Code less.
Create more.
Deploy everywhere.

Qt Quarterly

C++ and Qt Programmers' Newsletter

Issue 31 · Q3 2009

Contents

Qt Quarterly Goes HTML

This is the last time Qt Quarterly comes in the familiar magazine format, as a PDF file formatted for printing. From now on, articles will be put directly on the Web for the enjoyment of the world at large.

The first issue of Qt Quarterly was started in December 2001 and published in February 2002. Just as with classic films, issues one and two were only available in black and white. From issue three onwards, customers around the world were able to enjoy their copies of a full color newsletter, though it took until issue five until the standard twelve page format appeared. It was only with issue eleven that the familiar portrait photos of the authors were introduced.

One of the advantages of a carefully typeset and printed publication is the level of control the editors have over the appearance of the finished articles. The flip side of this, however, is the onerous requirement to cut down articles to fit a certain length, which has often led to marathon editing sessions. We hope that readers have appreciated the attention to detail of the editors over the years, as they have struggled to squeeze six-page articles down to three pages, while still leaving enough space for a photograph of the author and some amusing biographical information.

In the future, by eliminating the restrictions on article sizes, we hope that we can include a more diverse ranges of articles in Qt Quarterly, and let some authors to go into more depth than the old format could allow.

We in the Qt Documentation Team have had a lot of fun with Qt Quarterly—as readers, writers and editors. Qt Quarterly style articles will still be written, but I think we will always look back fondly on the crisp, colorful, paper magazines.

Qt Training

Are you new to Qt? Or do you need help to get your Qt project started?

Our Qt Training Partners have a broad geographic reach and can help you get up to speed with Qt faster and more efficiently. They offer Qt Training via open enrollment courses or through on-site courses that can be adapted to suit your needs.

For more information about Qt Training and other learning solutions, please visit http://qt.nokia.com/training and http://qt.nokia.com/learning.

Matching Parentheses with QSyntaxHighlighter

An often forgotten and neglected feature of QSyntaxHighlighter is the ability to attach user data to the text blocks being highlighted. Among the many possibilities that this opens up for advanced text editing, we find parenthesis matching—which no self-respecting editor can do without.

Although the main intention behind this article is to shed some light on some of `QSyntaxHighlighter`'s more obscure capabilities, and do something useful with them, we will also see how to use these add extra selections to a `QPlainTextEdit`.

Overview

To implement the parenthesis matching, a few classes are needed. First, we need a subclass of `QSyntaxHighlighter`. This subclass will register the position of each parenthesis and store that information in the text blocks. Then we inherit `QPlainTextEdit` to create an editor that can fetch the parenthesis information from the text blocks and highlight matching parentheses in a green color. We also have two helper classes for the text edit and highlighter.

```
; http://en.wikipedia.org/wiki/Scheme_(programming_language)
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

Before we embark on a code tour, let's take a quick look at the underlying Qt architecture. A `QTextDocument` consists of a series of `QTextBlock`s, which are containers for fragments of the text in the document. Qt allows you to store custom data in each text block by subclassing `QTextBlockUserData` for use with `QTextBlock`. The document can then be rendered in a widget such as `QTextEdit` or `QPlainTextEdit`, which have access to the text blocks and user data.

`QSyntaxHighlighter` gets access to visible text blocks before they are rendered, and can therefore store the user data. After that minimal description of the Qt text handling system, we are ready to look more closely at the classes needed for this experiment.

The Highlighter

For this article, we have implemented the `Highlighter` class, which inherits `QSyntaxHighlighter`. Let's take a look at its definition:

```
class Highlighter : public QSyntaxHighlighter
{
    Q_OBJECT

public:
    Highlighter(QTextDocument *document);

protected:
    void highlightBlock(const QString &text);
};
```

The `highlightBlock()` function is called with the text of the block that should be highlighted. Access to the block itself is given through functions in `QSyntaxHighlighter`. For instance, the `setFormat()` family, let you set the `QTextCharFormat` of parts of the block, and `setCurrentBlockUserData()` sets user data on the text block and is the function that helps us here.

We introduce two new types to support the highlighter and text edit: `TextBlockData`, a class which inherits `QTextBlockUserData`, and `ParenthesisInfo`, a struct which keeps track of the position and direction of a parenthesis. Here are their definitions:

```
struct ParenthesisInfo
{
    char character;
    int position;
};

class TextBlockData : public QTextBlockUserData
{
public:
    TextBlockData();

    QVector<ParenthesisInfo *> parentheses();
    void insert(ParenthesisInfo *info);

private:
    QVector<ParenthesisInfo *> m_parentheses;
};
```

We won't look at the implementation of `TextBlockData`. The only thing of importance to know here is that the list of parenthesis information is sorted by position in the text block.

Let's look at how `Highlighter` collects and stores the parenthesis information. This is done in `highlightBlock()`:

```
void Highlighter::highlightBlock(const QString &text)
{
    TextBlockData *data = new TextBlockData;

    int leftPos = text.indexOf('(');
    while (leftPos != -1) {
        ParenthesisInfo *info = new ParenthesisInfo;
        info->character = '(';
        info->position = leftPos;

        data->insert(info);
        leftPos = text.indexOf('(', leftPos + 1);
    }
    ...
    setCurrentBlockUserData(data);
}
```

First, we create an instance of `TextBlockData`, which we will fill with the parenthesis information and give to the text block we are highlighting. To find the parentheses in the block, we simply employ a linear search in `text`. For each parenthesis we find, we construct a `ParenthesisInfo` and add it to our user data.

The approach here is a bit inadequate for a real-world application. For instance, it does not catch parentheses that are inside literal strings, but for this example, it will suffice.

The Editor

We subclass `QPlainTextEdit` to create an editor that can highlight matching parentheses based on the user data from the highlighter. When it comes to editing code, we prefer `QPlainTextEdit` over `QTextEdit` because it is optimized to handle plain text; although, we could subclass `QTextEdit` instead without having to change any code. Also, in a `QPlainTextEdit`, each line will consist of one `QTextBlock`. While not of importance in this case, it makes other functionality, such as line numbers, easier to implement.

If the cursor is on a parenthesis, the time has come to search for its partner—or stop if it cannot find one. This check is done each time the position of the cursor changes. It is fortunate for us that `QPlainTextEdit` provides the `cursorPositionChanged()` signal, which we connect to `matchParentheses()`.

```
void TextEdit::matchParentheses()
{
    QList<QTextEdit::ExtraSelection> selections;
    setExtraSelections(selections);

    TextBlockData *data =
        static_cast<TextBlockData *>(
            textCursor().block().userData());
```

Matching parentheses are highlighted with an extra selection, so when the cursor position has changed, previous selections must be cleared. The user data from the `Highlighter` can then be fetched from the text block containing the cursor.

```
    if (data) {
        QVector<ParenthesisInfo *> infos =
            data->parentheses();

        int pos = textCursor().block().position();
        for (int i = 0; i < infos.size(); ++i) {
            ParenthesisInfo *info = infos.at(i);

            int curPos = textCursor().position() -
                textCursor().block().position();
            if (info->position == curPos - 1 &&
                info->character == '(') {
                matchLeftParenthesis(
                    textCursor().block(), i + 1, 0);
                return;
            }
            if (info->position == curPos - 1 &&
                info->character == ')') {
                matchRightParenthesis(
                    textCursor().block(), i - 1, 0);
            }
        }
    } ...
```

For each parenthesis present in the text block, we check whether the cursor is just after it. If so, the time has come to search for a match. This is done in `matchLeftParenthesis()` or `matchRightParenthesis()`, depending on the kind of parenthesis we are searching for. Since space is an issue, that search is left out of this article, but the idea is simply to do a linear search through the remaining parentheses in the document. You can pick up the code from the Qt Quarterly web page if you want to take a look at it.

If a matching parenthesis is found, we highlight it with an extra selection. Such selections are a feature of `QPlainTextEdit` where one can programatically add selections similar to the one the user makes with the mouse or keyboard.

```
void TextEdit::createParenthesisSelection(int pos)
{
    QList<QTextEdit::ExtraSelection> selections;

    QTextEdit::ExtraSelection selection;
    selection.format.setBackground(Qt::green);

    QTextCursor cursor = textCursor();
    cursor.setPosition(pos);
    cursor.movePosition(QTextCursor::NextCharacter,
                        QTextCursor::KeepAnchor);
    selection.cursor = cursor;

    selections.append(selection);

    setExtraSelections(selections);
}
```

Beyond Parenthesis Matching

We have seen how `QSyntaxHighlighter` can be used for other purposes than just syntax highlighting. Specifically, we looked at its ability to attach user data to the text blocks it highlights. If we venture beyond the matching of parentheses, we can discover other uses for this ability. Usually, we want to help the renderer of the text document with on-the-fly information about individual text blocks. For instance, an editor that does spell checking might want to underline typos with wavy red lines.

Geir Vattekar *is a Technical Writer at Qt Development Frameworks. Writing aside, he enjoys interactive fiction, functional programming, and machine learning.*

Qt Creator Revisited

Last year we started to cover the preview release of Qt Creator. In the meantime, the young IDE saw three releases and now the developers are working full steam ahead towards version 1.3, a preview for which has recently been released. In this article, we are going to revisit Qt Creator and look at the new features in the upcoming release.

When you start a recent version of Qt Creator, the first thing you will notice is the reworked welcome screen. It provides three tabs, each for a slightly different use case. At first start up, you will be presented with the Getting started tab, providing literally all you need to get going with Qt Creator: a quick tour, a full blown tutorial dedicated to writing a Qt application with Creator, as well as access to all Qt examples and useful tips and tricks.

For daily use, switch to the Develop tab, which provides convenient access to recently opened projects and sessions. If you want to stay up-to-date with the latest Qt news from Qt Labs or get in touch with the Qt community, check the Community tab. On restart, Qt Creator will jump to the tab which was opened before.

Windows Love

While it was perfectly possible to use Qt Creator with Microsoft's Visual C++ Compiler from day one, developers that had to rely on this compiler for their daily work did not have an integrated debugger like their colleagues who used MinGW.

Due to differences in the binary format, GDB is not capable of debugging Visual Studio binaries. To rectify this, the development team in Berlin added support for CDB, part of the Microsoft Debugging Tools. The introspection of native Qt types, provided by the so-called debugging helpers, is available for both debuggers.

This means that you can develop Qt applications on Windows using the freely available Windows SDK, CDB and Qt Creator without the need for Visual Studio. Unfortunately, there is no prebuilt GPL package for Qt which uses the Visual Studio Compiler. However, work is under way to provide such a package. In the meantime, simply build Qt yourself by running the configure command from the Visual Studio or Windows SDK Command Prompt.

Another bonus for Windows developers will be the integration of `jom` (http://qt.gitorious.org/qt-labs/jom) into Qt Creator to support parallel building. This is because Microsoft's `nmake` tool, in contrast to GNU `make` and other `make` implementations, lacks the `-j` option, which enables multiple instances of `make` to be run. `jom`, which started out as a Creative Friday project, solves this problem. It basically reimplements `nmake`, but does not only add support for the `-j` option—it also automatically adjusts the number of processes to the number of processor cores in your machine. While `jom` can already be used as an `nmake` replacement in all versions, Qt Creator 1.3 will ship with `jom` and automatically use it instead of `nmake`.

In Foreign Lands

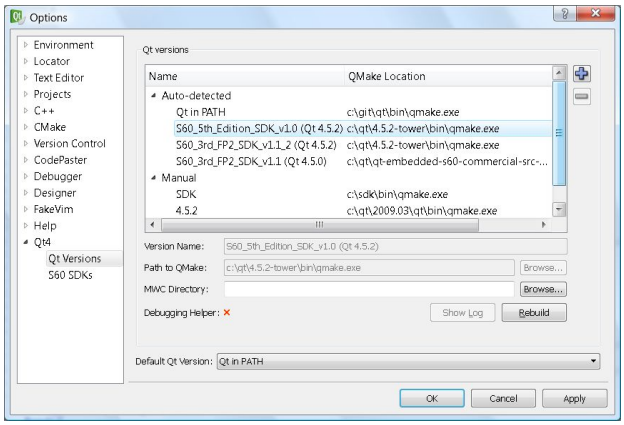
While `qmake` is still the best-supported build system, CMake is becoming increasingly popular, and even the KDE project has switched to it. Qt Creator supports CMake based projects on all supported platforms, be it Linux, Mac OS X or Windows. Just open your `CMakeLists.txt` file to import an existing project. The CMake support in Qt Creator 1.3 will work work best with the about-to-be-released CMake 2.8, which allows Creator to build CMake projects with the Microsoft Visual Studio toolchain and allows seamless editing of all CMake project files. However, visual project management is not implemented for CMake.

Developers who use neither CMake nor `qmake` for development, but base their work on Makefiles or project generators that produce Makefiles, can now use the generic project support. Vi lovers will appreciate the Vim editing mode in Qt Creator. Originating from a Creative Friday project at Qt Development Frameworks, it has been receiving lots of patches by external contributors from the community.

Finally, work has been done on adding more revision control systems. Those who store their projects in CVS will be happy to find their revision control system supported by the new Creator release. At the same time, the Subversion and Git plugins have seen improvements.

Symbian Love

After the port of Qt to the Symbian platform, it was only a matter of time before Qt Creator came into play. Starting with 1.3, Creator gets preliminary support for Qt on the Symbian platform. By preliminary, we mean that it is not ready for production use yet and we will improve this feature step by step. To ensure we are heading in the right direction, we depend on your feedback, so give it a try. Remember to install the S60 SDK and Qt for Symbian in order to make it work.

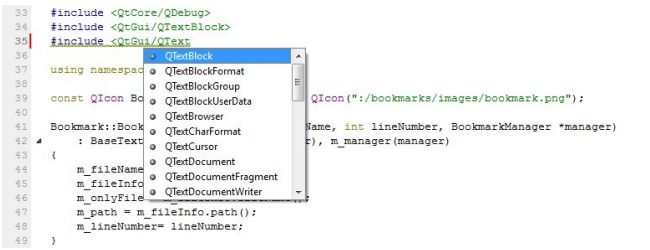


More Sweets and Candy Ahead

Qt Creator 1.3 will bring even more improvements. For example, the debugger allows for the introspection of raw memory using the same binary editor that is used to display binary files.

To maintain a free alternative to the Visual C++ tool chain, the MinGW tool chain has been upgraded to use GCC 4.4, an update long overdue which was made possible by the MinGW team releasing a stable version of GCC newer than GCC 3.4. GCC 4.4 is a lot more compiliant with with the C++ standard and features a faster exception mechanism. Dwarf2, the new debugging format which is also used on other GCC platforms, is now the default.

Now that we have catered for Windows users, Mac developers may be pleased to hear that Objective C support in the editor has also been improved. Users of all platforms will notice the automatic completion of include files, even when using subdirectories.



The editor has received many more new features: Qt Creator completes matching braces, brackets and quotes cleverly so it does not get in your way.

For those who have to deal with heavily nested code, the block highlighting feature will help you to keep an overview of your code. In another nice touch, `Ctrl+Click` will take you to the definition of the current symbol, just like `F2` always did.

Other people that might find this release very interesting are those who want to develop on a netbook. The UI has been made shrinkable. For instance, the Welcome screen will now fall back to using scroll bars as a last resort. Also, the Projects mode was turned into a scrolling expert interface, replacing the former implementation based on a more conventional layout. This also results in a much improved usability of the project settings.

Finally, the new refactoring engine will be put in place. You can already see a precursor of that in the 1.2.90 pre-release, with the in-place renaming of local variables. The final 1.3 release will see even more refactoring utilities geared towards making code more maintainable.

Speaking Your Language

With the release of Qt Creator 1.2, the application automatically adopts to the language of the system locale set in the operating system. This makes Creator accessible to people who feel more comfortable communicating with their IDE in their native language. Developers who absolutely prefer English, but do not work on an English operating system can export `LANG=en` to the environment before launching Qt Creator. This also works on Windows.

Translations are a natural entry point for contributions to Qt Creator. In fact, a significant amount of translations came into existance because of the new Qt Contribution Model (http://qt.gitorious.org/) which made it easy for people to not only contribute code, but also add translations and documentation.

If you would like to contribute in any way, you are very welcome to do so. Contributors are recommended to coordinate their efforts via the `qt-creator@trolltech.com` mailing list. People interested in translating Qt or Qt Creator to their native language may wish to subscribe to the new `qt-l10n@trolltech.com` (Qt localization) mailing list. You can subscribe to these lists at http://lists.trolltech.com.

If you are keen to try out a preview of Qt Creator 1.3, with almost all final features present, give the upcoming beta version a shot. It is expected to be released soon after this QQ arrives in your Inbox. Please use this opportunity to give us feedback so we can all enjoy a better, bug-free Qt Creator 1.3.

Daniel Molkentin *is a software engineer at Nokia, Qt Development Frameworks. As a Computer Science graduate still involved with the KDE project, he dropped Vim in favor of Qt Creator, and never looked back. Originating from the former German capital of Bonn, he moved to Berlin, the new capital, to join the Qt tools team.*

Giving the Doc a Facelift

Writing good documentation is not all about the quality of what we write, but also about how we present the information. A large base of knowledge is of no use to you if there is not a good way of accessing the information you are looking for. The documentation must be accessible and easy to use, as well as accurate and informative.

Lately, we have been looking for better ways to present the Qt documentation on the Web. Our current documentation consists of static pages linked together in the good old fashioned way of Web 1.0. Yes it works, but the potential in Web 2.0 is worth investigating and could increase the usability of the site. Ever since Qt version 2.3 (at least), the Qt documentation has had a uniform look and structure. During the time since those days, Qt has evolved, gained a lot of new features, and now covers a lot more ground than its predecessors. This evolution has also affected the documentation, making the base of information larger and more complex. Despite the growing amount of information the documentation has kept its simple but effective design because it works for our users. That is something we need to preserve when adding new features to it.

Looking into the Future

If you have peeked at the documentation in the Qt 4.6 snapshot, you have probably noticed the search box added in the top corner of the index page. This is a custom Google search engine, scanning our documentation for results matching your query. Now, there is nothing revolutionary about this feature, but this is one way we can improve the documentation usability. Adding such a feature has no remarkable impact on the rest of the documentation. However, adding new features to the site should not be done without research. If we are going to make changes we need an idea or a philosophy on what we want to accomplish.

We want the documentation to be as user friendly as possible. That means that it should be easy to find detailed information on a subject if you are an expert and know what you are looking for, as well as finding good tutorials and overviews if you are a beginner. We also want to provide a structure that requires a minimum of searches and mouse clicks. Today, a search for detailed information on a subject can require browsing as much as seven or eight levels down using links from the index page. This makes navigation hard and tiresome if it is not combined with queries to a search engine. Dynamic menus and a flat structure could help us on our way accomplish this. By creating a good cognitive design, we want to make your journey from question to answer as short as possible so you can focus on creating good applications.

Collecting Feedback

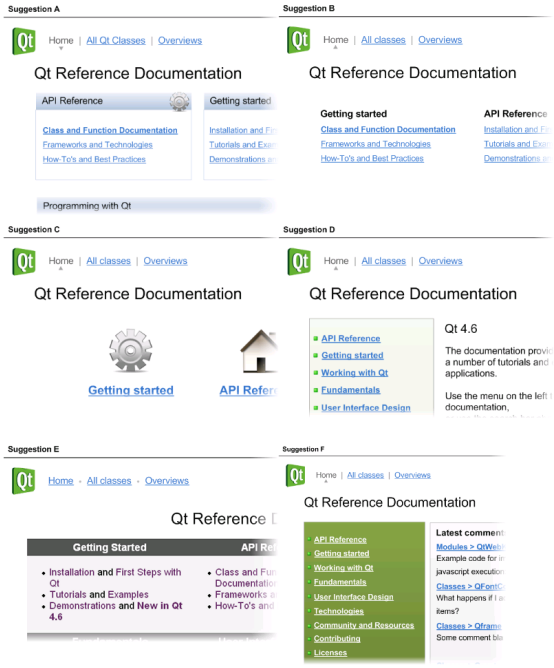
So with these goals in mind we are trying to find a structure and a set of new features that will improve the future Qt documentation. Finding such a structure requires recognizing how the different elements in the documentation relate to one another. Questions come to mind about how they should be categorized regarding technologies, types of features or platforms, and these need to be sorted out. Also, we need to sort out which categories are more relevant and which need to appear on the front page to keep the structure flat. To keep a flat structure, the real trick will be to find out how to present as much information as possible in the least amount of space, and without making it chaotic.

That said, there are probably as many opinions about how the documentation should be organized as there are Qt users, and it is exactly these opinions that we are interested in. That is also why we will talk to our users about this issue during Qt Developer Days in

Munich this year. We want to do some research on how developers using Qt use the documentation, and how they would like the documentation to evolve.

Having created a set of different sample pages, showing a range from super-socially-inspired pages to minimalistic and clean overviews, we want to visualize different concepts and get feedback on what our users think of them. The results of this user testing will be brought back to the team in Oslo and used as input for the design process.

In the collection of sample pages used in our research, we have inserted different features both directly relevant to the specific concept, like blog feeds in a social page concept, but also features that could be used across several different concepts, like instant query suggestions as you type in the search box. We want to encourage our users to contribute to the design by throwing useless features away and maybe even spin off new ideas that we have not thought of yet.



It's Your Turn

The participants at Developer Days will have a special opportunity to have their say on this matter through our user testing. However, this will also be available online. If you go to Qt Labs and our developer blogs, you can have a look at the concepts and give us your opinion. What we want to know is not which font size you like, or which color is your favorite. What is important for us is getting insight on how you work and how you want to find the information you are looking for, whether you are learning new concepts and ways to design your application, or simply finding out how you should use a specific function of a class.
http://labs.trolltech.com/blogs/2009/09/28/giving-the-doc-a-facelift/

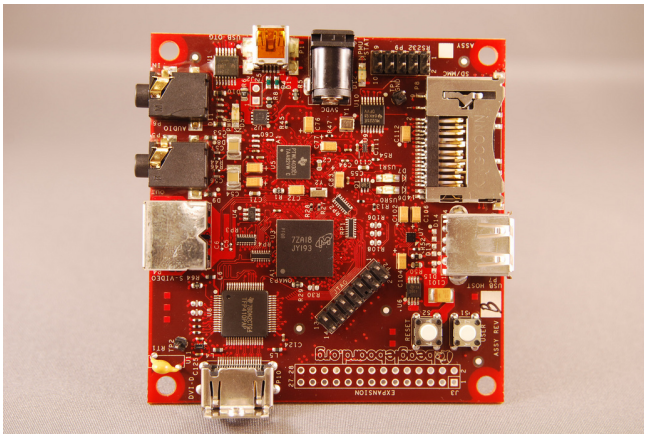
We do not want to create a flashy Web site with blinking stars around your name, congratulating you on solving your problem when you look something up. First and foremost we want documentation that works. We want documentation which is logically organized, using cognitive methods leading you to the information you need, quickly and efficiently.

Morten Engvoldsen *is a technical writer for Nokia, Qt Development Frameworks in Oslo, Norway. He has a passion for writing, good design, well-formed code and strong coffee. Commuting from far-away, he also likes riding trains.*

Qt Must not Languish

If you follow our labs in recent months, you might remember an example that stood out from all the others. Not only because it featured a dismembered finger, or the possibility of accidentally showing your colleagues content that was Not Safe For Work (NSFW), but because it introduced a new way of creating user interfaces in Qt. That's pretty significant in our neck of the woods. This was the QML Flickr example.

Qt has historically been developed to make sure that its users can create their applications without having to think about details of the target platforms, by abstracting away the details and providing features that capture the most common baseline functionality of all supported platforms. Fortunately, the target was desktop platforms and the common baseline was pretty much set.



Qt runs on the Beagle Board† single board computer.

However, in recent years Qt has been ported to an increasing number of platforms where the desktop paradigm isn't really that useful anymore. To be more specific: embedded systems, including anything from a TV to a naked board running Linux you might prototype on.

Making a Declaration

Meet **QtDeclarativeUI** and the QML language! Breaking down user interface construction into elements that both designers and developers can fathom, and collaborate on—it borrows properties from established languages and frameworks in the computing industry, and the result is not the multi-colored concoction you would expect:

```
import Qt 4.6

Rectangle {
    id: QmlExample
    width: 300; height; 250
    gradient: Gradient{
        GradientStop { position: 0.0; color: "#80C342" }
        GradientStop { position: 0.5; color: "transparent" }
        GradientStop { position: 0.0; color: "#80C342" }
    }
    Text {
        id: HelloText
        text: "Troll Acolytes Apply Within!"
        width: parent.width
        y: parent.height/2
    }
}
```

The example above is self-contained, uses a simple language, and is easy to understand and write for developers and designers alike.

Enriching this rather trivial example do something visually appealing is easy! Let's add an animation.

```
scale: SequentialAnimation {
    repeat: true
    running: true
    NumberAnimation {
        from: 0.8; to: 1
        duration: 750
        easing: "easeOutQuad"
    }
    NumberAnimation {
        from: 1; to: 0.8
        duration: 750
        easing: "easeOutQuad"
    }
}
```

This animation will cause the `scale` property to fluctuate smoothly between 1 and 1.5 over 1.5 seconds, then back again. The `SequentialAnimation` allows us to specify several animations that should be executed after each other. Similarly, `ParallelAnimation` allows you to make several animations run at the same time.

Animations can be set in system using states. A state is a set of properties across multiple items—see the documentation for states and transitions in the upcoming Qt 4.6 release for a full description: http://qt.nokia.com/doc/4.6-snapshot/statemachine-api.html

In our case, it will switch between a summary and detail view. When the user clicks the text our UI will move into a different configuration showing more detailed information. First of all, we will add the text, both transparent and placed off-screen:

```
Text {
   id: DetailText
   width: parent.width
   opacity: 0
   wrap: true
   text: "Thou shalt make UIs nice and lustful, and thine
uses will be appreciative. Thine users will rain praise
on thee for aeons to come and copious amounts of gold will
drizzle in your cupped hands for the foreseeable future"
}
```

Next, we create a new state where the details are made visible and moved onto the screen, in the scope of our top-level rectangle:

```
states: [
State {
   name: "DetailState"
   PropertyChanges {
      target: DetailText;
      anchors.top: HelloText.bottom;
      opacity: 1;
      height:(parent.height-HelloText.height)
   }
}
]
```

The base state, used for the `from` values, is implicitly inferred from the declared property values, and has an empty string as its name. With this, we can create a transition that defines how the properties should be changed when moving to a new state. In this example we are only making one state which will be used for both directions—but you could create a different state transition from the base state to a named state and vice versa by filling out `from` and `to` appropriately:

```
transitions: [
Transition {
    from: "*"
    to: "*"
    SequentialAnimation {
        NumberAnimation {
            target: HelloText
            properties: "y"
            easing: "easeInOutQuad"
            duration: 250
        }
        NumberAnimation {
            target: DetailText
            properties: "y,opacity, height"
            easing: "easeInOutQuad"
            duration: 500
        }
    }
}
]
```

Finally, we need to define *when* this transition occurs, and for that we will make the `HelloText` have a clickable area which changes state when clicked:

```
Text {
      id: HelloText
      text: "Hello Troll Acolytes!"
      anchors.centerIn: parent
      MouseRegion {
        anchors.fill: parent
        onClicked: { QmlExample.state=
        QmlExample.state=='DetailState'? '':'DetailState'
        }
      }
   }
```

A mouse region has a click handler, where we can execute JavaScript code. Here we toggle the state between the base state and `DetailState`, and QML handles the rest, using the transition to interpolate the specified properties from their base values to their new values. This results in a smooth slide in and the appearance of the details text underneath the title text.

We can further enrich the UI by applying effects to the items, who are all `QGraphicsItems`. In Qt 4.6 we have added graphics effects and our QML hackers have added these to the language as well. Let's make our details text have a drop shadow. This is simply added as a `effect` property on the `Rectangle`:

```
effect: DropShadow {
      blurRadius: 5
      offset.x: 4
      offset.y: 4
   }
```

Perhaps you want to define a more generic look and feel to re-use across multiple elements. There is no inherent theme support in QML, but as everything is property-based you can simply define a set of properties that define the relevant semantics in a separate file, `CustomTheme.qml`, like this:

```
Item {
    property string textColor: "darkgrey"
    property string borderColor: "#006284"
    property string gradient1: "#80C342"
    property string gradient2: "transparent"
    property string gradient3: "#80C342"
}
```

These properties can be applied elsewhere in your code by defining a theme property. The `CustomTheme` type is inferred from the previous `CustomTheme.qml` file and given an `id`:

```
CustomTheme { id: DefaultTheme }
...
Text {
   color: DefaultTheme.textColor
}
```

DeclarativeUI is based on Graphics View. This means that for each of the elements you use in QML there is a native class exposing a `QGraphicsItem`. The observant reader will also see that this means your own Graphics View components or third-party ones can be easily re-used in QML. However, this is of course not limited to `QGraphicsItems`, but in fact any `QObject` class or instance that you might want to access. This way you can easily encapsulate native functionality and business logic in a C++ object you expose to QML—providing a very distinct separation between the UI and the core algorithms and business logic.

So, to sugar coat our example with something that's written in C++, I will use the eminent robot from our Graphics View examples. To expose a C++ `QGraphicsItem` to QML, simply wrap it in a `QFxItem`

```
class MyItem : public QFxItem {
    Q_OBJECT
public:
    MyItem(QObject *parent=0) : QFxItem()
    {
        Robot *robot = new Robot();
        robot->setParentItem(this);
    }
};
```

and add two simple macros to ensure the type is accessible from the QML engine:

```
QML_DECLARE_TYPE(MyItem)
QML_DEFINE_TYPE(Qt,4,0,7,Robot,MyItem)
```

Then, all we need to do is load a QML file in the DeclarativeUI viewer class:

```
QmlView view;
QString fileName("Simple.qml");
view.setUrl(QUrl(fileName));
view.show();
view.execute();
```

When the example is built and executed, we'll see the dancing robot alongside the previous items we created with QML, simply by adding the following to our QML file:

```
Robot {
    width: 200
    height: 200
}
```

Declarations about the Future

The declarative UI features mentioned in this article are scheduled for Qt 4.7. Currently, to try out the example code accompanying this article, you will need to check out the `kinetic-declarativeui` branch of the Qt Kinetic repository:
http://qt.gitorious.org/+qt-kinetic-developers/qt/kinetic

See the Qt Quarterly site for the example code. Feel free to experiment, and tell us about the cool things you do with it!

Henrik Hartz *works as a Product Manager for Qt, with emphasis on UI and Visualization. When he's not trying to pass himself off as a programmer he tries to break a leg or two snowboarding, exceed the speed of sound on his mountain bike or enjoy the finer shades of various hop and barley brews.*

† *Image by jadonk (http://www.flickr.com/people/jadon/), licensed under a Creative Commons Attribution-Share Alike 2.0 Generic License (http://creativecommons.org/licenses/by-sa/2.0/)*

Qt Quarterly is published by Nokia Corporation, Oslo, Norway.

Authors: Morten Engvoldsen, Henrik Hartz, Daniel Molkentin, Geir Vattekar.
Contributors: Katherine Barrios, Geir Vattekar.
Editor: David Boddie.

A selection of *Qt Quarterly*'s articles and source code is available from `http://doc.trolltech.com/qq/`. Potential contributors should contact the editors at `qt.comments@nokia.com`.

Qt News

Qt 4.5.3 and Qt SDK 2009.4 Released

Qt 4.5.3, the latest in the stable line of Qt releases, is now available. Accompanying the new version of Qt is the release of Qt Visual Studio Add-in 1.1, which replaces the discontinued Qt Visual Studio Integration and can be used together with Qt under both commercial and LGPL licenses.

A new build of the Qt SDK (Build 2009.4) is also available. The Qt SDK contains everything needed to begin cross-platform development with Qt. An installer will set up the system and enable you to use Qt right away.

A list of the latest changes in Qt 4.5 can be found here:
http://www.qtsoftware.com/developer/changes/changes-4.5.3

Qt and the SDK can be downloaded from the usual place:
http://qt.nokia.com/downloads

Qt 4.6 and Qt Creator Previews

In September, Qt Development Frameworks released a technical preview of Qt 4.6 to gather initial feedback on new features in the run-up to the first beta release, due in early-to-mid-October. Hot on the heels of the Qt preview comes the Qt Creator 1.2.90 Technology Snapshot, which showcases some of the features that will be in the upcoming release of Qt Creator 1.3.

One of the long-awaited features that our developers are keen to receive feedback on is the support for the Symbian platform, which has been merged into the Qt 4.6 codebase. Preliminary support for Symbian development is also present in the Qt Creator snapshot.

To follow development of Qt Creator, take a look at the resources in the following article from Qt Labs:
http://labs.trolltech.com/blogs/2009/10/05/creator-13-whats-cooking/

Qt Developer Days 2009

The annual Qt Developer Days are nearly here! Featuring two full days of in-depth technical workshops and seminars on Qt topics, plus a day of introductory seminars and training sessions, these events give participants the chance to meet up and improve their Qt skills with other Qt users and developers.

The registration deadline has now passed for the Munich event, October 12–14. However, registration is still open for the San Francisco Developer Days, November 2–5. Registration for this event closes on October 23, so be sure to register as soon as you can to avoid disappointment.
http://qt.nokia.com/qtdevdays2009

Qt Certification

The brand new Qt Certification Program is launching at Qt Developer Days 2009!

The idea behind the Qt Certification Program is to contribute to the success of Qt by building a community of qualified Qt developers. We believe that more qualified Qt developers will result in better Qt-based products, which will in turn lead to a larger number of satisfied end users.

Take the exam at Qt Developer Days and save 50% on the price! Pre-registration for certification at Qt Developer Days is now available from the following Web site:
http://www.pearsonvue.com/nokiaqt/developerdays

For more information about the program, please visit
http://qt.nokia.com/certification