Rapid GUI Development with QtRuby

Caleb Tennis
Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking g device are trademarks of The Pragmatic Programmers, LLC. Qt® is a registered trademark of Trolltech in Norway, the United States and other countries.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

To see what we’re up to, please visit us at

http://www.pragmaticprogrammer.com

Copyright © 2006 The Pragmatic Programmers LLC.

All rights reserved.

This PDF publication is intended for the personal use of the individual whose name appears at the bottom of each page. This publication may not be disseminated to others by any means without the prior consent of the publisher. In particular, the publication must not be made available on the Internet (via a web server, file sharing network, or any other means).

Produced in the United States of America.

Lovingly created by gerbil #40 on 2006-11-11
Contents

1 Introduction
   1.1 Frameworks ......................... 1
   1.2 Our Assumptions ................... 2
   1.3 Acknowledgements ................. 3

2 About Qt
   2.1 A Little History .................. 5
   2.2 Versions ........................... 6
   2.3 Where to get Qt ................... 7
   2.4 How to install Qt from source .... 8
   2.5 Installation Issues ............. 10
   2.6 Exploring the toolkit .......... 11

3 About QtRuby
   3.1 Language Bindings ............... 13
   3.2 I smell SMOKE .................... 14
   3.3 Installing QtRuby ............... 14
   3.4 Installation Issues ............ 18

4 Get Your Feet Wet
   4.1 Your first program ............... 19
   4.2 Objects and Widgets and Parents, oh my! .......... 21
   4.3 The Qt Object Model ............. 26
   4.4 Other initialization items ....... 30
   4.5 The Qt::Application class ....... 31
   4.6 Summary ......................... 33
# Contents

5 **Take the Plunge**
- 5.1 Your First Custom Widget ........................................... 34
- 5.2 Widget Geometry .................................................. 36
- 5.3 Understanding Layouts ............................................. 38
- 5.4 Automating a task .................................................. 45
- 5.5 Signals and Slots .................................................... 47
- 5.6 Slot Senders ........................................................... 57
- 5.7 Summary .............................................................. 58

6 **Sink or Swim**
- 6.1 Event Methods ...................................................... 59
- 6.2 Event Filters .......................................................... 63
- 6.3 The Main Event ....................................................... 65
- 6.4 The Event Loop ........................................................ 66
- 6.5 Event posting ........................................................... 68
- 6.6 Summary .............................................................. 69

7 **Home Stretch**
- 7.1 Qt Modules ........................................................... 70
- 7.2 QtRuby tools ............................................................ 72
- 7.3 Taking Advantage of Ruby .......................................... 75
- 7.4 Disposing of Widgets ................................................. 76
- 7.5 Debugging a QtRuby Application ................................. 77

8 **Korundum**
- 8.1 Installing Korundum .................................................. 80
- 8.2 Using Korundum ....................................................... 81
- 8.3 DCOP—Interprocess Communication ........................... 82
- 8.4 Summary .............................................................. 87
A  Event Method Map  88

B  Resources  89
   B.1  Web Resources  89
   B.2  Bibliography  90
1.1 Frameworks

Creating a graphical application with a scripting language isn’t new. TCL, a popular scripting language of the early 1990s has Tk, a graphical extension using the Motif libraries. For years, these toolkits were the defacto standard for creating GUI applications both easily and quickly.

It’s probably no surprise that Ruby comes with libraries that support TCL and Tk.

But, as time moves on, tools come onto the scene that provide new features that users want. The GUI framework Qt is one such tool, built and refined over many years of use. Today, Qt is a powerhouse framework, providing a top notch interface for building applications on all three major computing platforms.

Qt and Ruby—A Lovely Marriage

We believe that Qt provides the perfect mix of features for creating robust GUI applications. We also believe that extending the use of Qt into the Ruby domain gives us incredible power to create high quality applications.

The choice of which toolkit to use is a personal one. For some developers, there is as much passion in the choice of toolkit as there is in their choice of Ruby, Perl, or Python. When starting out with a new framework like QtRuby, we recommend that you investigate all the possible competing options before making any decisions.
Other GUI/Ruby framework combinations are:

- FXRuby (for the FOX toolkit)
- wxRuby (for wxWidgets)
- Ruby/Gnome2 (for GTK)
- RubyCocoa (for Cocoa)

1.2 Our Assumptions

In this book, we assume that you have some familiarity with Ruby—that you understand and read Ruby code and can follow examples in the book. If not, pick up a copy of Programming Ruby [TFH05].

We do not assume you have familiarity with Qt, although, a moderate amount of familiarity will be a plus. For this, we recommend C++ GUI Programming with Qt 3 [BS04], which is also freely available on the web (see Appendix B, on page 89).

We also assume you’re comfortable with your platform—Linux or Mac—and that you are able to follow some of the instructions on installing the software. We’ve attempted to make it as easy as possible, but some troubleshooting on your part may be required if something doesn’t work right.

Last, we assume that you will follow through the examples as they are presented. Unfortunately, we don’t have the space or time to discuss every aspect of the toolkit. However, after learning the fundamentals presented within, we feel confident that you will have enough understanding of QtRuby to feel comfortable learning more on your own.
1.3 Acknowledgements

First, thanks to the developers who were responsible for QtRuby and SMOKE: Richard Dale, Ashley Winters, Germain Garand, David Faure, and others.

Thanks to the developers at Trolltech who produce Qt and provide the GPL version to the open source community.

Thanks to the two Pragmatists, Andy and Dave, who provided input, editing, and suggestions on the book.

Thanks to the Ruby community for being helpful and friendly to new comers who tend to ask the same questions over and over again.

Finally, thanks to my wife, Anna, who put up with many evenings of her husband paying more attention to this book than to her.
If you are already familiar with Qt and installing it on your system, you can skip ahead to Chapter 3, About QtRuby, on page 13.

Qt, by Trolltech, is a cross-platform GUI toolkit, written in C++. Some of the main selling points of Qt are:

• **Cross Platform**—Qt is available for Windows, Mac, and Unix. Qt follows the mantra: write once, compile anywhere. You literally only have to write one program that, after being compiled, will run on any supported platform.

• **Modular**—The toolkit comes with many modular, extensible components, such as the SQL, threading, and networking modules. While not all of these extra components are directly GUI related, they are very helpful for adding functionality within GUI programs while maintaining the cross platform nature of the toolkit.

• **Open Source**—Qt is licensed under the GPL. The source code is fully available and completely free. Trolltech benefits by having a large user base which can report feedback and provide source code patches for bugs found in the toolkit.

• **Binary Compatibility**—When a new version of the Qt toolkit is released, it won’t alter the way your existing programs function. You can drop the latest version of Qt in place and benefit from bug fixes and feature additions without worry that something in your program will stop working properly.
2.1 A Little History

Qt was born in 1991 as a product to aid in GUI application development. In 1996, student Matthias Ettrich began using Qt as a basis for the KDE project—an opensource Unix desktop environment. By 1997, the popularity of KDE and Qt was growing, but concerns about Qt licensing issues were also starting to develop by members of the open source community.

Some people involved within the open source community worried about the direction of the Unix desktop. Qt was the Unix desktop’s main toolkit, so having it controlled by a commercial entity worried many people. In 1998, the GNOME project was started to create an alternative desktop that would be more compatible with the goals of open source software.

Licensing

In response to the community’s moves, Trolltech licensed Qt under the QPL, an open source license. However, the Free Software Foundation, the figurehead of the open source movement, did not regard the QPL as being compatible with the GPL, its standard open source license of the time.

In 2000, Qt 2.2 was released under a dual QPL/GPL license which allowed the author using the toolkit to decide which of the licenses they wanted their application to fall under. With Qt 2.2, a fully GPL compatible Qt was available for Unix. Since then, Trolltech has also released versions of Qt under the GPL for Mac, starting with 3.1.2, and for Windows, starting with 4.0.0.

Current releases of Qt are licensed under a dual commercial/GPL license structure. This means that Qt is freely available, with full
source code, on each of the three major platforms under the GPL. This also means that any software written using the Qt libraries must abide by the GPL.

A non-GPL commercial option also exists, allowing customers to purchase a license from Trolltech so that software written using the toolkit can be licensed by the author. However, the source code between the GPL and commercial versions of Qt is the same.\textsuperscript{1}

**Philosophy**

It’s not our intention to discuss the philosophy of software licensing in this book. Instead, we want to make you aware of the licensing options of Qt and let you decide what works best for you. Before beginning any QtRuby project, we encourage you to research your Qt license (GPL, QPL, or commercial) and what implications it may have on your project.

### 2.2 Versions

Qt is classified by its major version number, with 4 being the most recently released. Below the major version is a minor version number, like 4.0, and a patch level number, such as 4.0.0. Patch level releases are done periodically to fix bugs that have been found in the code. 4.0.1 represents the next patch level release from 4.0.0. Minor version releases are done less frequently and usually involve larger additions to the library, such as the addition of a new class.

\textsuperscript{1}There are some subtle differences, but for the most part there is no difference in the commercial and GPL versions.
Qt releases with the same major version number are binary compatible. This means that you can drop in a later released version of the toolkit, and your programs will continue to function as before without the need to recompile them. You also may be able to drop in an earlier version of the same major release without problems, as long as your program doesn’t make use of any functionality added between the two releases.

The 4.0 version of Qt was released on June 28, 2005. Because of big changes between the previously released 3.3 series many existing applications are still using the older, more mature version 3 of the library. Because of this, we will focus solely on the major version 3 series of Qt. We hope to provide an updated revision of the book once Qt 4 becomes more widely adopted and the QtRuby bindings fully support it.

2.3 Where to get Qt

Qt can be installed using your system’s package management tools, or you can download and build it from the source (see Section 2.4, *How to install Qt from source*, on the following page).

**Linux/Unix**

Many Linux distributions come with Qt. Check your package management system to see what version of Qt is installed or available to be installed. You will also need any development packages (such as qt-devel) to be installed as well. On Linux, Qt relies on the X11 window management system, which must be installed and configured.
Mac OS X

Mac OS X users who use an automated installation manager such as Fink should install Qt from there. Fink users will most likely want to install all qt3 packages.

2.4 How to install Qt from source

If you’re an open source junkie like us, you probably want to install Qt from the source code. It’s actually pretty easy.

The first step is to download the source from the Trolltech web site and unpack the file.

Next, you need to use the configure program to set up the build. There are many configuration items, which are viewable using the -help option to configure:

```
user@localhost ~/qt $ ./configure -help
```

Usage: configure [-prefix dir] [-buildkey key] [-docdir dir]
[-headerdir dir] [-libdir dir] [-bindir dir] [-plugindir dir]
[-datadir dir] [-translationdir dir] [-sysconfdir dir] [-debug]
[disable-module] [with-module setting]
[without-module setting] [-fast] [-no-fast]

When installing Qt manually, you must already have a window system installed. Windows and Mac users should be fine, but Linux/Unix users will need the X11 window system.
Many of the options are documented below this output, but as you can see there are a lot of options to choose from. Luckily, Qt attempts to be smart about which options it chooses and in many cases will attempt to auto detect if certain options can be used or not.

We recommend that you use the defaults unless you explicitly know which settings are of value to you.

```
user@localhost ~/qt $ ./configure
```

This is the Qt/X11 Open Source Edition.

You are licensed to use this software under the terms of either the Q Public License (QPL) or the GNU General Public License (GPL).

Type 'Q' to view the Q Public License.
Type 'G' to view the GNU General Public License.
Type 'yes' to accept this license offer.
Type 'no' to decline this license offer.

Do you accept the terms of either license?

At this point you should make sure you understand any implications which may be involved with the presented licenses. Type yes to continue.

Eventually, Qt will stop configuring and should tell you that it’s time to start compiling. Typing make will start the compilation process.

```
user@localhost ~/qt $ make
```

The most popular compiler for building Qt is GCC, but C++ compilers built by other vendors are supported as well.

```
Qt is now configured for building. Just run /usr/bin/gmake.
To reconfigure, run /usr/bin/gmake confclean and configure.
```

```
user@localhost ~/qt $ make
```
Finally, when it’s done, Qt needs to be installed. If during configure time you specified an installation prefix (using the `-prefix` switch), then you can run `make install` now to install Qt to that location.

Alternatively, you can simply leave everything as is and move the entire Qt directory to the desired installation location.

The installation location is a bit of a personal preference. For Linux, we like to put Qt in `/usr/local/qt`. For OS X, we recommend `/Developer/Qt`.

```
user@localhost ~/qt $ cd ..
user@localhost ~ $ sudo mv qt /usr/local/qt
user@localhost ~ $
```

It’s important to make sure that after Qt is installed the libraries in the `lib` subdirectory are added to the dynamic linker search path. In Linux, this can be done in the `/etc/ld.so.conf` file or by using the `LD_LIBRARY_PATH` environment variable. With Mac OS, using the `DYLD_LIBRARY_PATH` will work as well.

It’s also important that the programs in the `bin` subdirectory are added to the executable path. This is done with the `PATH` environment variable.

We recommend you take a look at the `INSTALL` file in the Qt package. It has some important information in it about these installation matters.

### 2.5 Installation Issues

Many things can go wrong when installing a new software package, especially one that is as encompassing as Qt. It’s impossible to plan...
for every eventuality, but here are some pointers to help you try and troubleshoot what may have happened.

- Identify what phase you were in when the trouble happened. Did you get Qt unarchived? Did the problem occur after you typed make? What kind of output did you see?

- Make sure you didn’t make a spelling mistake when running the commands. Misspelling an option passed to configure could cause a problem. Likewise, running mkae instead of make is problematic. Be aware of any spurious error messages that could have come up.

- Did you specify an option to configure that isn’t valid for your system? Perhaps you need to specify a different value. Some options, like -thread, may be required if some of your underlying libraries that Qt links against (X11, for example) are also built as threaded.

- Did the build process die during the make stage? Try to find the error message that caused the failure. Sometimes the build process dies because different library versions are found during compile time than were found during configure time. Other times, the compiler itself has a bug and may cause a segmentation fault. We’ve also seen the compiler die simply because the computer was overheating during the compile process.

### 2.6 Exploring the toolkit

The Qt library is in the lib directory under your Qt installation prefix. If Qt was built non-threaded, the library name is something like libqt.so.3.3.5. If it is built threaded, the library name is libqt-mt.so.3.3.5.
Additionally, there’ll be soft links that point to these libraries with various version formats.

Executables that come with Qt are in the bin directory in the installation prefix. Some of the commonly used applications are:

- **assistant**—A tool for searching the Qt API reference files and program documentation
- **designer**—A graphical program for designing GUI interfaces
- **linguist**—A GUI tool for handling internationalization of Qt applications
- **moc**—A C++ code parser that translates special Qt specific extensions into C++ code.
- **qmake**—A Makefile generator based on project (.pro) files.
- **uic**—A tool for converting .ui files created by designer into C++ files.

Qt also comes with many examples and plugins (in directories named examples and plugins) to make programming as painless as possible. Because it’s impossible to explore all of the features Qt has to offer in this book, we recommend you spend some time looking at the demos, examples, and other available files to see all of the possibilities that exist for writing your application.
3.1 Language Bindings

Language bindings for libraries are very popular. They give freedom to you, the author, to use highly desired features within that library with the flexibility of the programming language of your choice. In fact, one reason we like Ruby so much is its intrinsic capability to extend the language from C libraries.

Creating a language binding to C++ libraries, like Qt, is difficult. Really difficult. Trust us, you’re better off not knowing the specifics. There are a couple of general reasons that make it so hard.

- **Name mangling**—In C, every function has a clear and concise name which gets tucked away inside of the library. In C++, objects can have multiple functions with the same name. Virtual methods also have the same name as their ancestors. To work around this, in a C++ library names are *mangled* so that every function gets a unique name within the library. Furthermore, different compilers may implement name mangling in different ways.

- **Templates**—The C++ feature of templates, while very powerful, are also very tricky to implement in a compiler. Different compilers offer varying levels of support for template implementation. Encapsulating that functionality within a wraparound language proves to be a difficult operation.

- **Qt extensions**—Qt C++ code is extended by the *MOC*, which creates some extra C++ code that automatically gets compiled
in with your application. Because of its additional MOC code, a one-size-fits-all C++ wrapper would have a hard time working with Qt.

Some tools make the process easier. The most popular C++ binding generator, SWIG, provides a powerful set of tools for creating bindings in a number of languages. Unfortunately, some of the specialized aspects of Qt listed above are beyond what SWIG is capable of handling autonomously. Luckily, there is a tool that can help us.

### 3.2 I smell SMOKE

The SMOKE utility creates a *wrapper* library for all of the Qt (and optionally KDE) method calls. **SMOKE** works by parsing the header files of the installed Qt libraries, and generates code which can call each of the methods in the Qt classes. In other words, SMOKE is the go-between between Ruby and Qt.

Other Qt language bindings, including PerlQt, use SMOKE as well.

### 3.3 Installing QtRuby

Some Linux distributions, such as Debian and Gentoo, come with the ability to automatically install QtRuby using their respective repository installation programs. For example, on Gentoo Linux, installing QtRuby is as easy as typing `emerge qtruby`. In other distributions, QtRuby may be distributed under the `kdebindings` package.

For manual installation, the current version of QtRuby is available at [http://rubyforge.org/projects/korundum](http://rubyforge.org/projects/korundum). We recommend downloading the most recent version (1.0.11 as of this writing).
You should download the Korundum package if you wish to use the KDE library bindings. The Korundum package contains QtRuby, so there’s no need to download both. See Chapter 8, Korundum, on page 80 for more information on how to use Korundum.

After unpacking the file, the QtRuby needs to be configured. To show a list of configurable options, use:

`~/qtruby> ./configure --help`

### Building on Linux

On Linux, we configured QtRuby like this:

`~/qtruby> ./configure --with-smoke="qt" \
   --with-qt-dir=/usr/qt/3 --prefix=/usr`

Your configuration items may vary based on where Qt is installed (`--with-qt-dir`) and where SMOKE will be installed (`--prefix`). You may also have additional options.

If all goes well, you should see a positive note to that effect and a prompt to begin the build process:

```
Good - your configure finished. Start make now
~/qtruby> make
```

The build process starts by first generating the SMOKE bindings for Qt. After this, the QtRuby specific code is compiled. When complete, you simply need to install the files.

`~/qtruby> sudo make install`

### Building on Mac

On the Mac, we configured QtRuby like this:
~/qtruby> ./configure --with-qt-dir=/Developer/qt \
   --enable-mac --prefix=/usr

Your configuration items may vary based on where Qt is installed (\n\-with-qt-dir) and where SMOKE will be installed (\-prefix). You may also \n\nadditional options.

If all goes well, you should see a positive note to that effect and a \nprompt to begin the build process. After this point, we followed the \nsteps outlined in the INSTALL file.

~/qtruby> cd smoke/qt
~/qtruby/smoke/qt> perl generate.pl
~/qtruby/smoke/qt> qmake -makefile
~/qtruby/smoke/qt> make
...
~/qtruby/smoke/qt> sudo make install

We also build the rest of the package, again using the steps high-\nlighted in the INSTALL file.

~/qtruby/smoke/qt> cd ../../..
~/qtruby> cd qtruby/rubylib/qtruby
~/qtruby/qtruby/rubylib/qtruby> ruby extconf.rb \
\-with-qt-dir=/Developer/qt --with-smoke-dir=/usr \
\-with-smoke-include=../../../smoke

~/qtruby/qtruby/rubylib/qtruby> make
...
~/qtruby/qtruby/rubylib/qtruby> sudo make install
...
~/qtruby/qtruby/rubylib/qtruby> cd ../../..
~/qtruby> cd qtruby/rubylib/designer/rbuic
~/qtruby/qtruby/rubylib/designer/rbuic> qmake -makefile
~/qtruby/qtruby/rubylib/designer/rbuic> make
...
~/qtruby/qtruby/rubylib/designer/rbuic> sudo make install
...
~/qtruby/qtruby/rubylib/designer/rbuic> cd ../uilib
~/qtruby/qtruby/rubylib/designer/uilib> ruby extconf.rb
   --with-qt-ruby-include=/../../qtruby
   --with-qt-dir=/Developer/qt
~/qtruby/qtruby/rubylib/designer/uilib> make
...
~/qtruby/qtruby/rubylib/designer/uilib> sudo make install

Verifying the installation

To ensure that QtRuby was correctly installed, we can use irb to verify that Ruby is able to properly find the library.

```ruby
~> irb
irb(main):001:0> require 'Qt'
=> true
irb(main):002:0>
```

On Mac OS, if QtRuby cannot link against the Qt library it will generate a runtime error:

```plaintext
dyld: NSLinkModule() error
dyld: Library not loaded: libqt-mt.3.dylib
   Referenced from: /usr/lib/ruby/. . ./qtruby.bundle
   Reason: image not found
Trace/BPT trap
```

On Linux, it generates a similar error:

```plaintext
libqt-mt.so.3: cannot open shared object file:
   No such file or directory - /usr/lib/ruby/. . ./qtruby.so
```
3.4 Installation Issues

Much like installing Qt, things can go wrong during this process. Here are a few points to try and help you through.

- If the error happened during the `configure` portion of the installation, make sure none of your arguments were misspelled. Also make sure that specified items exist. For example, if you specified the location of Qt, make sure that Qt is indeed installed in that directory.

- If the error happened during the `make` portion of the installation, see if it’s repeatable. We’ve seen errors due to buggy compilers. We’ve also seen them happen when the system is being overused and things get too hot. Faulty hardware, such as bad RAM, can also cause compiler faults.

- If the error happened during the `require `Qt` check, make sure that QtRuby is installed in your Ruby directory. For us, that directory is `/usr/lib/ruby/site_ruby/1.8`. There should be a Qt subdirectory and a Qt.rb file. Also, in your platform specific subdirectory, such as i686-linux or darwin-powerpc8.0, there should be a qtruby.so. This library links against the Qt library, so making sure that the Qt library is accessible to the dynamic library loader is important.
Instead of going through the routine of trying to learn the arcane details of QtRuby before using it, let’s jump straight into an example.

### 4.1 Your first program

Fire up your favorite text editor, and type in this program:

```ruby
require 'Qt'
app = Qt::Application.new(ARGV)
label = Qt::Label.new("Hello World", nil)
label.resize(150, 30)
app.setMainWidget(label)
label.show()
app.exec()
```

Then, try executing it.

```bash
~> ruby ex_hello_world.rb
```

If all went well, your program should popup something like Figure 4.1.
Let’s take a closer, line-by-line, look.

1. First, we load the QtRuby library.

2. Next, we create a `Qt::Application` object. `Qt::Application` is the foundation for all Qt programs. It handles all of the important behind-the-scenes details that are vital to our program running properly.

3. Creates a new `Qt::Label` object, and sets its text to *Hello World*. The `nil` argument will be explained shortly.

4. Resize the label to 150 pixels wide by 30 pixels tall to make sure the text is all visible.

5. Assigns the label as the main widget of the application.

6. Make the label visible.

7. Finally, we tell the application to start its processing.

### Comparing to C++

For comparison, the equivalent C++ code is:

```cpp
#include <qapplication.h>
#include <qlabel.h>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );

    QLabel label( "Hello world!", 0 );
```
label.resize(150, 30);

app.setMainWidget(&label);
label.show();
return app.exec();
}

Being the very astute reader you are, what fundamental differences did you notice between the two code styles?

Most importantly, you should have noticed the class names are slightly different. In the Qt world, all of the classes begin with the letter Q, like QApplication. The Ruby equivalent, however, lives in the Qt namespace, and as such, the Q is dropped. Thus, QApplication becomes Qt::Application. This convention holds true for all QtRuby classes.

From this point on, we will try to stick with the Qt::Classname naming style when referring to classes, unless we are referring to something that is Qt specific. Sometimes when we go deep into the Qt library, we’ll need to use the QClass syntax instead.

Now that your feet are wet, let’s wade a little deeper into the framework.

### 4.2 Objects and Widgets and Parents, oh my!

We’ve created our first QtRuby program and highlighted a couple of necessities. We’re ready to go a little deeper, but first we need to understand a little more about the basics of Qt.
Fundamentals

The fundamental class in QtRuby is the Qt::Object. This object contains many of the settings and properties that are needed in most of the classes we will be using.

Directly inheriting from Qt::Object is the Qt::Widget class. Qt::Widget is the base class for all GUI items, like sliders, text boxes, and labels. In fact, the Qt::Label class we used in the example on page on page 19 inherits from Qt::Widget.

The Qt::Widget class by itself is rather boring. We like to think of it as a blank canvas on which we can make a more interesting type of widget. Luckily for us, many of the commonly used types of widgets have already been created as part of the framework, such as the

---

1In general, any GUI object is referred to generically as a *widget*.
**Qt::Widget** is a good base class to use to make more sophisticated widgets.

Most widgets have multiple initializers taking different arguments lists.

Classes derived from **Qt::Widget** must have a **Qt::Widget** based parent.

Classes derived from **Qt::Object** only need a **Qt::Object** derived parent, which includes **Qt::Widget**.

---

**Qt::Label**, **Qt::SpinBox**, and **Qt::TextEdit** (screenshot on Figure 4.3).

As seen in the previous code example, the way to bring a widget alive is to initialize it with a call to `new()`. For example, this code creates a new **Qt::Widget** instance:

```cpp
widget = Qt::Widget.new(nil)
```

**Object ancestry**

But what is this mysterious parameter we've been passing to the initializer, you ask? It's the parent argument. Every time you create a new widget, you tell that widget who its parent widget is. Optionally, by passing nil, you're telling the widget it has no parent.

```cpp
w1 = Qt::Widget.new(nil)  # No parent
w2 = Qt::Widget.new(w1)   # w1 is parent
```
Parentless widgets are called *top level widgets*.

Here’s a slightly more complex example, shown diagrammatically in Figure 4.4

```ruby
# Create a widget with no parent
- toplevelwidget = Qt::Widget.new(nil)
- # Create children of the toplevelwidget
- child_1 = Qt::TextEdit.new(toplevelwidget)
- child_2 = Qt::ComboBox.new(toplevelwidget)
- # Create a grandchild of the toplevelwidget
- child_3 = Qt::Label.new("Some Text", child_1)
```

**How the family fits together**

A parent widget owns its children. Child widgets become contained within the physical geometry of the parent. Thus, if the parent gets destroyed, disposed of, or hidden, its children will also suffer the same fate. This feature is very valuable. We can create child widgets knowing that as long as they have a parent they will be cared for. If
this wasn’t the case, we could potentially have lots of spare widgets floating (literally) around.

For example, if you destroyed the `toplevelwidget` from line 2 above, you can be assured that `child_1` and `child_2` are destroyed. `child_3` would also be destroyed, since its parent, `child_1`, was destroyed.

**Why parents and children?**

You may be wondering the logic behind having to specify a parent when creating a new instance of a widget. It turns out that this methodology fits the GUI model very well.

The common convention is to have one top level widget for the application of which all the other widgets are children or grandchildren (or great grandchildren...). The top level widget for the application can be the blank canvas of a `Qt::Widget` or a more complex application interface like a `Qt::MainWindow`.

The parent/child model allows us to create objects and then not worry about their ownership. After creation, QtRuby handles all of the details for us.
The parent/child relationship is also used by Qt to help with the physical layout of the windows in the application, something we’ll see more of Section 5.3, *Understanding Layouts*, on page 38.

### Naming widgets

As well as the parent argument, `Qt::Object`-based initializers also accept an optional name argument. It is passed in immediately after the parent, like this:

```ruby
widget = Qt::Widget.new(parent_widget, "Fred")
```

The name can also be specified using the `setName()` method.

```ruby
widget = Qt::Widget.new(parent_widget) # Nameless
widget.setName("Fred")
```

Widgets whose names are not specified receive the name *unnamed*.

As we’ll see in the next section, there are ways to search groups of widgets by name, which is a reason why authors may choose to name their widgets. Another reason is that debugging and introspection tools can provide valuable insight when widgets have names. If your program has 100 `Qt::Label`s in it, and one of them was causing the program to crash, knowing the name of which one can help you track down the bug much faster.

### 4.3 The Qt Object Model

Through the heavily used base class `Qt::Object`, Qt provides a valuable set of introspection methods that can be used to query information about objects and their families. Many of these methods are fundamental to Ruby applications, but remember that Qt is a C++
toolkit and as such some of these ideas are not central to that language.

First, `Qt::Object` provides a `className()` method that returns the name of the class. Ruby provides the same information via the `class()` call.

```ruby
irb(main):001:0> w = Qt::Widget.new(nil)
irb(main):002:0> w.className
=> "Qt::Widget"
irb(main):003:0> w.class
=> Qt::Widget
```

Object types are also queryable via the `isA()` and `inherits()` methods. `isA()` returns true if the object is an instance of the class. `inherits()` returns true if the object has some inheritance back to the provided class.

```ruby
irb(main):004:0> w.isA("Qt::Widget")
=> true
irb(main):005:0> w.isA("Qt::Object")
=> false
irb(main):006:0> w.isA("QWidget")
=> false
irb(main):007:0> w.isA("QObject")
=> false
```

```ruby
irb(main):008:0> w.inherits("Qt::Widget")
=> true
irb(main):009:0> w.inherits("Qt::Object")
=> true
irb(main):010:0> w.inherits("QWidget")
=> true
irb(main):011:0> w.inherits("QObject")
=> true
```

Note that in the previous example, `inherits()` returns true for both `QObject` and `Qt::Object` syntax. In previous versions of QtRuby, this
was not the case; `inherits()` only returned true if you used the Qt style naming convention. Be careful of this fact if you are not using the most recent version available of QtRuby.

**Studying the Family**

We can also inspect a widget’s relatives. The `parent()` and `children()` methods return the widget’s direct ancestors.

```ruby
irb(main):013:0> w = Qt::Widget.new(nil)
irb(main):014:0> w2 = Qt::Widget.new(w)
irb(main):015:0> w3 = Qt::Widget.new(w)
irb(main):017:0> w2.parent == w => true
irb(main):016:0> w.children
=> [ w2, w3 ]
```

The `children()` method returns the widget’s children in the order in
which they were created as children of the widget. The `children()` method returns only direct children of the parent, not grandchildren.

To explicitly search for a child, you can use the `child()` method. The syntax is `child(name, inheritsClass = nil, recursiveSearch = true)`.

- # Search all children and grandchildren
  - # for a child named "Editor"
  - obj = w.child("Editor")

- # Search all children and grandchildren
  - # for a Qt::LineEdit named "Editor"
  - obj = w.child("Editor", "Qt::LineEdit")

- # Search only direct children
  - # for a child "Editor"
  - obj = w.child("Editor", nil, false)

- # Search only direct children
  - # for a Qt::LineEdit named "Editor"
  - obj = w.child("Editor", "Qt::LineEdit", false)
The `child()` method returns the first child found fitting the search criteria. To query for multiple children, the `queryList()` method can be used. The syntax is `queryList(inheritsClass = nil, objName = nil, regexpMatch = true, recursiveSearch = true)`.

```ruby
irb(main):004:0> w = Qt::Widget.new(nil)
irb(main):005:0> w2 = Qt::Widget.new(w,"Widget")
irb(main):006:0> w3 = Qt::Widget.new(w,"Widget3")
irb(main):007:0> w4 = Qt::Widget.new(w,"Foo")
```

```ruby
irb(main):012:0> w.queryList("Qt::Widget")
=> [ w2, w3, w4 ]
irb(main):013:0> w.queryList("Qt::Widget","Widget")
=> [ w2, w3 ]
irb(main):013:0> w.queryList("Qt::Widget","Widget", false)
=> [ w2 ]
```

### 4.4 Other initialization items

Notice line 7 from the example on page 24.

```ruby
child_3 = Qt::Label.new("Some Text", child_1)
```

The initialization of the `Qt::Label` class also includes a string literal to set the label's value. This is very common practice with Qt widgets—many have initializers which can take extra arguments to seed initial settings of the widget. One thing to note: the parent argument usually comes after these initial value arguments.

Indeed, we also could have written:

```ruby
child_3 = Qt::Label.new(child_1)
child_3.setText("Some Text")
```

You can also pass a block to the object initializer if preferred.
4.5 The Qt::Application class

In Section 4.1, Your first program, on page 19, we briefly discussed the necessity of a Qt::Application class. Let’s examine this a little further.

Qt::Application is the heart of the QtRuby application. It handles most of the underlying details that make up a GUI application—things like maintaining a common look and feel amongst widgets, managing an interprogram clipboard, mouse cursor settings, and internationalization of user visible text. It also talks with the window system and dispatches events to the widgets in the program.

The Qt::Application object is the first QtRuby object your program should initialize. Otherwise, your application most likely will abort.

Starting the Event Loop

After the Qt::Application instance has been created, you can initialize the widgets that make up the program.

child_3 = Qt::Label.new(child_1) { setText("Some Text") }
After the program has been completely set up, you call the `exec()` method of the `Qt::Application` object. The `exec()` method starts the event loop processing of the application. The event loop waits for GUI events to happen and processes them accordingly. For example, it might see that a keyboard button was pressed and attempt to send the information about the button press event to the widget which is interested in receiving it.

The `exec()` method only returns when the `mainWidget` of the application is destroyed or `Qt::Application’s exit()` is called.

```ruby
app = Qt::Application.new(ARGV)
widget = Qt::Widget.new(nil)
app.setMainWidget(widget)
app.exec
```

Event driven programming

Many programmers, even experienced ones, struggle the first time they write a GUI application. Most GUI applications have *event driven flow*, which differs from the linear flow that most common programming languages are written in.

In a QtRuby application, the event loop handles all of the processing of information. Prior to starting the event loop (using the `exec()` method of the `Qt::Application` class), we specify the types of events we are interested in and what to do when these events happen. In the most basic form, this is handled by signal and slot connections as described in Section 5.5, *Signals and Slots*, on page 47.

We’d like to stress that once the event loop has started, there is...
no real direct control over what’s happening. That is, we don’t have a section of Ruby code that is looping over and over again like in a linear flow program. Instead, we predefine the processing we’d like to have happen when events occur and we let the event loop take care of looking for these events and dispatching them to us accordingly.

We’ll see examples of how this works shortly.

Well, it seems like we’ve been pretty thorough in our discussion on the basics of Qt’s widgets. When you’re ready, let’s tie together what we’ve learned.

### 4.6 Summary

- All QtRuby widgets inherit from the base class `Qt::Widget`. This in turn inherits from `Qt::Object`.

- All QtRuby widgets fit into an overall family tree structure. Child widgets are contained within the physical geometry of the parent. Destruction of a widget causes all of its descendants to be destroyed as well.

- Every QtRuby program needs one and only one `Qt::Application` instance. It must be created before any GUI widgets are initialized.

- The application event loop starts with a call to `Qt::Application’s exec()` method. The method only returns when the main application widget is destroyed.
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 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.

```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)
a.setMainWidget(mw)
mw.show
```

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.
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.

```ruby
class MyWidget < Qt::Widget
    # Initialization code...

    # Child widget creation...
    @label = Qt::Label.new(self)
    @button = Qt::PushButton.new(self)
    @layout = Qt::VBoxLayout.new(self)
end
```

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.

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

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

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()`.

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

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:
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:

```ruby
@clicked_times = 0

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

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.

```ruby
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, on the following page

## 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.

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)`.

---

**Figure 5.1: Screenshot of Example 2**
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 conve-
The Qt Layout Classes guide gives some more insight into the use of these classes.

Figure 5.3: Layout class inheritance diagram

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

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

**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.
Sublayouts

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

```ruby
@layout = Qt::HBoxLayout.new
@sublayout = Qt::VBoxLayout.new
@w1 = Qt::Widget.new
@w2 = Qt::Widget.new
@w3 = Qt::Widget.new

@sublayout.addWidget(w1)
@sublayout.addWidget(w2)

@layout.addLayout(@sublayout)
@layout.addWidget(@w3)
```

```c++
gl.addWidget(w, 0, 0)
```
In Figure 5.6, on the next page we demonstrate why sublayouts are convenient. On the left side we created a `QVBoxLayout` containing three `QCheckBoxes`. Then we nested this layout inside of a `QHBoxLayout` and also put in a `QDial`. As you can see, the sublayout allows us to group related items together in a logical way and maintain the size and spacing policies we desire.

**Figure 5.5: Layout and Sublayout Example**

### Layout properties

All layouts have two fundamental properties, margin and spacing. These are shown on Figure 5.7, on page 43. **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 `QLayout`, there are...
some other interesting additions. `addSpacing()` allows you to add a fixed amount of space directly in the widget. `addStretch()` adds a stretchable 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.
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 `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`.

```ruby
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:

<table>
<thead>
<tr>
<th>Qt::SizePolicy</th>
<th>Minimum</th>
<th>Size</th>
<th>Preferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed</td>
<td><code>sizeHint()</code></td>
<td><code>sizeHint()</code></td>
<td><code>sizeHint()</code></td>
</tr>
<tr>
<td>Minimum</td>
<td><code>sizeHint()</code></td>
<td>none</td>
<td><code>sizeHint()</code></td>
</tr>
<tr>
<td>Maximum</td>
<td>none</td>
<td><code>sizeHint()</code></td>
<td><code>sizeHint()</code></td>
</tr>
<tr>
<td>Preferred</td>
<td>none</td>
<td>none</td>
<td><code>sizeHint()</code></td>
</tr>
<tr>
<td>Minimum-Expanding</td>
<td><code>sizeHint()</code></td>
<td>none</td>
<td>all available space</td>
</tr>
<tr>
<td>Expanding</td>
<td>none</td>
<td>none</td>
<td><code>sizeHint()</code>, will expand as necessary</td>
</tr>
<tr>
<td>Ignored</td>
<td>none</td>
<td>none</td>
<td>all available space</td>
</tr>
</tbody>
</table>

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

In Figure 5.8, on the next page we show two different ways of sizing widgets within the same layout. On the left side, we set the vertical `Qt::SizePolicy` of each of the contents to MinimumExpanding, which equalizes the spacing of all of the widgets. On the right, we set the `Qt::SizePolicy` of the two widgets to Preferred, which only takes up the amount of space the widget internally calculates that it needs. The spacer at the top has a `Qt::SizePolicy` of Expanding, which allows it to take up the rest of the space available in the layout.
5.4 Automating a task

In our example program at the beginning of the chapter there was an initialization of the text on @label to how many times the button has been clicked.

@clicked_times = 0

@label.setText("The button has been clicked " +
  @clicked_times.to_s + " times")
@button.setText("My Button")

But, any further clicks on the button do not change the text of label. Using QtRuby’s concept of signals and slots, we can change this behavior.

The first step is to define a method in our MyWidget class that handles updating the label text each time the button gets clicked.

def button_was_clicked

Note: We could have avoided these calls to setText() by using the initializers that set the text for us.
@clicked_times += 1

@label.setText("The button has been clicked " + 
    @clicked_times.to_s + " times")
end

What we want to do next is call this method any time that @button is clicked. Widget events, like the clicking of @button, are referred to as widget signals. The reactionary methods that we want to happen when these signals take place are called slots. By connecting these signals and slots together, we can automate the process of having @label update its text.

Before we get too far into signals and slots, let’s go back to our program to see just how easy using them really is. In order to take advantage of the automation, we need to define exactly what is a signal and what is a slot. Since our @button widget is a built in Qt::PushButton, its clicked() signal is already defined for us. However, the slot we will be connecting it to, button_was_clicked() has not been defined as a slot. A simple line in our MyWidget class handles that for us:

slots 'button_was_clicked()'

And last, in the MyWidget initializer, we need to connect the signal and slot together:

connect(@button, SIGNAL('clicked()'),
    self, SLOT('button_was_clicked()'))

When put all together, our final program looks like this:

require 'Qt'
class MyWidget < Qt::Widget
    slots 'button_was_clicked()'
    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 
  \+ \( \text{\@clicked\_times.to\_s} \+ \text{"times"} \))
@button.setText("My Button")

connect(@button, SIGNAL('clicked()'),
  self, SLOT('button\_was\_clicked()'))

end

```
def button\_was\_clicked
  \@clicked\_times += 1

  @label.setText("The button has been clicked " +
    \( \text{\@clicked\_times.to\_s} \+ \text{"times"} \))

end```

end

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

5.5 Signals and Slots

The example from the previous section showed how the connection of signals and slots can be harnessed to create dynamic applica-

C# users will find that Qt’s signals and slots are very similar to delegates from that language.
In this section, we will explore the concept of signals and slots even further.

Signals are triggers that happen in response to some kind of event. A widget whose signal has been activated is said to be emitting its signal. Usually, signals are emitted when a very simple event has occurred, such as the clicking of a button. Most widgets emit multiple types of signals to inform other widgets that an event has occurred. For example, a `Qt::LineEdit`, which is a one line text editor, emits signals when the text has changed, the return key was pressed, someone selects text with the mouse, or when the widget loses focus (see Figure 5.9, on the next page).

Some signals convey all of their information within their names. The `returnPressed()` signal from the `Qt::LineEdit`, for example, explains exactly what happened (the return key was pressed) when that signal is emitted. Other signals, like `Qt::LineEdit`’s `textChanged()` signal, tell us that the text has changed, and also contain the text that has changed in the signal.

This extra signal information is passed as an argument to the signal. In this case, the method signature is `textChanged(const QString &)`. When the signal gets emitted, this extra information goes with it.

Slots, the things we connect signals to, are ordinary methods within a widget class. Defining these methods as slots creates the ability to use signals to activate these slot methods automatically.

**Signals and Slots—The Connection**

Signals and slots are connected together using the `connect()` method defined in the `Qt::Object` class. The syntax of the `connect()` is:
The code snippets in this section demonstrate the syntax of making signal and slot connections. Remember, as we discussed in Chapter 4, Get Your Feet Wet, on page 19, a full Qt program will need a few more things set up (such as an instance of the Qt::Application class).

This syntax works as long as the connect() method is being used from within an object that inherits from Qt::Object. To make connections outside of a Qt::Object class, the method must be called more explicitly:

```cpp
Qt::Object::connect( object_with_signal, SIGNAL( 'signalName()' ),
                    object_with_slot, SLOT( 'slotName()' ) )
```

The most basic form of connection is between a signal and slot that have no arguments. For example, the following code causes
If you misspell a signal or slot name during connection QtRuby will generate a runtime error message on the application’s standard error file descriptor.

Figure 5.10: Simple Slot Connection

a Qt::LineEdit to clear when a Qt::PushButton is clicked (as seen in Figure 5.10).

```
@button = Qt::PushButton.new
@lineedit = Qt::LineEdit.new
Qt::Object::connect( @button, SIGNAL( 'clicked()' ),
                    @lineedit, SLOT( 'clear()' ) )
```

Signals and slots with arguments connect in the exact same way, as long as their arguments are of the same type (as seen in Figure 5.11, on the following page).

```
@lineedit = Qt::LineEdit.new
@label = Qt::Label.new
Qt::Object::connect( @lineedit,
                     SIGNAL( 'textChanged(const QString &)' ),
                     @label,
                     SLOT( 'setText(const QString &)' ) )
```
When connecting signals and slots, the Qt style syntax (not the QtRuby style syntax) is used. The method names and arguments are passed as strings wrapped by either SIGNAL() or SLOT().

Note that in the above example, the SIGNAL and SLOT arguments are listed using the Qt style syntax like `textChanged(const QString &)` and not the QtRuby style syntax like `textChanged(const Qt::String)`. This detail is very important. While QtRuby handles the conversion of Qt’s `QString` to Ruby’s `String` automatically, the definition of signals, slots, and the connection of the two must utilize the Qt style syntax. This is a QtRuby limitation, and is anticipated to be fixed in a later release of the toolkit.

**More advanced connections**

Another feature of signals and slots is the ability to connect a signal to more than one slot. Consider this code (the resulting structure is outlined in Figure 5.12, on the next page):

```ruby
@lineEdit_1 = Qt::LineEdit.new
@lineEdit_2 = Qt::LineEdit.new
```

Figure 5.11: Slot Connection with Arguments
When a signal is emitted, the order that the connected slots are executed in is arbitrary.

One caveat to multiple slot connections is that there is no defined order in which the slots are executed. That is, the order in which you make the connections is not guaranteed to be the order in which the slots are called.

It's also okay to connect a signal to another signal (as shown on Figure 5.13, on the following page).
Figure 5.13: Signal to Signal Connection

```
Qt::Object::connect( @lineedit_1,
    SIGNAL( 'textChanged(const QString &)' ),
    @lineedit_2,
    SIGNAL( 'textChanged(const QString &)' ) )
Qt::Object::connect( @lineedit_2,
    SIGNAL( 'textChanged(const QString &)' ),
    @label,
    SLOT( 'setText(const QString &)' ) )
```

This syntax allows one signal to trigger another signal, which then would trigger any slots connected to the second signal.

You can even connect to a slot which takes fewer arguments:

```
@lineEdit = Qt::LineEdit.new
@bar = Qt::StatusBar.new
Qt::Object::connect( @lineEdit,
    SIGNAL( 'textChanged(const QString &)' ),
    @bar,
    SLOT( 'clear()' ) )
```

The information that would normally be passed via the signal argu-
ment is discarded.

However, the opposite is not true:

```csharp
@button = Qt::PushButton.new
@label = Qt::Label.new

# This doesn't work
Qt::Object::connect( @button, SIGNAL( 'clicked()' ),
                   @label, SLOT( 'setText(const QString &)') )
```

The above code generates the following error which is displayed during runtime:

```
QObject::connect: Incompatible sender/receiver arguments
Qt::PushButton::clicked() --> Qt::Label::setText(const QString &)
```

**Disconnecting Signals and Slots**

Signal-slot connections can also be disconnected via the same syntax:

```csharp
@button = Qt::PushButton.new
@bar = Qt::StatusBar.new
Qt::Object::connect( @button, SIGNAL( 'clicked()' ),
                     @bar, SLOT( 'clear()' ) )
# Perform a disconnection
```
Qt::Object::disconnect( @button, SIGNAL( 'clicked()' ),
    @bar, SLOT( 'clear()' ) )

**Tying it all together**

The power with signals and slots lies in their flexibility. Signals can be used from existing widgets, or created in new widgets. New slots can be made in custom widgets that mask internal child widgets. Most importantly, these signals and slots cross widget boundaries and allow us to encapsulate child widgets through a parent widget interface.

Let’s see it in action. Consider this class:

```ruby
class MyTimer < Qt::Widget
  signals 'tripped_times_signal(int)'
  slots 'timer_tripped_slot()'

  def initialize(parent)
    super(parent)

    @timer = Qt::Timer.new(self)
    @label = Qt::Label.new(self)
    @tripped_times = 0

    connect(@timer, SIGNAL('timeout()'),
            self, SLOT('timer_tripped_slot()'))
    # Make the timer trip every second (1000 milliseconds)
    @timer.start(1000)

  end

  def timer_tripped_slot()
    @tripped_times += 1;
    @label.setText("The timer has tripped " +
                   @tripped_times.to_s + " times")
  end
```

In this example, we create a Qt::Timer that gets activated every second (1000 milliseconds). Each time @timer is activated, its timeout() signal is emitted. We’ve connected the timeout() signal to the timer_tripped_slot() slot. This slot updates the text on the label to reflect the total number of times the timer has tripped. The slot also emits the tripped_times_signal(), telling how many times the timer has tripped. The MyTimer does not make use of the tripped_times_signal() signal, but an external class might use that information by connecting the signal to one of its slots. We highlight this code example on Figure 5.15.
### 5.6 Slot Senders

Sometimes during a slot it is useful to know how the slot got started. The `sender()` method returns the `Qt::Object` which was responsible for the slot call. The `sender()` method only works for a slot when it was activated by a signal—manually calling the slot does not work.

Consider the following code:

```ruby
require 'Qt'

class SignalObject < Qt::Object
  signals 'mySignal()

  def initialize(parent=nil)
    super(parent)
  end

  def trigger
    emit mySignal()
  end
end

class SlotObject < Qt::Object
  slots 'mySlot()

  def initialize(parent=nil)
    super(parent)
  end

  def mySlot
    puts "Slot called by #{sender.class}" end
end

sig = SignalObject.new
```

```ruby
```
slot = SlotObject.new

Qt::Object::connect(sig, SIGNAL('mySignal()'),
                    slot, SLOT('mySlot()'))

Now look at the effects on the `sender()` method in a slot when it's activated by a signal:

```
irb(main):001:0> sig.trigger
Slot called by SignalObject
```

versus when it's called manually:

```
irb(main):002:0> slot.mySlot
Slot called by NilClass
```

### 5.7 Summary

- Custom widgets should inherit from base class `Qt::Widget`.
- Widgets have a two-dimensional geometry. This geometry can be set manually or handled automatically through layouts.
- Widgets define signals that are emitted when certain spontaneous events occur. They also define slots which are reactionary methods that can be connected to these signals.
- Widget slots can use the `sender()` method to find out how they were activated.
At this point, we've really tackled most of the concepts needed to make a robust QtRuby application. However, there's still a bit more to do.

### 6.1 Event Methods

Our earlier discussion about event driven programming led into the concept of signals and slots. But there's more to events than just signal emission. Remember, in the GUI world, events are the underlying paradigm of the program operation.

It turns out that many of the these GUI events are so important that they are handled in a much more direct way than just as an emitted signal. Instead, there are *event methods* which are directly called within a `Qt::Widget` object.

#### One event from start to finish

For the moment, let's look at one type of event: the mouse press. When a mouse button is pressed the following series of things happens:

1. The window system recognizes the mouse press, and passes the mouse information to the QtRuby application.

2. The application uses the information to create a `Qt::MouseEvent` object, containing information about which button was pressed and the location of the mouse when the button was pressed.
3. The application then determines which widget the mouse was directly on top of when the button was pressed. It dispatches the `Qt::MouseEvent` information to this widget’s `mousePressEvent()` method.

4. The widget chooses what to do at this point. It can act on this information, or can ignore it. If it ignores the information, the `Qt::MouseEvent` then gets sent on to the parent of the widget, to be acted upon.

5. The `Qt::MouseEvent` continues up the hierarchy until a widget accepts the event, or it reaches the top level and cannot go any further.
An Example of Qt::MouseEvent

```ruby
require 'Qt'

class MouseWidget < Qt::Widget
  def initialize(parent=nil)
    super(parent)
    @button = Qt::PushButton.new("PushButton", self)
    @layout = Qt::VBoxLayout.new(self)
    @layout.addWidget(@button)
    @layout.addStretch(1)
  end

  def mousePressEvent(event)
    if (event.button == Qt::RightButton)
      Qt::MessageBox::information(self, "Message",
          "You clicked the widget")
    else
      event.ignore
    end
  end
end

app = Qt::Application.new(ARGV)
mw = MouseWidget.new
app.setMainWidget(mw)
mw.show
app.exec
```

In this example the MouseWidget has implemented the mousePressEvent method, meaning that it wishes to handle this mouse event internally (see Figure 6.2, on the next page).

The MouseWidget checks the mouse button that was pressed to start the event—if it was the right (as opposed to the left) button, then
When creating a custom event method, such as `mousePressEvent()`, it’s important to remember that you are overriding the base class method with your own. If your goal is not to replace the base class functionality, but to extend it, then you must be sure to call the base class event method from within your event method.

It pops up a message. Otherwise, it ignores the event, which then would get passed on to the parent (if there was one).

The `mousePressEvent()` only gets invoked for presses in the empty space of the `MouseWidget`, however. Notice that the `MouseWidget` also contains a `Qt::PushButton`, but if that `Qt::PushButton` gets pressed, it has its own internal handling of the `mousePressEvent()`, and the `MouseWidget` never sees a `mousePressEvent()`.

More methods

Obviously, there are more event classes than just `Qt::MouseEvent`. `Qt::Widget` also has specialized handlers for these events. We’ve created a chart to overview the event methods and handlers available in `QtRuby` in Appendix A, on page 88.

We’ve seen how to create event methods that are invokeded auto-
matically when certain events happen. Another powerful aspect of the QtRuby toolkit is the ability to install event filters between objects.

### 6.2 Event Filters

Event filters allow objects to listen in to and intercept events from other objects. To create an event filter, we use the `installEventFilter()` method that is part of `Qt::Object`.

```ruby
object_to_be_filtered.installEventFilter( intercepting_object )
```

With this syntax, all of `object_to_be_filtered`'s events get sent directly to `intercepting_object`'s `eventFilter()`.

The `eventFilter()` method has two arguments, the `Qt::Object` that is being filtered and the `Qt::Event` that was received. With this information, the `eventFilter()` method can intercept the event and perform the desired action.

If `eventFilter()` returns `true`, the event is considered to have been successfully intercepted. Otherwise, the event will continue to propagate through the normal event handling chain. We show the event filter logic on Figure 6.3, on the next page.

Here's an example of an event filter:

```ruby
require 'Qt'

class MouseFilterWidget < Qt::Widget

  def initialize(parent=nil)
    super(parent)
    @button = Qt::PushButton.new("PushButton", self)
    @layout = Qt::VBoxLayout.new(self)
    @layout.addWidget(@button)
    @layout.addWidget(@button)
  end

end
```
@layout.addStretch(1)

# Override events for the button
@button.installEventFilter(self)
end

def eventFilter(obj, event)
    if(obj == @button && event.type == Qt::MouseButtonPress)
        if(event.button == Qt::RightButton)
            Qt::MessageBox::information(self, "Message", "You right clicked the button")

Figure 6.3: Event Filter Flow
This code installs an event filter on a `Qt::PushButton` referenced by `@button`. The `eventFilter()` method then checks if we've received a button press for the `@button` and if it's the right mouse button, display a message. Otherwise, event handling occurs as normal. This means that a left button click results in the same thing that would normally happen—the button accepts the click.

### 6.3 The Main Event

The event methods we've seen so far are specialized event methods that get called for specific types of events. `Qt::Object` also has a generic event method, which handles the dispatching of the specialized events. This method, aptly named `event()`, can also be overridden to provide customized behavior. `Qt::Widget` overrides this method to handle GUI related events.

Thus, the overall logic of a widget's event handling is as follows:

1. A new event (`Qt::Event`) is created, and gets passed to the `event()` method.
2. Assuming that the default `event()` method has not been over-ridden, `QWidget`'s `event()` method is executed. Otherwise, the custom `event()` method is called.

3. The default `event()` checks for any installed event filters, and sends the event to the filter if installed.

4. If the event is intercepted, then we're done. Otherwise, `event()` determines what type of `QEvent` it's processing, and converts the `QEvent` into the a new class (such as `QMouseEvent`).

5. The proper specialized event method then gets called (in this case, that's `mousePressEvent()`).

### 6.4 The Event Loop

When a new event occurs, the QtRuby application does not act on it immediately. Instead, the event goes into a waiting queue. The mandatory `QApplication` object is the keeper of this event queue. Periodically, `QApplication` checks this queue and dispatches the pending events to their proper places. These events are referred to as asynchronous events. This cycle of storing the events and then acting on them is referred to as the **event loop**.

A typical QtRuby program has only one thread of execution. This means that when the `QApplication` is ready to act on the queued events, it must dispatch all of them to the proper objects before it can come back to handle the next batch. Ideally, this process happens very quickly. If, however, your program has some computationally intensive code, such as the opening of a large file, this could lead to a slowdown of the event loop processing.
If your application spends a large amount of time handling a certain event it may become unresponsive to other events that occur later. To a user of a GUI program, this unresponsiveness is highly undesirable. For example, if you click on the mouse, you want the receiving widget to act relatively fast. Waiting a few seconds for the widget to respond to the mouse click is usually unacceptable.

**Maintaining a Responsive Application**

There are a few ways to keep the application responsive. One is to use threads. In the Qt world, it is possible to create a separate thread of execution using a `QThread`. Unfortunately, a compatible `Qt::Thread` class doesn’t exist in the QtRuby toolkit. We hope that a future version of QtRuby will include one.

Another common method of maintaining a responsive application is to break the intensive computation up into smaller segments and use a `Qt::Timer` to periodically trigger a slot that does a small amount of the work. This method keeps the application responsive to events and also allows a good portion of work to continue on in the background.

```ruby
# Bad
def someSlot()
  really_expensive_computation()
end

# Better
timer = Qt::Timer.new(self)
connect(timer, SIGNAL('timeout()'), self, SLOT('someSlot()'))
timer.start(100) # Trigger the timer every 100 milliseconds

def someSlot()
```
6.5 Event posting

Sometimes we want to generate our own events to send to other objects. The easiest way uses `Qt::Application`'s `postEvent` method. This takes the event and receiver that you define and puts the event in the event queue.

```plaintext
small_portion_of_expensive_computation()
end
```

```plaintext
button = Qt::PushButton.new(nil)
# Construct a Qt::MouseEvent
event = Qt::MouseEvent.new(
    Qt::Event::MouseButtonPress,
    Qt::Point.new(0,0),
    Qt::LeftButton,
    0 )

# Send the event to button, asynchronously
Qt::Application::postEvent(button,event)
# Continue on - the mouse event will be sent later
```

The event loop takes control of the event once you post it. Thus, any event you construct and send to an object via `postEvent` becomes the property of the event loop. You shouldn't attempt to use the same event again. Construct a new event, if necessary.

It's also possible to *synchronously* send an event directly to another object using the `sendEvent` method. This works just like `postEvent`, except there is no delay—the event is handled immediately.

```plaintext
# Send the event to button, synchronously
Qt::Application::sendEvent(button,event)
```
At this point, button has already received the `mouseEvent`

`sendEvent()` returns a boolean value specifying whether the object accepted the event or not.

`postEvent()` is favored over `sendEvent()`, because it allows the event system to work asynchronously.

### 6.6 Summary

- Widgets have a collection of event methods (`mousePressEvent`, `resizeEvent`, ...) that get called when these certain events happen to the widget.
- Widgets can choose to ignore events, in which case the event gets sent on to the widget’s parent.
- Widgets can intercept and filter events that were destined to go to other widgets.
- New events can be posted directly into the event queue. They can be created both synchronously and asynchronously.
7.1 Qt Modules

Qt comes with a set of extra modules also available through QtRuby. Some of these modules are not directly GUI related, but useful functions for many GUI applications.

There is an overlap of functionality between these Qt modules and the libraries and modules that come with Ruby. In some instances, it may make more sense to use the Ruby libraries instead of the Qt ones. In other cases, the QtRuby classes may integrate easier with your QtRuby application, because of their built in signal and slot methods.

Network Module

The network module simplifies network programming. There are three levels of classes available in the module.

- **Low level**—classes such as `Qt::Socket` and `Qt::ServerSocket`, a TCP client and TCP server, respectively.

- **Abstract Level**—abstract classes such as `Qt::NetworkOperation` and `Qt::NetworkProtocol` that can be used to create subclasses that implement network protocols.

- **Passive Level**—classes such as `Qt::Url` which handles URL parsing and decoding.
**SQL Module**

The SQL module provides a database-neutral way of handling common SQL database operations: updates, inserts, and selects, and so on.

In order to be able to handle connections of a specific database, Qt has to be configured for that database during its initial setup. This is specified as a configure option, as noted in Section 2.4, *How to install Qt from source*, on page 8. This also requires database drivers to be present at configuration time.

**XML Module**

Qt provides interfaces to two XML parsers:

- SAX2—an event-based standard for XML parsing, and
- DOM—a mapping of XML to a tree structure

**OpenGL Module**

OpenGL is a standard API for creating three-dimensional objects. Qt provides a set of classes that support OpenGL drawing from within an application.

OpenGL support must be specified as a configure option, as noted in Section 2.4, *How to install Qt from source*, on page 8. This requires that the OpenGL libraries be present at configuration time.

**Canvas Module**

The Canvas module provides a two-dimensional blank canvas on which primitive geometric and text drawing structures can be created to form complex pictures.
Other GUI related modules

Qt also provides a number of GUI related modules:

- **Iconview.** `Qt::IconView` visualizes multiple items in icon form.
- **Table.** `Qt::Table` displays and edits tabular data.
- **Workspace.** `Qt::Workspace` can contain a number of windows.

7.2 QtRuby tools

QtRuby comes with a number of additional tools which can help you create custom GUI applications.

QtRuby UIC

One facet of Qt is Qt Designer, a GUI application that allows you to design custom widgets graphically.

Qt Designer generates `.ui` (user interface) files, which are XML files describing the widget properties. Qt’s build system uses these `.ui` files to generate valid C++ code that gets compiled into the project. The Qt utility that does this is **uic**, or User Interface Compiler.

QtRuby comes with the **rbuic** utility to generate Ruby code out of `.ui` files. To generate QtRuby code from a `.ui` file, use this syntax:

```bash
$ rbuic mywidget.ui -o mywidget.rb
```

You can use the `-x` switch to direct **rbuic** to generate the code to handle the creation of the `Qt::Application`.

```bash
$ rbuic mywidget.ui -x -o mywidget.rb
```
Figure 7.1: Screenshot of Qt Designer
This generates the Ruby code, and adds the following to the end of the file:

```ruby
if $0 == __FILE__
  a = Qt::Application.new(ARGV)
  w = MyWidget.new
  a.setMainWidget(w)
  w.show
  a.exec
end
```

With this code in place, you’re got a Ruby application that’s ready to run:

```bash
$ ruby mywidget.rb
```

### QtRuby API lookup

The rbqtapi command will look up methods available for a class in the QtRuby API.

```bash
$ rbqtapi QTimer
Timer* QTimer::QTimer()
Timer* QTimer::QTimer(QObject*)
Timer* QTimer::QTimer(QObject*, const char*)
void QTimer::changeInterval(int)
const char* QTimer::className() const
...
```

It’s also possible to search for classes containing certain method names using the `-r` option.

```bash
$ rbqtapi -rsetName
void QColor::setNamedColor(const QString&)
void QDir::setNameFilter(const QString&)
QDomNode QDomNamedNodeMap::setNamedItem(const QDomNode&)
QDomNode QDomNamedNodeMap::setNamedItemNS(const QDomNode&)
void QFile::setName(const QString&)
```
7.3 Taking Advantage of Ruby

So far, in all of our discussion about QtRuby we've attempted to keep the syntax as close to pure Qt as possible. We've done this to keep the interface and examples similar to what they would be had they been written in C++.

Now we're going to present some Ruby specific extensions to the language.

Qt object properties are typically written to using the `setProperty-Name()` syntax. For example, the `Qt::Application` class has the method `setMainWidget()` which we've used in many previous examples.

QtRuby simplifies this a little bit by allowing you to use the more Rubylike `propertyName =` syntax. This means that:

```ruby
app = Qt::Application.new(ARGV)
app.setMainWidget(widget)
```

becomes:

```ruby
app = Qt::Application.new(ARGV)
app.mainWidget = widget
```

Similarly, methods with names beginning `is` or `has` can be changed into the more idiomatic predicate method form:

```ruby
widget = Qt::Widget.new(nil)
widget.isTopLevel and puts "Widget is top level"
widget.hasMouse and puts "Mouse is over widget"
```
becomes:

```ruby
widget = Qt::Widget.new(nil)
puts "Widget is top level" if widget.topLevel?
puts "Mouse is over widget" if widget.mouse?
```

Note that you can use either style in your QtRuby programs. The first style is more Qt like while the second is more Ruby like.

### Your Choice of Case

As you may have noticed, Qt class methods use the camel case naming convention, with the first word always being lower case.

QtRuby also provides an interface (via Ruby’s `method_missing()` functionality) to use an all lowercase notation with underscores between the words. For example, the following two lines of code are equivalent.

```ruby
widget.someLongMethodName # Qt way
widget.some_long_method_name # ok in QtRuby
```

### 7.4 Disposing of Widgets

In a native C++ Qt program, widgets are created using the operator `new`, similar to Ruby’s `new()` initializer. However, since C++ lacks automatic memory management, it also has a `delete` operator which can destroy objects. Ruby has no such command, instead relying on garbage collection to remove objects which are no longer being referenced.

While this usually “just works,” there are times when it is valuable to be able to destroy objects that are not lined up for garbage collection. QtRuby provides this ability with the `dispose` methods.

*Note: The `dispose()` method is local to QtRuby and is not available in the Qt toolkit.*
To free an object, use the dispose() method. You can also use the disposed?() method to see if an object has been disposed.

```ruby
@parent = Qt::Widget.new(nil)
@child = Qt::Widget.new(@parent)

# Dispose the parent
@parent.dispose

# @child should be disposed if @parent was
@child.disposed? and puts "Child disposed"
```

### 7.5 Debugging a QtRuby Application

When things don’t quite work right, you sometimes need the ability to view a little deeper within the goings-on of the toolkit to see exactly what is going wrong. QtRuby provides some debugging methods for this.

The most common problem by far is the unresolved method error:

```ruby
irb(main):001:0> require 'Qt'
=> true
irb(main):002:0> app = Qt::Application.newARGV)
=> #<Qt::Application:0xb6b1795c name="irb">
irb(main):005:0> w1 = Qt::Widget.new("blah", nil)
 ArgumentError: unresolved constructor call Qt::Widget
```

This error happens when you attempt to call the method (in our case, the initializer) with an argument list not recognized by QtRuby.

To diagnose this, you can turn on more extensive debugging output by setting the variable Qt.debug_level. The debug levels are:

- Off
• Minimal
• High
• Extensive

```ruby
terb(main):007:0> Qt::debug_level = Qt::DebugLevel::High
=> 2
terb(main):008:0> w1 = Qt::Widget.new("MyWidget", nil)
classname == QWidget
:: method == QWidget
-> methodIds == []
candidate list:
No matching constructor found, possibles:
  QWidget* QWidget::QWidget()
  QWidget* QWidget::QWidget(QWidget*, const char*)
  QWidget* QWidget::QWidget(QWidget*)
  QWidget* QWidget::QWidget(QWidget*, const char*, Qt::WFlags)
setCurrentMethod()
ArgumentError: unresolved constructor call Qt::Widget
```

The output shows that `Qt::Widget` has four forms of initializer, none of which fit the syntax we were trying.

### Debug Channels

QtRuby also has a number of debug channels which can be turned on or off. These let you look into the toolkit at runtime.

- `QTDB_NONE`—No debug information
- `QTDB_AMBIGUOUS`—unused
- `QTDB_METHODMISSING`—unused
- `QTDB_CALLS`—show method call information
- `QTDB_GC`—show garbage collection information
- `QTDB_VIRTUAL`—display virtual method override information
- `QTDB_VERBOSE`—unused
• QTDB_ALL—Turn on all debug channels

These debug channels are set using:

```cpp
Qt::Internal::setDebug(Qt::QtDebugChannel::QTDB_VIRTUAL)
```
The **KDE** project is an open source Unix desktop environment. As discussed in Section 2.1, *A Little History*, on page 5, it is heavily based on **Qt**.

Many aspects of **KDE** are extensions of things available in **Qt**, so there are many **Qt** widgets that have been extended in **KDE**. **Korundum** provides a set of Ruby bindings to these additional classes. In other words, **QtRuby** is to **Qt** as **Korundum** is to **KDE**.

### 8.1 Installing Korundum

You need the **Korundum** package, available from the same location as **QtRuby**.

First, the **KDE** libraries must be installed. This can be accomplished in a very similar manner to the installation of **Qt** as described in Section 2.4, *How to install Qt from source*, on page 8. **KDE** is readily available from most Linux distributions and from Fink on Mac OS. **KDE** can be installed from source as well. To find out how to install **KDE**, we recommend the excellent documentation at [http://www.kde.org/download](http://www.kde.org/download).

Second, **SMOKE** must be configured to generate bindings for **KDE**. One of the configuration options described in Section 3.3, *Installing QtRuby*, on page 14 was `--with-smoke="qt`. This needs to be changed to `--with-smoke="qt kde"`. Alternatively, the whole option can be dropped, as building bindings for both **Qt** and **KDE** is the default behavior.

Finally, generate the **Korundum** bindings in the same way as you did for **QtRuby**.
8.2 Using Korundum

Although using the Korundum classes is as easy as using QtRuby classes, you’ll need to change your programs slightly.

1. Change require ‘Qt’ to require ‘Korundum’.

2. KDE classnames go into the KDE namespace, and the initial K is dropped. For example, the class KPopupMenu becomes KDE::PopupMenu.

3. Use KDE::Application instead of Qt::Application.

KDE::Application requires a little bit more setup than Qt::Application.

```ruby
about = KDE::AboutData.new("appname", "MyProgramName", "1.0")
KDE::CmdLineArgs.init(ARGV, about)
app = KDE::Application.new
w = KDE::DateTimeWidget.new
app.setMainWidget(w)
w.show
app.exec
```

As you can see, the KDE::Application class requires the initialization of KDE::AboutData and the KDE::CmdLineArgs classes.

KDE classes

KDE comes with a large number of classes, grouped into libraries. Some of these are:

- **kdeui**—Widgets that provide standard user interface elements.
- **kdecore**—Classes that aren't widgets, but are very useful to GUI programs.
- **kio**—A library of easy-to-use file management related classes.
• khtml—HTML rendering classes
• DCOP—A library for interprocess communication among KDE programs. We describe DCOP in more detail in the next section.

8.3 DCOP—Interprocess Communication

DCOP allows our Korundum programs to call remote methods in other Korundum (or KDE) programs.

To make a method callable by DCOP, define it in the same way we defined a regular Qt slot.

```ruby
require 'korundum'

class MyWidget < KDE::Dialog
  k_dcop 'QSize mySize()'

  def initialize(parent=nil, name=nil)
    super(parent, name)
  end

  def mySize()
    return size()
  end
end

In this case, we used the k_dcop declaration to make the mySize() method remotely callable. We can then initialize and run the widget.

```ruby
about = KDE::AboutData.new("app1", "MyApplication", "1.0")
KDE::CmdLineArgs.init(ARGV, about)
a = KDE::Application.new()
w = MyWidget.new
a.dcopClient.registerAs("app1", false)
a.setMainWidget(w)
```
In a second program, we can attempt to call this remote method.

```ruby
require 'Korundum'

about = KDE::AboutData.new("app2", "App2", "1.0")
KDE::CmdLineArgs.init(ARGV, about)
app = KDE::Application.new()

ref = KDE::DCOPRef.new("app1", "MyWidget")
res = ref.call("mySize()")

puts "W: #{res.width} H: #{res.height}"
```

```
$ ruby ex_korundum_size_remote.rb
>>>> W: 640 H: 384
```

We illustrate this example in figure Figure 8.1.
What’s going on here

What’s we’re seeing here is a lot of behind-the-scenes magic. First, applications wishing to use DCOP are considered DCOP clients and need to register with the DCOP server (which runs automatically after the KDE desktop starts). Clients can communicate with each other via message calls which get sent to the server and dispatched to the appropriate destination client. Some messages are one-way; they are sent and the client continues without waiting for a reply. Others are sent by clients who await a reply.

The KDE::Application class provides the method dcopClient() which returns a DCOPClient client connection to the server. The client can attach() to or, detach() from the server. It can also change its registration information with registerAs().

Objects within an application can use the DCOPObject object to create an interface via the DCOPClient for remote method calls. By using the k_dcop syntax, this DCOPObject interface is automatically created for us.

Finally, remote applications can use DCOPRef to create a connection to a remote DCOPObject. The DCOPRef initializer has two arguments, the name of the application to connect to (as was specified by the DCOPClient registerAs() method), and the remote object to connect to (as specified by the objId() of the DCOPObject.

The DCOPRef class can then use call() to call remote functions.

```ruby
ref = DCOPRef.new("appname","objname")
ref.call("someFunction") # objname.someFunction
ref.call("someFunction()") # objname.someFunction
ref.call("someFunction()","arg") # objname.someFunction(arg)
```
For methods that have no return value, there is the send() method.

```ruby
ref = DCOPRef.new("appname", "objname")
ref.send("someFunction", "arg")  # objname.someFunction(arg)
```

### DCOP Signals

DCOP has signals, just like Qt. We can make remote signal/slot connections in a very similar manner. Consider the following program:

```ruby
require 'Korundum'

class SignalWidget < KDE::Dialog
  k_dcop_signals 'void mySizeSignal(QSize)'
  slots 'timerSlot()'

  def initialize(parent=nil, name=nil)
    super(parent,name)
    t = Qt::Timer.new(self)
    connect(t, SIGNAL('timeout()'), self,
             SLOT('timerSlot()'))
    t.start(5000)
  end

  def timerSlot
    puts "emitting signal"
    emit mySizeSignal(size())
  end
end

about = KDE::AboutData.new("appname", "MyApplication", "1.0")
KDE::CmdLineArgs.initARGV, about)
a = KDE::Application.new()
w = SignalWidget.new
w = a.dcopClient.registerAs("appname",false)
a.setMainWidget(w)
```
w.show
a.exec

The corresponding remote application:

```ruby
require 'Korundum'

class SlotWidget < KDE::Dialog
  k_dcop 'void mySlot(QSize)'

  def initialize(parent=nil, name=nil)
    super(parent,name)
  end

  def mySlot(size)
    puts "mySlot called #{size}"
    dispose
  end
end

about = KDE::AboutData.new("remote", "Remote", "1.0")
KDE::CmdLineArgs.init(ARGV, about)
a = KDE::Application.new()
w = SlotWidget.new()
w.connectDCOPSignal("appname","SignalWidget",
  "mySizeSignal(QSize)", "mySlot(QSize)", false)
a.setMainWidget(w)
a.exec
```

In executing these two programs, the follow logic occurs:

1. We create the SignalWidget. Every five seconds its internal timer calls its slot `timerSlot()`, which emits the DCOP signal `mySizeSignal()`. 

2. We create the SlotWidget in a separate application. We a DCOP
slot called mySlot() and connect SignalWidget’s signal to this slot.

3. We run the program and watch as every five seconds the DCOP signal and slots get activated.

### 8.4 Summary

- Applications using KDE classes must use KDE::Application instead of Qt::Application.

- DCOP provides an interface for calling remote methods in running applications.
# Event Method Map

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Event Class</th>
<th>Event Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mouse</td>
<td>Qt::MouseEvent</td>
<td>mousePressEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mouseDoubleClickEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mouseMoveEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mouseReleaseEvent</td>
</tr>
<tr>
<td></td>
<td>Qt::WheelEvent</td>
<td>wheelEvent</td>
</tr>
<tr>
<td></td>
<td>Qt::Event</td>
<td>enterEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>leaveEvent</td>
</tr>
<tr>
<td>Keyboard</td>
<td>Qt::KeyEvent</td>
<td>keyPressEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>keyReleaseEvent</td>
</tr>
<tr>
<td>Tablet</td>
<td>Qt::TabletEvent</td>
<td>tabletEvent</td>
</tr>
<tr>
<td>Input Method</td>
<td>Qt::KeyEvent</td>
<td>imStartEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>imComposeEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>imEndEvent</td>
</tr>
<tr>
<td>Drag and Drop</td>
<td>Qt::DragEnterEvent</td>
<td>dragEnterEvent</td>
</tr>
<tr>
<td></td>
<td>Qt::DragMoveEvent</td>
<td>dragMoveEvent</td>
</tr>
<tr>
<td></td>
<td>Qt::DragLeaveEvent</td>
<td>dragLeaveEvent</td>
</tr>
<tr>
<td></td>
<td>Qt::DropEvent</td>
<td>dropEvent</td>
</tr>
<tr>
<td>Drawing</td>
<td>Qt::ShowEvent</td>
<td>showEvent</td>
</tr>
<tr>
<td></td>
<td>Qt::HideEvent</td>
<td>hideEvent</td>
</tr>
<tr>
<td></td>
<td>Qt::PaintEvent</td>
<td>paintEvent</td>
</tr>
<tr>
<td></td>
<td>Qt::ResizeEvent</td>
<td>resizeEvent</td>
</tr>
<tr>
<td></td>
<td>Qt::CloseEvent</td>
<td>closeEvent</td>
</tr>
<tr>
<td>Tablet</td>
<td>Qt::TabletEvent</td>
<td>tabletEvent</td>
</tr>
<tr>
<td>Input Method</td>
<td>Qt::ObjectEvents</td>
<td>Qt::TimerEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>timerEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Qt::ChildEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>childEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Qt::CustomEvent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>customEvent</td>
</tr>
</tbody>
</table>
Appendix B

Resources

B.1 Web Resources

**Qt**
Trolltech’s homepage

**KDE**
KDE homepage

**KDE Ruby Bindings Homepage**
Brief introduction to QtRuby and Korundum

**Qt Documentation**
Online documentation for Qt and utilities

**QtRuby Online Tutorial**
Learn Qt and QtRuby online with this 14 step tutorial

**QtRuby/Korundum at RubyForge**
Rubyforge download site for QtRuby

**C++ GUI Programming with Qt 3**
Link to book website, with freely downloadable PDF.
B.2 Bibliography


Pragmatic Fridays
Timely and focused PDF-only books. Written by experts for people who need information in a hurry. No DRM restrictions. Free updates. Immediate download. Visit our web site to see what’s happening on Friday!

More Online Goodness

QtRuby
Source code from this book and other resources. Come give us feedback, too!

Free Updates
Visit the link, identify your book, and we’ll create a new PDF containing the latest content.

Errata and Suggestions
See suggestions and known problems. Add your own. (The easiest way to report an errata is to click on the link at the bottom of the page.

Join the Community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
Check out the latest pragmatic developments in the news.

Contact Us
Phone Orders: 1-800-699-PROG (+1 919 847 3884)
Online Orders: www.pragmaticprogrammer.com/catalog
Customer Service: orders@pragmaticprogrammer.com
Non-English Versions: translations@pragmaticprogrammer.com
Pragmatic Teaching: academic@pragmaticprogrammer.com
Author Proposals: proposals@pragmaticprogrammer.com