Chapter 5

Take the Plunge

When creating your own widget classes, it is important to remember not to give them names in the Qt namespace, such as Qt::MyWidget. While not technically wrong, classes you create in this namespace could conflict with existing classes already in the namespace, causing erratic program behavior.

As we discussed in the last chapter, widgets are the building blocks of GUI applications. With QtRuby, we can use widgets from the toolkit and combine them into more create complex widgets, encapsulating their functionality.

5.1 Your First Custom Widget

Let's take a look at a more complicated program, in which we create our own custom widget. See if you can figure out what's going on.

File 2

```ruby
require 'Qt'
class MyWidget < Qt::Widget
  def initialize(parent=nil)
    super(parent)
    @label = Qt::Label.new(self)
    @button = Qt::PushButton.new(self)
    @layout = Qt::VBoxLayout.new(self)
    @layout.addWidget(@label)
    @layout.addWidget(@button)
    @clicked_times = 0
    @label.setText("The button has been clicked " +
        @clicked_times.to_s + " times")
    @button.setText("My Button")
  end
end
a = Qt::Application.new(ARGV)
mw = MyWidget.new
a.setMainWidget(mw)
mw.show
a.exec
```

Some of the concepts discussed before are repeated in this code. However, there's some new stuff. First, note that we create a new widget, MyWidget, from an existing widget class.

File 2

```ruby
class MyWidget < Qt::Widget
```

When creating a new GUI widget, it is important to inherit from a base QtRuby widget class such as Qt::Widget. By doing so, we gain the built in methods and properties that all widgets should have, such as a size.

Since our goal is to make a new widget that is the combination of a couple of other widgets, we base our widget off of Qt::Widget. If we wanted to extend an already existing widget, we could have based our new class directly off of it instead.

In the next part, we define the initialization code for our widget.

File 2
```
def initialize(parent=nil)
  super(parent)
  @label = Qt::Label.new(self)
  @button = Qt::PushButton.new(self)
  @layout = Qt::VBoxLayout.new(self)
```

The first thing we do in our initializer is make a call to super(). This step is very important. Calling super() explicitly runs the initializer in our inherited class (Qt::Widget in this case). Setup code defined within our base class initializer will only be executed with a call to super().

Note: Supplying the argument list to super() is optional in Ruby, as long as the superclass has the same argument list as the subclass.

We also create some child widgets in our MyWidget class. In this case, we are creating a Qt::Label, Qt::PushButton, Qt::VBoxLayout.

Okay, we fibbed a little. Some items that get used from the toolkit aren't technically widgets. In the example above, Qt::Label and Qt::PushButton are both widgets, because they inherit from the Qt::Widget class. However, items such as the Qt::VBoxLayout class don't inherit from Qt::Widget (because they don't need to).

When creating new widgets, we pass self as their parent argument. This tells each of the new widgets that their parent is the instance of the widget currently being defined.

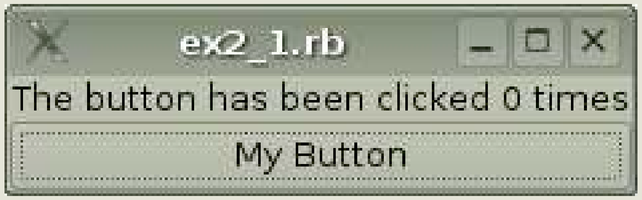
In the next section, we add our child widgets to the layout:

File 2
```
@layout.addWidget(@label)
@layout.addWidget(@button)
```

We put our widgets into the layout because we want to make use of the layout's ability to automatically resize and maintain our widgets within the program boundaries.

Finally, we put a few finishing touches on our widgets:

File 2
```
@clicked_times = 0
@label.setText("The button has been clicked " +
    @clicked_times.to_s + " times")
@button.setText("My Button")
```



The button has been clicked 0 times

Figure 5.1: Screenshot of Example 2

Both the Qt::Label and Qt::PushButton classes have setText() methods that, well, set the text displayed on the widget.

With our MyWidget widget class fully defined, we can finally create a Qt::Application to display the widget on screen.

In these examples, we could have gotten away with not creating a layout, but the widgets would not change size if we resized the application window and they may have overlapped each other. This is usually not desirable behavior.

File 2

```
a = Qt::Application.new(ARGV)
mw = MyWidget.new
a.setMainWidget(mw)
mw.show
a.exec
```

Finally, we can run the code and see our program pop up a window like that in Figure 5.1

5.2 Widget Geometry

Qt::Widget classes provide several functions used in dealing with the widget geometry. The methods width() and height() return the width and height of the widget, in pixels. The width and height values do not take into account a window frame which may surround a top level widget.

The method size(), which returns a Qt::Size object, contains the same information encapsulated inside of a Qt::Size object.

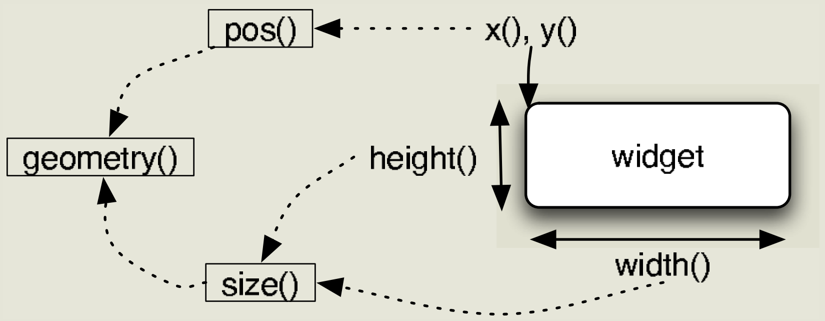


Figure 5.2: Widget Geometry

Since some methods take into account window frame geometry (for top level widgets) and other don't, we recommend reading over Qt's Window Geometry documentation. It also includes tips on how to save and restore a widget's geometry between application sessions.

Another method, geometry() returns a Qt::Rect object containing both the widget's size and relative position within its parent. The position is defined in x and y coordinates, with x being the pixel distance from the left side of the parent and y being the pixel distance from the top of the parent.

Other methods include: x(), y(), and pos() which also return the widget's relative position from within its parent. These methods, however, *do* take into account a window frame if the widget happens to be a top level widget.

Changing Geometry

It is possible to move a widget around within its parent using the methods move(int x,int y) and move(Qt::Point). You can also resize a widget using the methods resize(int x,int y) and resize(Qt::Size).

To perform both operations at the same time, use the methods setGeometry(int x,int y,int h, int w) or setGeometry(Qt::Rect).

5.3 Understanding Layouts

As we've seen, we can set the widget size and position within its parent manually. However, manual geometry management of widgets is tough. Each application is only given a select amount of screen real estate to work with and each widget in that application has to have its geometry managed. If a parent widget gets resized smaller, for example, at least one child will need to be resized as well, or some clipping of the child will occur.

Fortunately, QtRuby comes with a rich set of layout management classes which greatly simplify this task.

The class Qt::Layout is at the heart of layout management. Qt::Layout provides a very robust interface for management of widget layout. In many cases, there is no need for the complex interface provided by Qt::Layout. For the simpler cases, QtRuby provides three convenience classes based on Qt::Layout: Qt::HBoxLayout, Qt::VBoxLayout, and Qt::GridLayout.

The Qt Layout Classes guide gives some more insight into the use of these classes.

Layout classes

The BoxLayout classes handle laying out widgets in a straight line (vertically with Qt::VBoxLayout or horizontally with Qt::HBoxLayout). To utilize a BoxLayout class, simply create an instance of whichever layout is desired and use its addWidget() method to add widgets into the layout.

Alternatively, the Qt::GridLayout allows you to place widgets into a grid as shown in Figure 5.4, on the next page.

```
w = Qt::Widget.new(nil)
gl = Qt::GridLayout.new(3,4) # 3 rows by 4 columns
# put w into the first row and column
gl.addWidget(w, 0, 0)
```

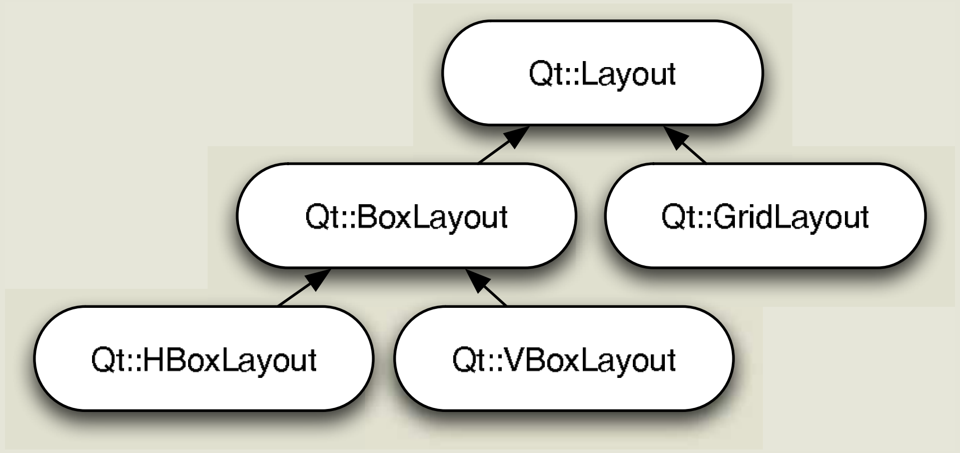


Figure 5.3: Layout class inheritance diagram

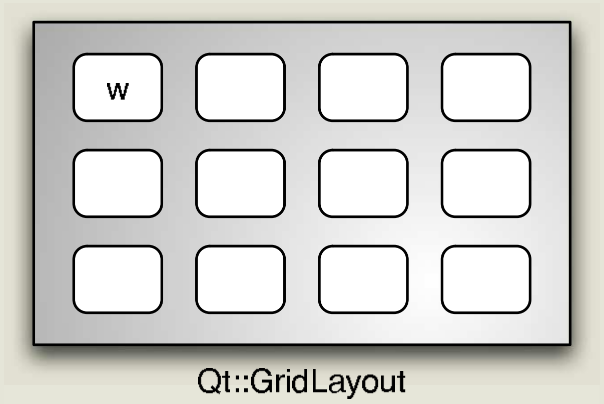


Figure 5.4: Qt::GridLayout Example

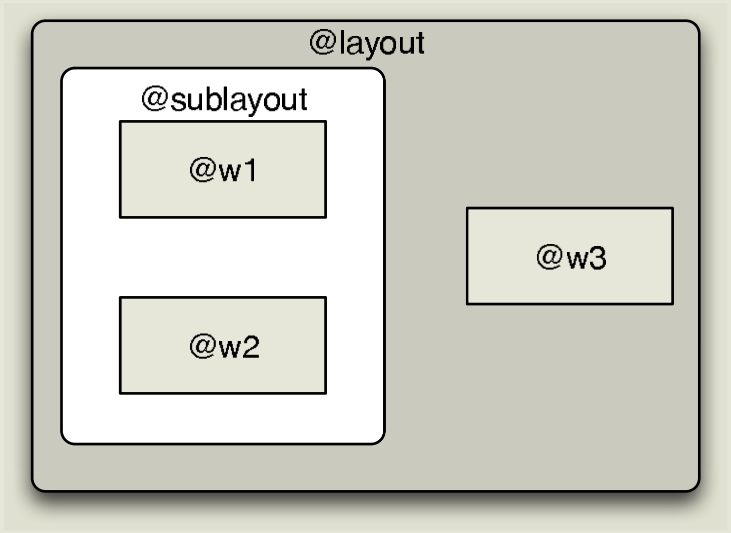


Figure 5.5: Layout and Sublayout Example

Sublayouts

Layouts can also have sublayouts contained within them. For example this code creates a sublayout as shown on Figure 5.5 .

File 10

```
@layout = Qt::HBoxLayout.new
@sublayout = Qt::VBoxLayout.new
@w1 = QWidget.new
@w2 = QWidget.new
@w3 = QWidget.new
@sublayout.addWidget(w1)
@sublayout.addWidget(w2)
@layout.addLayout(@sublayout)
@layout.addWidget(@w3)
```

In Figure 5.6, on the next page we demonstrate why sublayouts are convenient. On the left side we created a Qt::VBoxLayout containing three Qt::CheckBoxes. Then we nested this layout inside of a Qt::HBoxLayout and also put in a Qt::Dial. As you can see, the sublayout allows us to group related items together in a logical way and

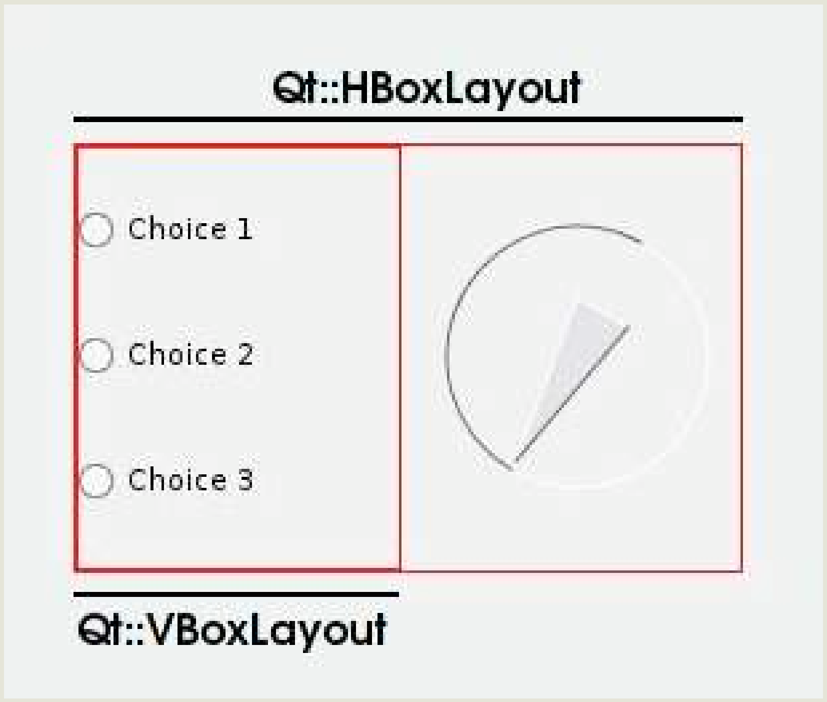


Figure 5.6: A Layout with a Nested Sublayout

maintain the size and spacing policies we desire.

Layout properties

All layouts have two fundamental properties, margin and spacing. These are shown on Figure 5.7, on the following page. *Spacing* represents the pixel space between each of the items within the layout. *Margin* represents an outer ring of pixel space surrounding the layout. Both are settable properties using the setMargin() and setSpacing() methods.

In lieu of adding a widget or a sublayout into a Qt::Layout, there are

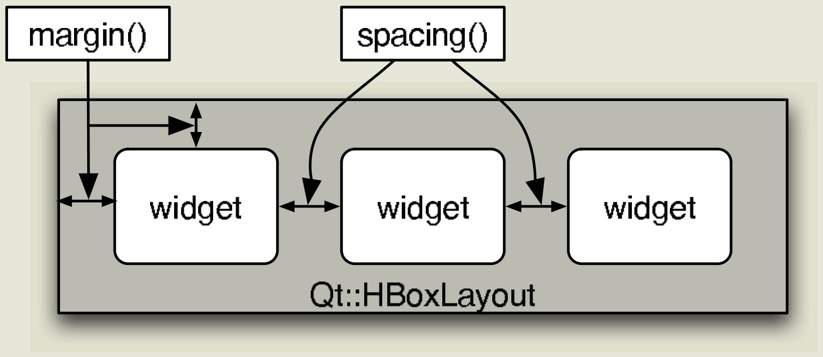


Figure 5.7: Layout Margin and Spacing

some other interesting additions. addSpacing() allows you to add a fixed amount of space directly in the widget. addStretch() adds a stetchable space in the widget.

Sizing up the situation

Layouts only define the placement of objects, not the space that they are allotted. From an outside perspective it may seem as though all of the widgets should take up a proportionate amount of space based on how many other widgets are in the layout. This layout style, though, is not always ideal.

We highly recommend using the layout classes over manual manipulation of widget geometry.

Enter Qt::SizePolicy. This class, which is also a settable property of Qt::Widget (using the setSizePolicy() method), contains the information a widget uses to determine the amount of space it will take up inside a layout. When coupled with all of the other widgets in the layout, the SizePolicies are all calculated and a final overall layout is achieved.

Each size policy utilizes a calculated geometry called a sizeHint(). The

The sizeHint() method returns a Qt::Size object, which is nothing more than an encapsulated set of width and height properties.

sizeHint()is a method built into Qt::Widget which calculates the recommended size of the widget. A sizeHint() is calculated based on the design of the widget. For example, a Qt::Label's sizeHint() is calculated based on the text that is written on the label. This is to help ensure that all of the text always fits on the Qt::Label.

```
irb(main):001:0> require 'Qt'
=> true
irb(main):002:0> app = Qt::Application.new(ARGV)
=> #<Qt::Application:0xb6adfb24 name="irb">
irb(main):003:0> Qt::Label.new("Blah",nil).sizeHint
=> #<Qt::Size:0xb6adc44c width=30, height=17>
irb(main):004:0> Qt::Label.new("BlahBlahBlahBlahBlah",nil).sizeHint
=> #<Qt::Size:0xb6ad86bc width=142, height=17>
```

The above shows that a sizeHint() for a Qt::Label is dependent on the text being displayed on the label.

There are seven types of size policies:

Qt::SizePolicy	Size		
	Minimum	Maximum	Preferred
Fixed	sizeHint()	sizeHint()	sizeHint()
Minimum	sizeHint()	none	sizeHint()
Maximum	none	sizeHint()	sizeHint()
Preferred	none	none	sizeHint()
Minimum-Expanding	sizeHint()	none	all available space
Expanding	none	none	sizeHint(), will expand as necessary
Ignored	none	none	all available space

These Qt::SizePolicy types are set independently for both the horizontal and vertical directions.