1

Programming GUIs using Qt

1.1 An introductory example

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

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

```
require(qtbase)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
w$show()
```

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

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

We will discuss callbacks more completely in .

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

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

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

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

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

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

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

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

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

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

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

1.2 Overview

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

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

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

1.3 The `qtbase` package

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

The package exports very few items. The main one is an environment, Qt, that contains the bulk of the functionality. The components of this environment preserve the class structure from Qt. For example, the `QWidget` class being represented through the component `Qt$QWidget`.

These components have class

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

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

These inherit from functions, as they act as constructors for instances of the class. For example

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

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

```
class(w)
```

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

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

```
ls(w)
```

```
[1] "setToolTip"      "isModal"         "deleteLater"     "addActions"
[5] "x11Info"         "setInputContext"
```

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

```
w$windowTitle
```

```
NULL
```

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

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

However, most properties in Qt are accessed through getter and setter methods. In this case, we have the setter setWindowTitle available

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

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

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

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

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

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

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

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

Constructors

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

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

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

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

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

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

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

Buttons may also have icons, for example

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

The class name prefaced with a tilde is the destructor for a widget. We can call this as follows:

```
get("~QPushButton", b)()
```

Common methods for QWidgets and QObjects

The widgets we discuss in the sequel inherit many properties and methods from the base `QObject` and `QWidget` classes. The `QObject` class is the base class and forms the basis for the object heirarchy and the event processing

system. The QWidget class is the base class for objects with a user interface. Defined in this class are several methods inherited by the widgets we discuss.

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

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

```
[1] 426   5
```

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

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

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

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

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

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

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

«ChangeGeometry, results=hid>e>= w <- Qt$QWidget()rect < -wframeGeometry rect$width()rect$setWidth(2 * rect$width())w$setGeometry(rect)

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

When a widget is resized, one can constrain how it changes by specifying a minimum size or maximum size. The properties `minimumSize`, `minimumWidth`, `minimumHeight`, `maximumSize` `maximumWidth` and `maximumHeight`, and their corresponding setters, are the germane ones. One can also adjust the `sizePolicy`roperty to fine-tune how widgets expand. For example, buttons will only grow in the x direction – not the y direction due to their default size policy.

Properties and enumerations

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

Often there can be more than two states for a property, in which case values other than logical ones must be used. For example, the label widget (among others) has a property for how its text is aligned. For alignment there are options left, right, center, top, bottom, etc. In Qt these are represented by integer values whose value can be gleaned from the appropriate manual page. However, `qtbase` provides these values as named enumerations in the `Qt$Qt` object. For example, right alignment is specified with

```
Qt$Qt$AlignRight
```

```
AlignRight
         2
attr(,"class")
[1] "QtEnum"
```

Thes values are used as follows, the last case showing how the `|` operator is used to combine alignments.

```
l <- Qt$QLabel("Our text")
l$setAlignment(Qt$Qt$AlignRight)
l$setAlignment(Qt$Qt$AlignRight | Qt$Qt$AlignVCenter)
```

A full list of the enumerated values can be teased out of the `Qt$Qt` object via

```
ls(attr(Qt$Qt,"env"))
```

```
[1] "AA_AttributeCount"                 "AA_DontCreateNativeWidgetSiblings"
[3] "AA_DontShowIconsInMenus"           "AA_ImmediateWidgetCreation"
[5] "..."
```

Fonts

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

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

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

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

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

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

Styles

Qt uses styles to provide a means to customize the look and feel of an application for the underlying operating system. Each style implements a palette of colors to indicate the states "active" (has focus), "inactive" (does not have focus), and "disabled" (not sensitive to user input). Many widgets do not have a visible distinction between active or inactive.

The role an object plays determines the type of coloring it should receive. A palette has an enumeration of `ColorRoles`. From the man page, these include, among others, the following:

```
qpaletteroles <- c(
   Window=10L,          # a general background color
   WindowText=0,        # a general foreground color
   Base=9L,             # background color for text entry widgets
   Text=6L,             # a foreground text color with 'Base'
   BrightText=7L,       # Color to contrast to "Base"
   Highlight=12L,       # indicate selected item
   HighlightedText=13L  # text color to contrast with Highlight
   )
```

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

Style Sheets

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

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

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

One can also set a background image:

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

1.4 Signals

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

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

The signal names are defined within a class or inherited through subclasses. Sometimes the callback has arguments. [1] The `clicked` signal used above inherits from `QAbstractButton`, which also is a base class for check boxes. As such, this signal passes in information if the button is checked.

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

The `qconnect` function has no return value. In `RGkt2` the return value is used to remove a callback. For Qt, the `disconnect` method can be used to

[1] One advantage of the signal-slot architecture is the type-checking of arguments.

remove connections with some degree of granularity, but this method is not implemented in `qtbase`. Rather, one can block all signals from being emitted with the `blockSignals` method, which takes a logical value to toggle whether the signals should be blocked.

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

1.5 Defining Classes and Methods

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

The `qsetClass` creates a new subclass and defines the constructor. The signature includes the arguments `x` to specify the name of the new class; `parent` to specify the class the new class will be a subclass of (for example, `Qt$QWidget` – the function, not the call, so no parentheses); `constructor` to specify the function used for construction; and a `where` argument to override where the class is defined. Of these, the constructor is the most important. In the introductory example we saw this minimal use of `qsetClass`.

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

Here we see `dateValidator` is the new class name, a sub-class of the `QValidator` class. The constructor has a single call to `super` to inherit the parent's methods.

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

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

Within the method, the special `super` function can be used to call the next method, if the sub class overrides a method. An example is in Example 1.2.

The basic call looks like `super(meth_name, arg1, arg2, ...)`.

Example 1.1: A "error label"
A common practice when validation is used for text entry is to have a "buddy label." That is an accompanying label to set an error message. As Qt uses "buddy" for something else, we call this an "error label" below. We show how to implement such a widget in `qtbase` where we subclass the single-line text edit widget constructed by `QLineEdit`. We begin by defining our subclass and constructor

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

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


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

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

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

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

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

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

```
qsetMethod("message", ErrorLabel, function() error$text)
```

```
qsetMethod("setMessage", ErrorLabel, function(msg="") {
  if(nchar(msg) > 0)
    error$setText(sprintf("<font color='red'>%s</font>", msg)) # using "red"
  setErrorHeight(nchar(msg) > 0)
  })
qsetMethod("clear", ErrorLabel, function()  setMessage())
```

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

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

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

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

1.6 Drag and drop

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

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

Initiating a drag and drop source When a drag and drop sequence is initiated, the widget receiving the mouse press event needs to set up a `QDrag`

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

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

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

Example 1.2: Drag and drop

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

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

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

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

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

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

```
})
```

Implementing a label as a drop target is a bit more work. Our basic constructor follows:

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

  setText(text)
  this$bgrole <- backgroundRole()

  setMinimumSize(200, 200)
  setAcceptDrops(TRUE)
  setAutoFillBackground(TRUE)
  clear()
})
```

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

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

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

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

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

```
qsetMethod("dragEnterEvent", DropLabel, function(e) {
  super("setText", "<Drop Text Here>")
  setBackgroundRole(qpaletteroles['Highlight'])

  e$acceptProposedAction()
})
```

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

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

```
  e$acceptProposedAction()
})
```

Finally, the important drop event. The following shows how to implement this in more generality than is needed for this example, as we only have text in our mime data.

```
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()
})
```

```
NULL
```

```
NULL
```

2

Layout managers

3

Widgets

4

Widgets using an MVC framework

5

Qt paint