Community Experience Distilled

Qt 5 Blueprints

Design, build, and deploy cross-platform GUI projects using the amazingly powerful Qt 5 framework

Symeon Huang

[PACKT] PUBLISHING open source* community experience distilled

Qt 5 Blueprints

Design, build, and deploy cross-platform GUI projects using the amazingly powerful Qt 5 framework

Symeon Huang

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Qt 5 Blueprints

First published: March 2015

Production reference: 1240315

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-461-5

www.packtpub.com

Credits

Author
Symeon Huang

Reviewers
Lee Zhi Eng
Sudhendu Kumar
Mickael Minarie

Acquisition Editor
Shaon Basu

Content Development Editor
Sriram Neelakantan

Technical Editors
Novina Kewalramani
Shruti Rawool

Copy Editor
Sonia Michelle Cheema

Project Coordinator
Judie Jose

Proofreaders
Simran Bhogal
Safis Editing

Indexer
Rekha Nair

Graphics
Sheetal Aute
Disha Haria
Abhinash Sahu

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

About the Author

Symeon Huang is an amateur developer who's currently doing his master's degree at Trinity College, Dublin. He has been contributing to open source projects for several years. He has worked in various areas, including the maintenance of Linux servers, desktop application development, and image recognition and analysis.

Symeon has always been passionate about cool technology and elegant programming techniques. He has been programming Qt and QML applications for 2 years and has also been developing pure C and C++ programs for many years. Most of the projects he's working on can be found on his GitHub and Gitorious pages.

I would like to thank my family, especially my parents, for supporting me all through this process. I would never have been able to achieve what I have today without their hard work and unconditional love.

I would also like to thank my mentor, Ting Dai, from Southeast University, China, for his teaching. Without the things I have learned from him, I wouldn't have started programming in C++ with Qt. He also taught me a lot about common software development and gave me helpful programming tips.

About the Reviewers

Lee Zhi Eng is a 3D artist-turned-programmer who worked as a game artist and programmer in several local game studios in his native country before becoming a contractor and a part-time lecturer at a local university and teaching game development subjects, particularly those related to Unity Engine and Unreal Engine 4. You can find more information about him at `http://www.zhieng.com`.

Sudhendu Kumar has been a GNU/Linux user for more than 7 years. Currently, he is a software developer for a networking giant, and in his free time, he also contributes to KDE.

I would like to thank the publishers for giving me the opportunity to review this book. I hope readers find it useful and enjoy reading it and playing around with Qt/Qml applications, not only on desktop devices but also on mobile platforms.

Mickael Minarie is a software developer who graduated from the University of Clermont-Ferrand (bachelor's in embedded systems) and Robert Gordon University, Aberdeen (bachelor's in computer science). He has worked on freelance projects, developing some programs in C++/Qt for embedded systems or in programs linked with photos and videos.

He now lives in France, but he has lived in the UK and Canada for some years.

He is an analog photography and video enthusiast and has written articles for photography fanzines.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Qt has been developed as a cross-platform framework and has been provided free to the public for years. It's mainly used to build GUI applications. It also provides thousands of APIs for easier development.

Qt 5, the latest major version of Qt, has once again proven to be the most popular cross-platform toolkit. With all these platform-independent classes and functions, you only need to code once, and then you can make it run everywhere.

In addition to the traditional and powerful C++, Qt Quick 2, which is more mature, can help web developers to develop dynamic and reliable applications, since QML is very similar to JavaScript.

What this book covers

Chapter 1, *Creating Your First Qt Application*, takes you through the fundamental concepts of Qt, such as signals and slots, and helps you create your first Qt and Qt Quick applications.

Chapter 2, *Building a Beautiful Cross-platform Clock*, teaches you how to read and write configurations and handle cross-platform development.

Chapter 3, *Cooking an RSS Reader with Qt Quick*, demonstrates how to develop a stylish RSS Reader in QML, which is a script language quite similar to JavaScript.

Chapter 4, *Controlling Camera and Taking Photos*, shows you how to access camera devices through the Qt APIs and make use of the status and menu bars.

Chapter 5, *Extending Paint Applications with Plugins*, teaches you how to make applications extendable and write plugins, by using the Paint application as as an example.

Chapter 6, *Getting Wired and Managing Downloads*, shows you how to utilize Qt's network module using the progress bar, as well as learning about threaded programming in Qt.

Chapter 7, *Parsing JSON and XML Documents to Use Online APIs*, teaches you how to parse JSON and XML documents in both Qt/C++ and Qt Quick/QML, which is essential to obtain data from online APIs.

Chapter 8, *Enabling Your Qt Application to Support Other Languages*, demonstrates how to make internationalized applications, translate strings using Qt Linguist, and then load translation files dynamically.

Chapter 9, *Deploying Applications on Other Devices*, shows you how to package and make your applications redistributable on Windows, Linux, and Android.

Chapter 10, *Don't Panic When You Encounter These Issues*, gives you some solutions and advice for common issues during Qt and Qt Quick application development and shows you how to debug Qt and Qt Quick applications.

What you need for this book

Qt is cross-platform, which means you can use it on almost all operating systems, including Windows, Linux, BSD, and Mac OS X. The hardware requirements are listed as follows:

- A computer (PC or Macintosh)
- A webcam or a connected camera device
- Available Internet connection

An Android phone or tablet is not required, but is recommended so that you can test applications on a real Android device.

All the software mentioned in this book, including Qt itself, is free of charge and can be downloaded from the Internet.

Who this book is for

If you are a programmer looking for a truly cross-platform GUI framework to help you save time by side-stepping issues involving incompatibility between different platforms and building applications using Qt 5 for multiple targets, this book is most certainly intended for you. It is assumed that you have basic programming experience of C++.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The UI files are under the `Forms` directory."

A block of code is set as follows:

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
#include <QStyleOption>
#include <QPainter>
#include <QPaintEvent>
#include <QMouseEvent>
#include <QResizeEvent>
#include "canvas.h"

Canvas::Canvas(QWidget *parent) :
  QWidget(parent)
{
}

void Canvas::paintEvent(QPaintEvent *e)
{
  QPainter painter(this);

  QStyleOption opt;
  opt.initFrom(this);
```

```
    this->style()->drawPrimitive(QStyle::PE_Widget, &opt, &painter,
      this);

    painter.drawImage(e->rect().topLeft(), image);
}

void Canvas::updateImage()
{
    QPainter painter(&image);
    painter.setPen(QColor(Qt::black));
    painter.setRenderHint(QPainter::Antialiasing);
    painter.drawPolyline(m_points.data(), m_points.count());
    this->update();
}

void Canvas::mousePressEvent(QMouseEvent *e)
{
    m_points.clear();
    m_points.append(e->localPos());
    updateImage();
}

void Canvas::mouseMoveEvent(QMouseEvent *e)
{
    m_points.append(e->localPos());
    updateImage();
}

void Canvas::mouseReleaseEvent(QMouseEvent *e)
{
    m_points.append(e->localPos());
    updateImage();
}

void Canvas::resizeEvent(QResizeEvent *e)
{
    QImage newImage(e->size(), QImage::Format_RGB32);
    newImage.fill(Qt::white);
    QPainter painter(&newImage);
    painter.drawImage(0, 0, image);
    image = newImage;
    QWidget::resizeEvent(e);
}
```

Any command-line input or output is written as follows:

```
..\..\bin\binarycreator.exe -c config\config.xml -p packages
internationalization_installer.exe
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: " Navigate to **File** | **New File** or **Project**."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Creating Your First Qt Application

GUI programming is not as difficult as you think. At least it's not when you come to the world of Qt. This book will take you through this world and give you an insight into this incredibly amazing toolkit. It doesn't matter whether you've heard of it or not, as long as you have essential knowledge of C++ programming.

In this chapter, we will get you comfortable with the development of Qt applications. Simple applications are used as a demonstration for you to cover the following topics:

- Creating a new project
- Changing the layout of widgets
- Understanding the mechanism of signals and slots
- Connecting two signals
- Creating a Qt Quick application
- Connecting C++ slots to QML signals

Creating a new project

If you haven't installed Qt 5, refer to `http://www.qt.io/download` to install the latest version of it. It's recommended that you install the Community version, which is totally free and compliant with GPL/LGPL. Typically, the installer will install both **Qt Library** and **Qt Creator** for you. In this book, we will use Qt 5.4.0 and Qt Creator 3.3.0. Later versions may have slight differences but the concept remains the same. It's highly recommended that you install Qt Creator if you don't have it on your computer, because all the tutorials in this book are based on it. It is also the official IDE for the development of Qt applications. Although you may be able to develop Qt applications with other IDEs, it tends to be much more complex. So if you're ready, let's go for it by performing the following steps:

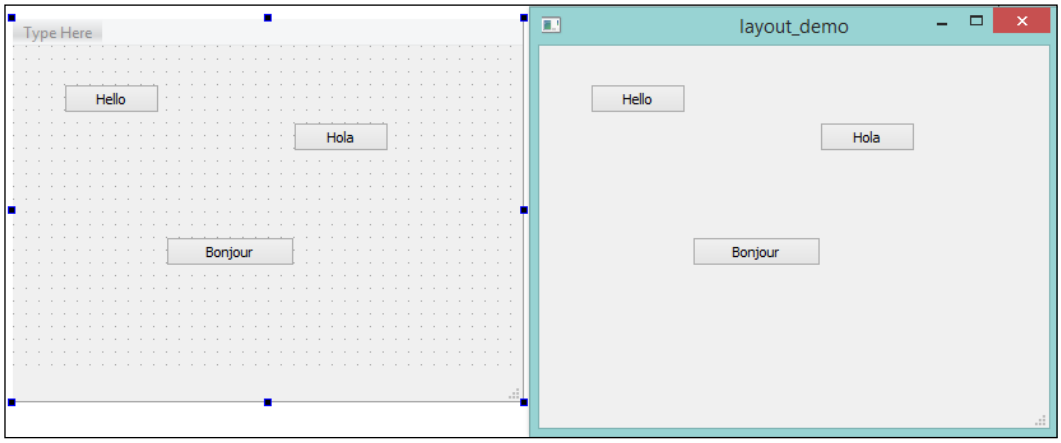
1. Open Qt Creator.
2. Navigate to **File** | **New File** or **Project**.
3. Select **Qt Widgets Application**.
4. Enter the project's name and location. In this case, the project's name is `layout_demo`.

You may wish to follow the wizard and keep the default values. After this process, Qt Creator will generate the skeleton of the project based on your choices. The UI files are under the `Forms` directory. When you double-click on a UI file, Qt Creator will redirect you to the integrated designer. The mode selector should have **Design** highlighted, and the main window should contain several sub-windows to let you design the user interface. This is exactly what we are going to do. For more details about Qt Creator UI, refer to `http://doc.qt.io/qtcreator/creator-quick-tour.html`.

Drag three push buttons from the widget box (widget palette) into the frame of **MainWindow** in the center. The default text displayed on these buttons is **PushButton**, but you can change the text if you want by double-clicking on the button. In this case, I changed the buttons to `Hello`, `Hola`, and `Bonjour`, accordingly. Note that this operation won't affect the `objectName` property. In order to keep it neat and easy to find, we need to change the `objectName` property. The right-hand side of the UI contains two windows. The upper-right section includes **Object Inspector** and the lower-right side includes **Property Editor**. Just select a push button; you can easily change `objectName` in **Property Editor**. For the sake of convenience, I changed these buttons' `objectName` properties to `helloButton`, `holaButton`, and `bonjourButton` respectively.

It's a good habit to use lowercase for the first letter of `objectName` and an uppercase letter for **Class name**. This helps your code to be more readable by people who are familiar with this convention.

Okay, it's time to see what you have done to the user interface of your first Qt application. Click on **Run** on the left-hand side panel. It will build the project automatically and then run it. It's amazing to see that the application has the exact same interface as the design, isn't it? If everything is alright, the application should appear similar to what is shown in the following screenshot:



You may want to look at the source code and see what happened there. So, let's go back to the source code by returning to the **Edit** mode. Click on the **Edit** button in the mode selector. Then, double-click on `main.cpp` in the `Sources` folder of the **Projects** tree view. The code for `main.cpp` is shown as follows:

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

The `QApplication` class manages the GUI application's control flow and the main settings.

Actually, you don't need to and you probably won't change too much in this file. The first line of the main scope just initializes the applications on a user's desktop and handles some events. Then there is also an object, `w`, which belongs to the `MainWindow` class. As for the last line, it ensures that the application won't terminate after execution but will keep in an event loop, so that it is able to respond to external events such as mouse clicks and window state changes.

Last but not least, let's see what happens during the initialization of the `MainWindow` object, `w`. It is the content of `mainwindow.h`, shown as follows:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;
};

#endif // MAINWINDOW_H
```

You may feel a bit surprised seeing a `Q_OBJECT` macro if this is your first time writing a Qt application. In the QObject documentation, it says:

> *The* `Q_OBJECT` *macro must appear in the private section of a class definition that declares its own signals and slots or that uses other services provided by Qt's meta-object system.*

Well, this means that `QObject` has to be declared if you're going to use Qt's meta-object system and (or) its signals and slots mechanism. The signals and slots, which are almost the core of Qt, will be included later in this chapter.

There is a private member named `ui`, which is a pointer of the `MainWindow` class of the `Ui` namespace. Do you remember the UI file we edited before? What the magic of Qt does is that it links the UI file and the parental source code. We can manipulate the UI through code lines as well as design it in Qt Creator's integrated designer. Finally, let's look into the construction function of `MainWindow` in `mainwindow.cpp`:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

Did you see where the user interface comes from? It's the member `setupUi` function of `Ui::MainWindow` that initializes it and sets it up for us. You may want to check what happens if we change the member function to something like this:

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    ui->holaButton->setEnabled(false);
}
```

What happened here? The `Hola` button can't be clicked on because we disabled it! It has the same effect if the **enabled** box is unchecked in the designer instead of writing a statement here. Please apply this change before heading to the next topic, because we don't need a disabled push button to do any demonstrations in this chapter.

Changing the layout of widgets

You already know how to add and move widgets in the **Design** mode. Now, we need to make the UI neat and tidy. I'll show you how to do this step by step.

A quick way to delete a widget is to select it and press the **Delete** button. Meanwhile, some widgets, such as the menu bar, status bar, and toolbar can't be selected, so we have to right-click on them in **Object Inspector** and delete them. Since they are useless in this example, it's safe to remove them and we can do this for good.

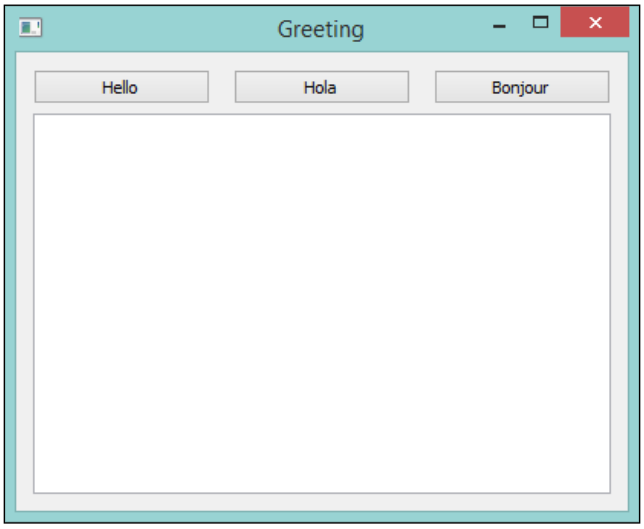
Okay, let's understand what needs to be done after the removal. You may want to keep all these push buttons on the same horizontal axis. To do this, perform the following steps:

1. Select all the push buttons either by clicking on them one by one while keeping the *Ctrl* key pressed or just drawing an enclosing rectangle containing all the buttons.
2. Right-click and select **Layout** | **LayOut Horizontally**, The keyboard shortcut for this is *Ctrl* + *H*.
3. Resize the horizontal layout and adjust its `layoutSpacing` by selecting it and dragging any of the points around the selection box until it fits best.

Hmm…! You may have noticed that the text of the **Bonjour** button is longer than the other two buttons, and it should be wider than the others. How do you do this? You can change the property of the horizontal layout object's `layoutStretch` property in **Property Editor**. This value indicates the stretch factors of the widgets inside the horizontal layout. They would be laid out in proportion. Change it to `3,3,4`, and there you are. The stretched size definitely won't be smaller than the minimum size hint. This is how the zero factor works when there is a nonzero natural number, which means that you need to keep the minimum size instead of getting an error with a zero divisor.

Now, drag **Plain Text Edit** just below, and not inside, the horizontal layout. Obviously, it would be neater if we could extend the plain text edit's width. However, we don't have to do this manually. In fact, we could change the layout of the parent, **MainWindow**. That's it! Right-click on **MainWindow**, and then navigate to **Lay out** | **Lay Out Vertically**. Wow! All the children widgets are automatically extended to the inner boundary of **MainWindow**; they are kept in a vertical order. You'll also find **Layout** settings in the `centralWidget` property, which is exactly the same thing as the previous horizontal layout.

The last thing to make this application halfway decent is to change the title of the window. `MainWindow` is not the title you want, right? Click on **MainWindow** in the object tree. Then, scroll down its properties to find **windowTitle**. Name it whatever you want. In this example, I changed it to `Greeting`. Now, run the application again and you will see it looks like what is shown in the following screenshot:

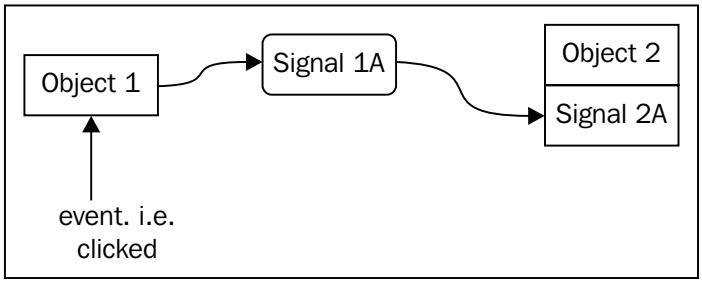


Understanding the mechanism of signals and slots

It is really important to keep your curiosity and to explore what on earth these properties do. However, please remember to revert the changes you made to the app, as we are about to enter the core part of Qt, that is, signals and slots.

Signals and slots are used for communication between objects. The signals and slots mechanism is a central feature of Qt and probably the part that differs the most from the features provided by other frameworks.

Have you ever wondered why a window closes after the **Close** button is clicked on? Developers who are familiar with other toolkits would say that the **Close** button being clicked on is an event, and this event is bound with a callback function that is responsible for closing the window. Well, it's not quite the same in the world of Qt. Since Qt uses a mechanism called signals and slots, it makes the callback function weakly coupled to the event. Also, we usually use the terms signal and slot in Qt. A signal is emitted when a particular event occurs. A slot is a function that is called in response to a particular signal. The following simple and schematic diagram helps you understand the relation between signals, events, and slots:



Qt has tons of predefined signals and slots, which cover its general purposes. However, it's indeed commonplace to add your own slots to handle the target signals. You may also be interested in subclassing widgets and writing your own signals, which will be covered later. The mechanism of signals and slots was designed to be type-safe because of its requirement of the list of the same arguments. In fact, the slot may have a shorter arguments list than the signal since it can ignore the extras. You can have as many arguments as you want. This enables you to forget about the wildcard `void*` type in C and other toolkits.

Since Qt 5, this mechanism is even safer because we can use a new syntax of signals and slots to deal with the connections. A conversion of a piece of code is demonstrated here. Let's see what a typical connect statement in old style is:

```
connect(sender, SIGNAL(textChanged(QString)), receiver,
  SLOT(updateText(QString)));
```

This can be rewritten in a new syntax style:

```
connect(sender, &Sender::textChanged, receiver,
  &Receiver::updateText);
```

In the traditional way of writing code, the verification of signals and slots only happens at runtime. In the new style, the compiler can detect the mismatches in the types of arguments and the existence of signals and slots at compile time.

As long as it is possible, all `connect` statements are written in the new syntax style in this book.

Now, let's get back to our application. I'll show you how to display some words in a plain text edit when the **Hello** button is clicked on. First of all, we need to create a slot since Qt has already predefined the clicked signal for the `QPushButton` class. Edit `mainwindow.h` and add a slot declaration:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:
    void displayHello();

private:
    Ui::MainWindow *ui;
};

#endif // MAINWINDOW_H
```

As you can see, it's the `slots` keyword that distinguishes slots from ordinary functions. I declared it private to restrict access permission. You have to declare it a `public` slot if you need to invoke it in an object from other classes. After this declaration, we have to implement it in the `mainwindow.cpp` file. The implementation of the `displayHello` slot is written as follows:

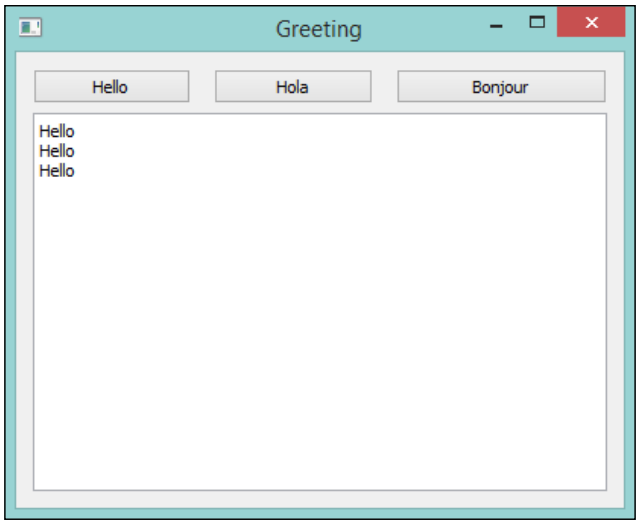
```
void MainWindow::displayHello()
{
    ui->plainTextEdit->appendPlainText(QString("Hello"));
}
```

It simply calls a member function of the plain text edit in order to add a `Hello` QString to it. `QString` is a core class that Qt has introduced. It provides a Unicode character string, which efficiently solves the internationalization issue. It's also convenient to convert a `QString` class to `std::string` and vice versa. Besides, just like the other `QObject` classes, `QString` uses an implicit sharing mechanism to reduce memory usage and avoid needless copying. If you don't want to get concerned about the scenes shown in the following code, just take `QString` as an improved version of `std::string`. Now, we need to connect this slot to the signal that the **Hello** push button will emit:

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    connect(ui->helloButton, &QPushButton::clicked, this,
        &MainWindow::displayHello);
}
```

What I did is add a `connect` statement to the constructor of `MainWindow`. In fact, we can connect signals and slots anywhere and at any time. However, the connection only exists after this line gets executed. So, it's a common practice to have lots of `connect` statements in the construction functions instead of spreading them out. For a better understanding, run your application and see what happens when you click on the **Hello** button. Every time you click, a **Hello** text will be appended to the plain text edit. The following screenshot is what happened after we clicked on the **Hello** button three times:



Getting confused? Let me walk you through this. When you clicked on the **Hello** button, it emitted a clicked signal. Then, the code inside the `displayHello` slot got executed, because we connected the clicked signal of the **Hello** button to the `displayHello` slot of `MainWindow`. What the `displayHello` slot did is that it simply appended `Hello` to the plain text edit.

It may take you some time to fully understand the mechanism of signals and slots. Just take your time. I'll show you another example of how to disconnect such a connection after we clicked on the **Hola** button. Similarly, add a declaration of the slot to the header file and define it in the source file. I pasted the content of the `mainwindow.h` header file, as follows:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:
    void displayHello();
    void onHolaClicked();

private:
    Ui::MainWindow *ui;
};

#endif // MAINWINDOW_H
```

It's only declaring a `onHolaClicked` slot that differed from the original. Here's the content of the source file:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
```