cursor anchor to the cursor position. Normally, the anchor and cursor are at the same position. To make a selection, we move the cursor independently of its anchor. To set the selection to include the first three words of the text, we have:

```
cursor <- Qt$QTextCursor(te$document())
cursor$movePosition(Qt$QTextCursor$Start)
cursor$movePosition(Qt$QTextCursor$WordRight, Qt$QTextCursor$KeepAnchor, 3)
te$setTextCursor(cursor)
```

```
cursor$selectedText()
```

```
[1] "ï£ijThe quick brown "
```

We move the cursor and anchor to the start of the document. Next, we move the cursor, without the anchor, across the right end of three words. Finally, we need to commit the modified cursor.

To listen to changes in the selection (according to the user visible cursor), connect to `selectionChanged`:

```
qconnect(te, "selectionChanged", function() {
  message("Selected text: '", te$textCursor()$selectedText(), "'")
})
```

The `copyAvailable` signal is largely equivalent, except it passes a boolean argument indicating whether the selection is non-empty.

Formatting By default, the widget will wrap text as entered. For use as a code editor, this is not desirable. The `lineWrapMode` takes values from the enumeration `QTextEdit::LineWrapMode` to control this:

```
te$lineWrapMode <- Qt$QTextEdit$NoWrap
```

The `setAlignment` method aligns the current paragraph (the one with the cursor) with values from `Qt::Alignment`.

Searching The `find` method will search for a given string and adjust the cursor to select the match. 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)
```

```
[1] TRUE
```

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

```
[1] "qui"
```

The second parameter to `find` takes a combination of flags from `QTextDocument::FindFlag`, with values `"FindBackward"`, `"FindCaseSensitively"` and `"FindWholeWords"`.

Context menus As we introduce Section 5, one can enable a dynamic context menu on a widget by overriding the `contextMenuEvent` virtual. For our demonstration, we aim to list candidate completions based on the currently selected text:

```
qsetClass("QTextEditWithCompletions", Qt$QTextEdit)
#
qsetMethod("contextMenuEvent", QTextEditWithCompletions, function(e) {
  m <- this$createStandardContextMenu()
  if(this$textCursor()$hasSelection()) {
    selection <- this$textCursor()$selectedText()
    comps <- utils:::matchAvailableTopics(selection)
    comps <- setdiff(comps, selection)
    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) {
          insertPlainText(i)
        })
        m$addAction(a)
      })
    }
  }
  m$exec(e$globalPos())
})
te <- QTextEditWithCompletions()
```

The `createStandardContextMenu` method returns the base context menu, including functions like copy and paste. We add an action for every possible completion. Triggering an action will paste the completion into the document.

Example 4.6: 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 with a sub-class of `QTextEdit` that specially represents a editing component in a notebook of a text editor. The constructor initializes some parameters and connects handlers to update the application actions when the state of the editor changes:

```
qsetClass("TextEditSheet", Qt$QTextEdit, function(parent=NULL) {
  super(parent)
  this$lineWrapMode <- Qt$QTextEdit$NoWrap
  setFontFamily("Courier")

  actions <- window()$actions()
  qconnect(this, "redoAvailable", function(yes) {
    if(isCurrentSheet())
      actions$redo$setEnabled(yes)
  })
  qconnect(this, "undoAvailable", function(yes) {
    if(isCurrentSheet())
      actions$undo$setEnabled(yes)
  })
  qconnect(this, "selectionChanged", function() {
    hasSelection <- this$textCursor()$hasSelection()
    if(isCurrentSheet()) {
      actions$cut$setEnabled(hasSelection)
      actions$copy$setEnabled(hasSelection)
    }
  })
  qconnect(this, "textChanged", function() {
    if(isCurrentSheet()) {
      mod <- this$document()$isModified()
      actions$save$setEnabled(mod)
    }
  })
})
```

In this simple example, the TextEditSheet will keep track of its file name. In a real design, the filename would probably be associated with the underlying data. The file name is used for saving the sheet and for labeling the notebook tab. The latter requires special logic in the setter for the property:

```
qsetProperty("filename", TextEditSheet, write = function(fname) {
  this$filename <- fname
  ## update tab label
  notebook <- this$window()$notebook()
  ind <- notebook$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 as the default destination.

```
qsetMethod("saveSheet", TextEditSheet, function(fname = filename) {
  txt <- this$toPlainText()
  writeLines(strsplit(txt, "\n")[[1]], con=fname)
})
```

The next method returns whether this editing sheet is current one in the application:

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

Next, we construct the window that contains the notebook of documents. We subclass QMainWindow (see Chapter 5), so that we can add a toolbar. Our constructor customizes the notebook, sets up the actions and toolbar, then opens with a blank sheet:

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

  notebook <- Qt$QTabWidget()
  notebook$tabsClosable <- TRUE
  notebook$usesScrollButtons <- TRUE
  notebook$documentMode <- TRUE
  qconnect(notebook, "tabCloseRequested",
           function(ind) notebook$removeTab(ind))
  qconnect(this$notebook, "currentChanged", function(ind) {
    if(ind > 0)
      updateActions()
  })
  setCentralWidget(notebook)

  initActions()
  makeToolbar()
  newSheet()
})
```

We have several actions possible in our GUI, such as the standard cut, copy and paste. We define them for the entire application, but the actions primarily work at the sheet level. The initActions constructs the actions and adds them to the window:

```
## initialize the actions
qsetMethod("initActions", TextEditWindow, function() {
  makeSheetAction <- function(name) {
    x <- Qt$QAction(name, this)
    x$setShortcuts(Qt$QKeySequence[[name]])
    qconnect(x, "triggered", function() {
      get(tolower(x$text), currentSheet())()
    })
    addAction(x)
    x
  }

  sapply(c("Redo", "Undo", "Cut", "Copy", "Paste", "Save"), makeSheetAction)

  x <- Qt$QAction("open", this)
  x$setShortcuts(Qt$QKeySequence$Open)
  qconnect(x, "triggered", function() {
    fname <- Qt$QFileDialog$getOpenFileName(this, "Select a file...", getwd())
    openSheet(fname)
  })
  addAction(x)

  x <- Qt$QAction("new", this)
  x$setShortcuts(Qt$QKeySequence$New)
  qconnect(x, "triggered", newSheet)
  addAction(x)
})
```

The enclosed makeSheetAction function creates an action that calls a method of the same name on the current sheet.

The makeToolbar method adds the actions to a toolbar:

```
## Not shown. A bit repetitive
## Could also provide a menubar as this gets *very* crowded.
qsetMethod("makeToolbar", TextEditWindow, function() {
  tb <- Qt$QToolBar()
  a <- actions()
  lapply(a, tb$addAction)
  tb$insertSeparator(a$cut)
  tb$insertSeparator(a$undo)
  this$addToolBar(tb)
})
```

Our GUI might also benefit from a menu bar, an exercise left for the reader.

This method opens a new sheet, with optional initial text:

```
qsetMethod("newSheet", TextEditWindow,
function(title="*scratch*", str="") {
  a <- TextEditSheet()         # a new sheet

  this$notebook$addTab(a, "")  # add to the notebook
  ind <- this$notebook$indexOf(a)
  this$notebook$setCurrentIndex(ind)

  a$setPlainText(str)
  a$setFilename(title)         # also updates tab
})
```

The openSheet method reads the text of a given file and displays it in a new sheet:

```
qsetMethod("openSheet", TextEditWindow, function(fname) {
  txt <- paste(readLines(fname), collapse="\n")
  newSheet(fname, txt)
})
```

The `updateActions` method is called whenever the current sheet changes, so that the state of the actions reflect that sheet:

```
qsetMethod("updateActions", TextEditWindow, function() {
  cur <- currentSheet()
  if(is.null(cur))
    return()
  a <- actions()
  a$redo$enabled <- FALSE
  a$undo$enabled <- FALSE
  a$cut$enabled <- cur$textCursor()$hasSelection()
  a$copy$enabled <- cur$textCursor()$hasSelection()
  a$paste$enabled <- cur$canPaste()
})
```

Finally, we define some accessors for getting the tabbed notebook, the currently edited sheet and the list of application actions:

```
qsetMethod("notebook", TextEditWindow, function() {
  this$centralWidget()
})
qsetMethod("currentSheet", TextEditWindow, function() {
  this$notebook()$currentWidget()
})
qsetMethod("actions", TextEditWindow, function() {
  actions <- super("actions")
  names(actions) <- sapply(actions, ‘$‘, "text")
  actions
})
```

Syntax highlighting The text edit widget supports syntax highlighting through the `QSyntaxHighlighter` class. To implement a specific highlighting rule, one must subclass `QSyntaxHighlighter` and override the `highlightBlock` method to apply highlighting. This is of somewhat special interest, so we will not give an example. For a syntax highlighting R code viewer and editor, see `qeditor` in the `qtutils` package.

5

Qt: Application Windows

Many applications have a central window that typically contains a menubar, toolbar, an application-specific area, and a statusbar at the bottom. This is known as an application window and is implemented by the `QMainWindow` widget. Although any widget in Qt might serve as a top-level window, `QMainWindow` has explicit support for a menubar, toolbar and status bar, and also provides a framework for dockable windows.

To demonstrate the `QMainWindow` framework, we will create a simple spreadsheet application. First, we construct a `QMainWindow` object:

```
mainWindow <- Qt$QMainWindow()
```

The region between the toolbar and statusbar, known as the central widget, is completely defined by the application. We wish to display a spreadsheet, i.e., an editable table:

```
data(mtcars)
model <- qdataFrameModel(mtcars, editable = TRUE)
tableView <- Qt$QTableView()
tableView$setModel(model)
mainWindow$setCentralWidget(tableView)
```

We will continue by adding a menubar and toolbar to our window. This depends on an understanding of how Qt represents actions.

Actions

The buttons in the menubar and toolbar, as well as other widgets in the GUI, might share the same action. Thus, it is sensible to separate the definition of an action from any individual control. An action is defined by 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. The properties of a `QAction` include the label `text`, `icon`, `toolTip`, `statusTip`, keyboard `shortcut` and whether the action is `enabled`.

We construct an action for opening a file:

```
openAction <- Qt$QAction("Open", mainWindow)
```

The label text is passed to the constructor. We can specify additional properties, such as the text to display in the status bar when the user moves the mouse over a widget proxying the action:

```
openAction$statusTip <- "Load a spreadsheet from a CSV file"
```

One could also set an icon from a file:

```
iconFile <- system.file("images/open.gif", package="gWidgets")
openAction$setIcon(Qt$QIcon(iconFile))
```

Actions emit a `triggered` signal when activated. The application should connect to this signal to implement the command behind the action:

```
qconnect(openAction, "triggered", function() {
  filename <- Qt$QFileDialog$getOpenFilename()
  tableView$model <- qdataFrameModel(read.csv(filename), editable=TRUE)
})
```

Toggle and radio actions An action may have a boolean state, i.e., it may be checkable. This is controlled by the `checkable` property. When a checkable action is triggered, its state is toggled and the current state is passed to the trigger handler. For example, we could have an action that toggled whether the spreadsheet will be saved on exit:

```
saveOnExitAction <- Qt$QAction("Save on exit", mainWindow)
saveOnExitAction$checkable <- TRUE
```

A checkable action in isolation behaves much like a check button. If checkable actions are placed together into a `QActionGroup`, the default behavior is such that only one is checked at once, analogous to a set of radio buttons. We could have an action for controlling the justification mode for the text entry:

```
justGroup <- Qt$QActionGroup(mainWindow)
leftAction <- Qt$QAction("Left Align", justGroup)
leftAction$checkable <- TRUE
rightAction <- Qt$QAction("Right Align", justGroup)
rightAction$checkable <- TRUE
centerAction <- Qt$QAction("Center", justGroup)
centerAction$checkable <- TRUE
```

Keyboard shortcuts Every platform has a particular convention for mapping key presses to typical actions. Qt abstracts some common commands via the `QKeySequence::StandardKey` enumeration, a member of which may refer to multiple key combinations, depending on the command and the platform. We assign the appropriate shortcuts for our “Open” action:

```
openAction$setShortcut(Qt$QKeySequence(Qt$QKeySequence$Open))
```

Whenever the window has focus and the user presses the conventional key sequence, such as `Ctrl-O` on Windows, our action will be triggered. It is important not to confuse this shortcut mechanism with mnemonics, which are often indicated by underlining a letter in the label text of a menu item. A mnemonic is active only when the parent menu is active. Mnemonics are disabled by default on Windows and Mac installations of Qt and thus are not covered here.

Menubars

Applications often support too many actions to display them all at once. The typical solution is to group the actions into a hierarchical system of menus. The menubar is the top-level entry point to the hierarchy. The placement of the menubar depends on the platform. On Mac OS X, applications share a menubar area at the top of the screen. On other platforms, the menubar is typically found at the top of the main window for the application.

We create an instance of `QMenuBar`:

```
menubar <- Qt$QMenuBar()
```

A `QMenuBar` is a container of `QMenu` objects, which represent the submenus. We create a `QMenu` for the "File" and "Edit" menus and add them to the menubar:

```
fileMenu <- Qt$QMenu("File")
menubar$addMenu(fileMenu)
editMenu <- Qt$QMenu("Edit")
menubar$addMenu(editMenu)
```

To each `QMenu` we may add:

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

We demonstrate each of these operations by populating the "File" and "Edit" menus:

```
fileMenu$addAction(openAction)
fileMenu$addSeparator()
fileMenu$addAction(saveOnExitAction)
fileMenu$addSeparator()
quitAction <- fileMenu$addAction("Quit")
justMenu <- editMenu$addMenu("Justification")
justMenu$addAction(leftAction)
justMenu$addAction(rightAction)
justMenu$addAction(centerAction)
```

In the above, we take advantage of the convenient overloads of `addAction` and `addMenu` that accept a string title and return a new `QAction` or QMenu, respectively.

Context menus

Sometimes, actions pertain to a single widget or portion of a widget, instead of the entire application. In such cases, the menubar is an inappropriate container. An alternative is to place the actions in a menu specific to their context. This is known as a context menu. The simplest approach to providing a context menu involves two steps. First, add the desired actions to the widget:

```
sortMenu <- Qt$QMenu("Sort by")
sapply(colnames(qdataFrame(model)), sortMenu$addAction)
tableView$addAction(sortMenu$menuAction())
```

Second, we configure the widget to display a menu of the actions when a context menu is requested:

```
tableView$contextMenuPolicy <- Qt$Qt$ActionsContextMenu
```

The simple approach is appropriate in most cases. One limitation, however, is that the actions need to be defined prior to the context menu request. For example, if we allowed adding and removing columns in the spreadsheet, we would need to adjust the actions in the sort context menu. Another example is a code entry widget, where a popup window could list possible code completions. Under the default context menu policy, we can implement this logic in an override of the `contextMenuEvent` virtual method:

```
showCompletionPopup <- function(e) {
  popup <- Qt$QMenu()
  comps <- utils:::matchAvailableTopics(this$text)
  comps <- head(comps, 10) # trim if large
  sapply(comps, function(i) {
    a <- popup$addAction(i)
    qconnect(a, "triggered", function(...) this$setText(i))
  })
  popup$popup(e$globalPos())
}
qsetClass("CodeEntry", Qt$QLineEdit)
qsetMethod("contextMenuEvent", CodeEntry, showCompletionPopup)
e <- CodeEntry()
```

If subclassing is undesirable, one could change the context menu policy and connect to the signal `customContextMenuRequested`:

```
e <- Qt$QLineEdit()
```

```
e$contextMenuPolicy <- Qt$Qt$CustomContextMenu
qconnect(e, "customContextMenuRequested", showCompletionPopup)
```

Toolbars

The toolbar manages a compact layout of frequently executed actions, so that the actions are readily available to the user without consuming an excessive amount of screen space. We create a `QToolBar` and add it to our main window:

```
toolbar <- Qt$QToolBar()
mainWindow$addToolBar(toolbar)
```

```
NULL
```

The main window places the toolbar into a toolbar area, which might contain multiple toolbars. It is possible, by default, for the user to rearrange the toolbars by clicking and dragging with the mouse. If the toolbar is pulled out of the toolbar area, it will become an independent window.

To add items to a toolbar we might call

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

Before adding some actions to our toolbar, we define a function `getIcon` that loads a `QIcon` from a file in the `gWidgets` package:

We create each action, set its icon, and store it in a list for ease of manipulation at a later time in the program:

```
fileActions <- list()
fileActions$open <- Qt$QAction("Open", mainWindow)
fileActions$open$setIcon(getIcon("open"))
fileActions$save <- Qt$QAction("Save", mainWindow)
fileActions$save$setIcon(getIcon("save"))
plotActions <- list()
plotActions$barplot <- Qt$QAction("Barplot", mainWindow)
plotActions$barplot$setIcon(getIcon("barplot"))
plotActions$boxplot <- Qt$QAction("Boxplot", mainWindow)
plotActions$boxplot$setIcon(getIcon("boxplot"))
```

Finally, we add the actions to the toolbar, with a separator between the file actions and plot actions:

```
sapply(fileActions, toolbar$addAction)
toolbar$addSeparator()
sapply(plotActions, toolbar$addAction)
```

QToolBar will display actions as buttons, and the precise configuration of the buttons depends on the toolbar style. For example, the buttons might display only text, only icons or both. By default, only icons are shown. We instruct our toolbar to display an icon, with the label underneath:

```
toolbar$setToolButtonStyle(Qt$Qt$ToolButtonTextUnderIcon)
```

By default, toolbars pack their items horizontally. Vertical packing is also possible; see the orientationroperty.

Statusbars

Main windows reserve an area for a statusbar at the bottom of the window. The status bar is used to display messages about the current state of the program, as well as any status tips assigned to actions.

A statusbar is an instance of the QStatusBar class. We create one and add it to our window:

```
statusbar <- Qt$QStatusBar()
mainWindow$setStatusBar(statusbar)
```

There are three types of messages in a statusbar:

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 and appears at the far right.

In addition to messages, one can embed widgets into the statusbar.

We could communicate a temporary message when a dataset is loaded:

```
statusbar$showMessage("Load complete", 1000)
```

The second argument is optional and indicates the duration of the message in milliseconds. If not specified, the message must be explicitly cleared with clearMessage.

Normal and permanent messages must be placed into a QLabel, which is then added to the statusbar like any other widget:

```
statusbar$addWidget(Qt$QLabel("Ready"))
statusbar$addPermanentWidget(Qt$QLabel("Version 1.0"))
```

Dockable widgets

QMainWindow supports window docking. There is a *dock area* for each of the four sides of the window (top, bottom, left and right). If a widget is assigned to a dock area, the user may, by default, drag the widget between the docking areas. If multiple widgets are placed into the same area, they are grouped into a tabbed notebook. Dragging a docked widget to a location outside of a dock area will convert the widget into a top-level window.

For example, we could add an R graphics device as a dockable widget. The first step is to wrap the widget in a `QDockWidget`:

```
library(qtutils)
device <- QT()
dock <- Qt$QDockWidget()
dock$setWidget(device)
```

```
NULL
```

By default, the dock widget is closable, movable and floatable. This is adjustable through the `features`roperty. For example, we could disable closing of the graphics device:

```
dock$features <- Qt$QDockWidget$DockWidgetMovable |
                 Qt$QDockWidget$DockWidgetFloatable
```

The `allowedAreas`roperty specifies the valid docking areas for a dock widget. By default, all are allowed.

After configuring the dock widget, we add it to the main window, in the left docking area:

```
mainWindow$addDockWidget(Qt$Qt$LeftDockWidgetArea, dock)
```

A second graphics device could be added with the first, on a separate page of a tabbed notebook:

```
device2 <- QT()
dock2 <- Qt$QDockWidget(device2)
mainWindow$tabifyDockWidget(dock, dock2)
```

To make `dock2` a top-level window instead, we could set the `floating`roperty to `"TRUE"`:

```
dock2$floating <- TRUE
```

Example 5.1: A main window for an IDE

This example shows how to begin constructing a main window (Figure **??**) similar to that of the web application R-Studio. (`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")
```

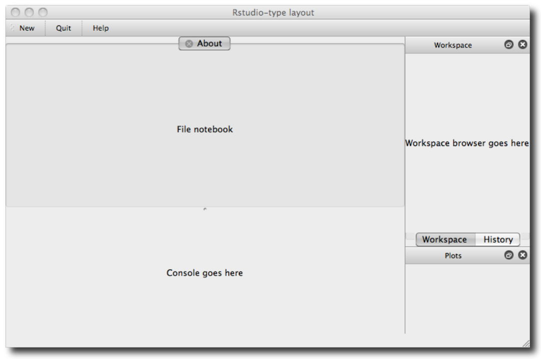


Figure 5.1: A GUI using dockable widgets and a `QMainWindow` instance

We add a menu bar and toolbar. Here only the file menu definitions are shown:

```
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 contains only some of the most important actions:

```
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. A `QSplitter` divides the space between the two main widgets:

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

As we may want to edit multiple files, we embed the editor widgets in a tabbed notebook, with closable tabs:

```
fileNotebook <- Qt$QTabWidget()
l <- Qt$QLabel("File notebook")
l$setAlignment(Qt$Qt$AlignCenter)
fileNotebook$addTab(l, "About")
fileNotebook$setTabsClosable(TRUE)
qconnect(fileNotebook, "tabCloseRequested", function(ind, nb) {
  nb$removeTab(ind)
}, user.data=fileNotebook)
centralWidget$addWidget(fileNotebook)
```

Our console widget is just a stub:

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

The right side of the layout will contain 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. 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 (whose construction is not shown):

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

We next place a plot notebook in its own space:

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

Finally, we add a status bar and show a transient message:

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