An Introduction to **qtbase**

Michael Lawrence

April 10, 2017

1 Overview

The **qtbase** package aims to interface all of Qt with R. Qt is an application framework, best known for its collection of GUI widgets, and is developed by Nokia. Please see `http://qt.nokia.com` for more information.

All methods in Qt may be invoked from R, and R may extend any Qt class. Here is a basic example of **qtbase** syntax:

```
> button <- Qt$QPushButton("Press Me!")
> qconnect(button, "pressed", function() print("Pressed"))
> button$show()

NULL
```

In the above, we create a button, register a callback that is called when the button is pressed, and show the button on the screen.

In the above example, we manipulate objects from all three types in the core **qtbase** API: libraries (*RQtLibrary*), classes (*RQtClass*), and instances (*RQtObject*). These have an approximately hierarchical relationship, and we will discuss each in the following sections.

2 Libraries

Each package that binds a library provides a *RQtLibrary* object. The **qtbase** package itself provides the `Qt` object, which binds the Qt library.

```
> Qt

Module 'qt' with 694 top-level classes
```

As evident from the above output, each library object is a container of class objects, of type *RQtClass*.

A *RQtLibrary* is a special type of `environment` and may be manipulated as any other environment:

```
> head(ls(Qt))
> Qt$QPushButton

Class 'QPushButton' with 364 public methods
```

We have just extracted the R class object for the C++ `QWidget` class, and we describe such objects in the next section.

3 Classes

A class object might represent an actual C++ class, an R derivative of a C++ class, or a C++ namespace. A class object is a special type of R `function` that serves as the constructor for the class:

```
> button <- Qt$QPushButton("Press Me!")
```

Beyond its role as a constructor, the class object is a container of static methods (or simple functions in the case of a namespace) and enumerations. We invoke the static method `tr` for translating text:

```
> Qt$QPushButton$tr("Hello World")
```

```
[1] "Hello World"
```

The above code relies on a method for `$` that is specially defined for the *RQtClass*.

4 Objects

The `button` object constructed above is a *RQtObject*. Like *RQtClass*, *RQtObject* is an `environment`. It contains methods and, for `QObject` derived instances, properties.

```
> button$show()
```

```
NULL
```

In the above, we obtain the `show` method and invoke it to show the button on the screen.

As `QPushButton` extends `QObject`, it has properties, and one of its properties is its `text` label:

```
> button$text
> button$text <- "PUSH ME!"
```

5 Connecting Signal Handlers

In any GUI, the application needs to react to user events. Qt supports this with signals. Here, we connect an R handler to a signal that is emitted when the button is `pressed`:

```
> qconnect(button, "pressed", function() print("pushed"))
```

```
QObject instance
```

The signal connection is achieved with the `qconnect` function. The R function is invoked when the `pressed` signal is emitted on `button`.

We now have a trivial but complete GUI. A widget, specifically a button, is displayed on the screen, and R code is responding to user input, a click of the button. For more examples, please see `demo(package="qtbase")`. The rest of this vignette treats advanced concepts, including the ability to extend C++ classes in R.

6 Extending C++ Classes

Many C++ libraries expect the user to extend C++ classes in normal course. For interfacing R with Qt, this presents a complication: the R user must be able to extend a Qt/C++ class.

We will demonstrate this functionality by example. Our aim is to extend the `QValidator` class to restrict the input in a text entry (`QTextEdit`) to positive numbers. The first step is to declare the class, and then methods are added individually to the class definition, in an analogous manner to the **methods** package. We start by declaring the class:

```
> qsetClass("PositiveValidator", Qt$QValidator)

Class 'R:.GlobalEnv::PositiveValidator' with 58 public methods
```

The class name is given as `PositiveValidator` and it extends the `QValidator` class in Qt. Note that only single inheritance is supported.

As a side-effect of the call to `qsetClass`, a variable named `PositiveValidator` has been assigned into the global environment (the scoping is similar to `methods::setClass`):

```
> PositiveValidator

Class 'R:.GlobalEnv::PositiveValidator' with 58 public methods
```

To define a method on our class, we call the `qsetMethod` function:

```
> validatePositive <- function(input, pos) {
+   val <- suppressWarnings(as.integer(input))
+   if (!is.na(val)) {
+     if (val > 0)
+     Qt$QValidator$Acceptable
+     else Qt$QValidator$Invalid
+   } else {
+     if (input == "")
+     Qt$QValidator$Acceptable
+     else Qt$QValidator$Invalid
+   }
+ }

> qsetMethod("validate", PositiveValidator, validatePositive)

[1] "validate"
```

The virtual method `validate` declared by `QValidator` has been overriden by the `PositiveValidator` class. The `validatePositive` function implements the override and has been defined invisibly for readability.

As an *RQtClass* object, we can create an instance by invoking `PositiveValidator` as a `function`:

```
> validator <- PositiveValidator()
```

Now that we have our validator, we can use it with a text entry:

```
> e <- Qt$QLineEdit()
> v <- PositiveValidator(e)
> e$setValidator(v)
> e$show()

NULL
```

Often, it is necessary to customize the constructor of an R class. The R function implementing the constructor must be passed during the call to `qsetClass`. Here, we extend `QMessageBox` to create a dialog, shown when the application is closing, that asks the user whether a document should be saved:

```
> qsetClass("SaveConfirmationDialog", Qt$QMessageBox,
+ function(filename = NULL, parent = NULL)
+ {
+   super(icon = Qt$QMessageBox$Question, title = "Save confirmation",
```

```
+   text = "Save the current document?",
+   buttons = Qt$QMessageBox$Cancel | Qt$QMessageBox$Discard |
+   Qt$QMessageBox$Save,
+   parent = parent)
+   this$filename <- filename
+ })

Class 'R:.GlobalEnv::SaveConfirmationDialog' with 428 public methods
```

The `super` function exists only within the scope of the constructor. It passes its arguments to the constructor of the super (parent) class. Above, we pass various parameters of the dialog to the *QMessageBox* constructor. By convention, every *QObject* derivative, including any widget, accepts its parent instance as an argument to its constructor and forwards it to the super constructor. Another special variable, `this`, corresponds to the current instance being constructed. We reference it to create an attribute for the `filename` on the instance. Similar to attributes on ordinary R objects, these attributes are dynamically typed and are implicitly defined at the instance-level by setting a value.

Within a method implementation, `super` will call a named method in the parent class. We demonstrate in our override of `accept`, which is invoked when the user decides to save the document:

```
> qsetMethod("accept", SaveConfirmationDialog, function() {
+   saveDocument(filename)
+   super("accept")
+ })

[1] "accept"
```

After saving the current document, the method calls `super` to forward the user response to one of the super classes. This is similar to `callNextMethod`, except `super` will invoke any named method, not only the current one. Also, `super` does not implicitly forward method arguments: they must be passed after the name argument.

For more examples of extending C++ classes, please see `demo(package="qtbase")`.