Code less.
Create more.
Deploy everywhere.

Qt Quarterly

C++ and Qt Programmers' Newsletter

Issue 29 · Q1 2009

Contents

Qt 4.5.0 Release Makes a Splash

On 3 March 2009, Qt released a mega-product release including the much anticipated Qt 4.5, the Qt Creator 1.0 IDE, and Qt's very first SDK. It was received with open arms, nearly paralyzing our servers for a slight moment, which have since gone on to serve about 13 TB of Qt files so far.

What is particularly interesting about this jumbo release is the addition of the LGPL licensing option. This opens up the doors to millions of developers, making it cost-effective to create application masterpieces powered by Qt. This release also introduces a new support and service offering to all Qt users regardless of the license choice.

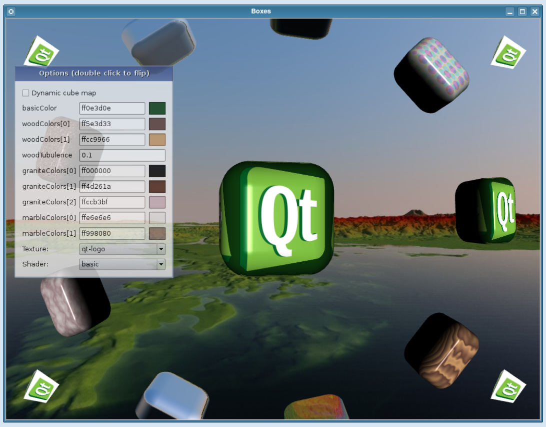
Of course, it would not be a jumbo release if we did not mention the fact that Qt Software will be opening the Qt source code repositories, making it easier for you to contribute patches and add-ons.

The Qt 4.5 release concentrated on enhancing Qt's performance, adding significant improvements to WebKit, porting Qt to Cocoa and a bundle of other key features. With the simultaneous release of Qt Creator 1.0 and the Qt SDK, we are sure that you have more than enough to play around with.

Get started with Qt 4.5, Qt Creator 1.0 and the Qt SDK, and learn about the new features by visiting this link:
http://www.qtsoftware.com/products/whats-new-in-qt/whats-new-in-qt

New Demos and Examples in Qt 4.5

The Qt 4.5.0 release is not short of new examples and demos, showcasing new features and covering useful techniques.



☐ Dynamic cube map
basicColor ff0e3d0e
woodColors[0] ff5e3d33
woodColors[1] ffcc9966
woodTubulence 0.1
graniteColors[0] ff000000
graniteColors[1] ff4d261a
graniteColors[2] ffccb3bf
marbleColors[0] ffe6e6e6
marbleColors[1] ff998080
Texture: qt-logo
Shader: basic

New examples cover drag and drop, data-aware widgets, item views, text handling and XML manipulation, and even an example aimed at developers on Embedded Linux.

Translucent Widgets in Qt

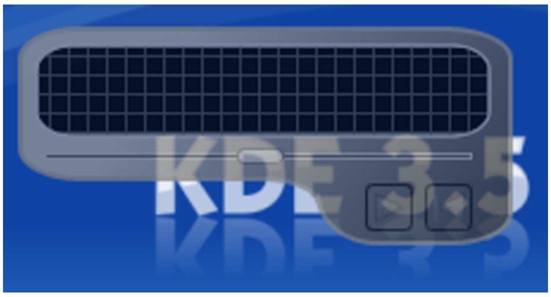
A new and unflaunted feature in Qt 4.5 is translucent top-level widgets—windows that one can see through. Applications that use translucency—many media players, for instance—have become a common sight on today's desktops, and requests about this feature are quite common on the Qt mailing lists. Surprisingly, there are not many GUI toolkits out there that support translucency in a cross-platform way, so we saw the need to flaunt this feature by writing this article.

The process of applying translucency effects to Qt widgets is trivial; simply use colors with an alpha channel while painting the widgets with a `QPainter`. If you want the entire widget to have the same degree of translucency, we have a nice convenience function that removes the hassle of painting the widget yourself. In this article, we will explore the possibilities Qt gives us by implementing a media player (that is, an empty and silent front-end of a media player) and an analog clock.

A Translucent Media Player

In this first example, we let the widget, the media player, keep the same degree of translucency. As we will see later, Qt also supports per-pixel translucency effects.

We start off by giving an image of the player to show what we want to achieve:



When the mouse is over the media player, we keep it opaque. When the mouse leaves the player, we set the player to be translucent. Luckily for us, Qt provides a convenience function to achieve such effects. Here are the two event handlers that are invoked when the mouse enters and leaves the player.

```
void Player::enterEvent(QEvent *event)
{
    setWindowOpacity(1.0);
}

void Player::leaveEvent(QEvent *event)
{
    setWindowOpacity(0.5);
}
```

Developers on Windows 2000 and later, Mac OS X, and modern X11 platforms have had the ability to change the opacity of a whole window for some time. This effect is useful, but limited, especially if we need to change the opacity of individual regions or pixels.

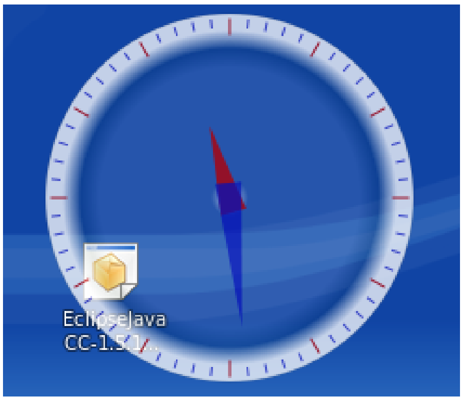
The limitations of this effect are highlighted in the case where we want to apply a mask to the widget to make an unusually-shaped window—see Qt's Shaped Clock example for details.

Shaped windows created in this way often have jagged outlines because there is no way to blend their edges with their surroundings.

What we really want is to be able to make individual pixels at the edge of the window translucent while keeping other parts of the window opaque. The next example shows how this can be done.

A Translucent Analog Clock

In this example, we paint the widget using colors with an alpha channel. We return now to the familiar analog clock found in Qt's standard examples. But we will beef it up a notch by making the widget transparent outside of the clock and giving the center of the clock a nice translucent effect. You can see an image of the clock below.



Let's look at the header file of our translucent clock:

```
class Clock : public QWidget
{
    Q_OBJECT

public:
    Clock(QWidget *parent = 0);
    QSize sizeHint() const;

protected:
    void mouseMoveEvent(QMouseEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void paintEvent(QPaintEvent *event);

private:
    QPoint dragPosition;
};
```

Since we don't have a window frame, we make it possible to move the clock by dragging it with the mouse. We implement this in `mouseMoveEvent()` and `mousePressEvent()`. The `dragPosition` keeps track of where the player is located on the screen.

But it is the translucent painting that interests us here. We do all of that in the `paintEvent()` function. However, before we can start painting, we need to turn translucency on. The code that does this lives in the constructor.

```
Clock::Clock(QWidget *parent)
    : QWidget(parent, Qt::FramelessWindowHint |
                      Qt::WindowSystemMenuHint)
{
    ...

    setAttribute(Qt::WA_TranslucentBackground);

    ...

    setWindowTitle(tr("Analog Clock"));
}
```

Setting the `Qt::WA_TranslucentBackground` widget flag is all that is required before we start painting. In addition, the areas we don't paint in will become fully transparent (unless we decide to paint the background in an opaque color, of course). Note also that we set the `Qt::FramelessWindowHint` window hints to get a frameless window.

We move on to the painting code:

```
void Clock::paintEvent(QPaintEvent *)
{
    ...

    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing);
```

The first thing we do is to set the anti-aliasing rendering hint. As mentioned, this gives the clock the nice, smooth outline. Next, we start painting:

```
    QRadialGradient gradient(0.0, 0.0, side*0.5,
                             0.0, 0.0);
    gradient.setColorAt(0.0, QColor(255, 255, 255, 255));
    gradient.setColorAt(0.1, QColor(255, 255, 255, 31));
    gradient.setColorAt(0.7, QColor(255, 255, 255, 31));
    gradient.setColorAt(0.8, QColor(0, 31, 0, 31));
    gradient.setColorAt(0.9, QColor(255, 255, 255, 255));
    gradient.setColorAt(1.0, QColor(255, 255, 255, 255));
    painter.setPen(QColor(0, 0, 0, 32));
    painter.setBrush(gradient);
    painter.drawEllipse(-side/2.0 + 1, -side/2.0 + 1,
                        side - 2, side - 2);

    ...
```

As you can see, we do not have to do anything special with the painter. After setting the `Qt::WA_TranslucentBackground` attribute, we can just paint as usual. Here, we draw the inner circle of the clock using a translucent colored radial gradient. It is this that gives the effect inside the clock, as you can see from the screenshot.

We will not bore you with a full walkthrough of the `paintEvent()` function. Suffice to say, we paint the rest of the clock; i.e., both hands, and the hour and minute markers on the band.

The X Server and Window Managers on Linux

On Windows 2000 (and later versions) and on the Mac, translucency and transparency should work out of the box. On Linux, some tweaking may be required—as is often the case with state of the art graphic effects (and often some of the not-so-fancy effects, too). Notably, the X server needs to support ARGB pixel values and a compositing window manager is required.

We cannot go into details on all available distributions and window managers. This is mainly because it would take up to much space in this small article (and it has not been possible to check up on them all). A quick Web search should give you the information you need. As an example, we can mention that on Kubuntu using KDE 3, we need to explicitly turn translucency on as it is considered experimental. With the new KDE 4, it is turned on by default; in fact, several desktop effects that come with KDE 4 use translucency effects.

The End?

There are many possibilities when it comes to translucent widgets. In this article, we have only played with top-level widgets. It is also possible to paint child widgets using translucent colors. This could be used for translucent item views, for instance.

As mentioned, the media player is now just an empty, rather dull application. We are planning to extend it with animations using the new Qt Animation Framework, which is planned to be released in Qt 4.6. For good measure, we might also implement the media functionality using the Phonon multimedia framework. So stay tuned for further development.

Geir Vattekar *is a Technical Writer at Qt Software. Writing aside, he enjoys interactive fiction and machine learning. More often than not, however, he is found at home with a cup of tea and Bach on the stereo.*

Experimenting with the Qt Help System

Qt's Help System comes with a powerful full text search back-end based on CLucene. CLucene is the C++ port of Lucene—a high-performance, full-featured text search engine written in Java. Previously, we wrote our own search engine for use with Qt Assistant. However, with the release of Qt 4.4, we began using CLucene to provide a fully-fledged search solution.

To illustrate a use case different from Qt Assistant's document search, we came up with an example on how to use `QHelpSearchEngine` as the base for a predictive text implementation. This experiment is inspired by an example found in a book on the original Lucene search engine. In this article, we'll concentrate on the parts of the example that deal with the Qt Help System; the full example code is available from the Qt Quarterly Web site.

Using QHelpSearchEngine Beyond Qt Assistant

The example displays a small cellular phone-like interface, with numerical buttons mapped to groups of characters. As the user presses keys on the keypad, the sequence of key presses will be saved, and the display will suggest a word that fits the sequence typed. If there is more than one search result, the user can iterate through them.

Typically, CLucene can index several files at the same time. So, in our implementation, we put each word into a separate file. However, we need to build our own compressed help file with words we would like to use in this example.

Generating the Help Files

First, we generate the files required for the full text search: a help project file (`.qhp`) and a help collection project file (`.qhcp`). We begin by running the `pd` application with the `-generate` command. This command will build the `.qhp` and `.qhcp` files. Ensure that your `PATH` variable has been set correctly, so that the `qcollectiongenerator` command line application can be located.

Based on the word list file, `word.lst`, that is included in the example, the application will generate the files that provide the necessary content to be used in the full text search.

So, how does all this work under the hood? To answer that question, let's take a look at the `Generator` class. The constructor builds a `QHash` containing a mapping between the characters and their number on the keypad. This makes it possible to represent a word as a sequence of key presses. It is such sequences we will search for in the `QHelpSearchEngine`.

```
Generator::Generator()
    : appPath(qApp->applicationDirPath())
{
    QStringList ncList;
    ncList << "1abc" << "2def" << "3ghi" << "4jkl"
           << "5mno" << "6pqrs" << "7tuv" << "8wxyz";

    foreach(const QString& nc, ncList) {
        for (int i = 1; i < nc.length(); ++i)
            hash.insert(nc.at(i), nc.at(0));
    }
}
```

Now we need to generate the files for CLucene to index. This is done by reading the `word.lst` content line by line and write each word back into a file named after it. The written file is stored in a list that will provide the files to be included in our `.qhp` file. Notice that we save each word as their numeric sequence in the files. We still have the mapping between the numeric sequence and the actual word (since each file name contains the word), and we can use that to show matches in the display.

A Helpful Reminder

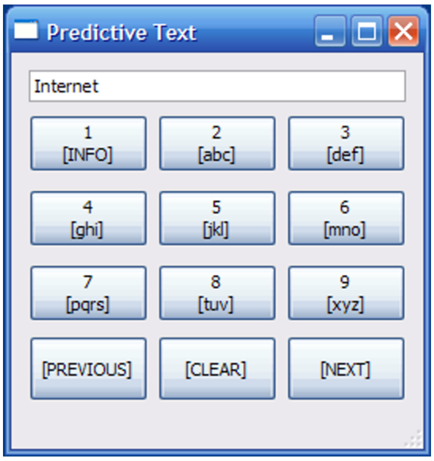
- *Qt Help Project* (`.qhp`): The input file for the help generator, containing table of contents, indices, and references to the actual documentation files (`.html`).
- *Qt Help Collection Project* (`.qhcp`): The input file for the help collection generator, containing references to `.qch` files to be included in the collection.
- *Qt Compressed Help* (`.qch`): The output file of the help generator—a binary file containing information specified in the `.qhp` and compressed documentation files.
- *Qt Help Collection* (`.qhc`): The output of the help collection generator. The `QHelpEngine` operates on this file, as it contains references to any number of `.qch` files.

```
void Generator::generateIndexFiles()
{
    ...
    QFile file(appPath + "/word.lst");
    if (file.open(QIODevice::ReadOnly)) {
        QTextStream stream(&file);
        while (!stream.atEnd()) {
            const QString& word = stream.readLine();
            QFile file(outputPath.path() + "/" + word +
                       ".txt");
            if (file.open(QIODevice::WriteOnly)) {
                file.write(swap(word).toUtf8());
              filesList.append(QString("files/%1.txt").arg(word));
            }
        }
    }
}

QString Generator::swap(const QString& word)
{
    QString nc;
    for (int i = 0; i < word.length(); ++i)
        nc.append(hash.value(word.at(i).toLower()));
    return nc;
}
```

The other functions in this class are responsible for writing the contents of the `.qhp` and `.qhc` files, as well as running the `qcollectiongenerator` command line application.

Predicting with the Help Search Engine

With the `.qhc` and `.qch` files generated, we start the application without any arguments. Each keypress will be cached and the resulting *keyword* (i.e., the sequence of keypresses typed) is used to issue a search query on the full text search.



Internet

Since we only require a simple search query, we will use the default help search query. The string we pass on comprises of the following components: the number we are looking for, the number again with two unknowns represented by *??*, and finally the number appended by a tilde (~). The tilde provides a degree of uncertainty to the number. To illustrate, suppose we are looking for the number *1234*. Our search string will be *1234*, *1234 + ??*, and *1234 + ~*. If you look at the code, you will notice that these extra parameters are escaped with the slash character.

```
void MainWindow::searchNext(const QString& num)
{
    current = 0;
    number += num;

    QList<QHelpSearchQuery> list;
    QString searchString(number + " OR " + number +
        "\?\? OR " + number + "/~");
    QHelpSearchQuery query(QHelpSearchQuery::DEFAULT,
        QStringList(searchString));

    searchEngine->search(list << query);
}
```

Once the query has been executed, the search engine will emit the `searchFinished()` signal containing the number of search hits returned. Now, all we need to do is pickup the first result and display it. Since our search is file-based, the hits just contain the file path for the word that was found. If the file is an HTML file, the search hit will contain the page's title, too.

We already mapped the file's name to its numerical expression and stored it in the file. So, now we just have to extract the filenames to generate a plain list of words.

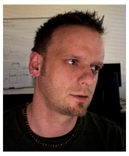
```
void MainWindow::searchFinished(int hitCount)
{
    const int numberLen = number.count();
    QList<QHelpSearchEngine::SearchHit> searchHits =
        searchEngine->hits(0, hitCount);

    QSet<QString> tmp;
    foreach (const QHelpSearchEngine::SearchHit& hit,
             searchHits) {
        const QString& path = hit.first;
        const int start = path.lastIndexOf("/") + 1;
        const int length = path.lastIndexOf(".txt") -
            start;
        tmp.insert(path.mid(start,
            (numberLen < length ? numberLen : length)));
    }
    hits = tmp.toList();

    updateDisplay();
}
```

We can iterate through the hits we obtained, using the **NEXT** and **PREVIOUS** buttons. The **CLEAR** button resets the search hits and clears the internal keyword buffer.

```
void MainWindow::updateDisplay()
{
    QString word;
    if (hits.count() > 0)
        word = hits.at(current);
    ui->lineEdit->setText(word);
}
```

Conclusion

We have seen that the new search engine of the Qt Help System is not limited to searching through documentation. A requirement is that the information can be stored in Qt Help files, though.



***Karsten Heimrich** is a Software Engineer at Nokia, Qt Software in Berlin, Germany. He works on everything related to Qt Assistant and the Qt Help System. In his spare time, he enjoys spending time with his friends and colleagues.*

The Qt Jambi Generator

With the abundance of programming languages existing today, it is not unusual to provide the same set of APIs in more than one language. The Qt framework, for instance, has bindings to languages as diverse as Python, Ruby and Java.

The Java bindings, part of the Qt Jambi product, were created here at Qt Software. Most of this process was automated by a porting tool called the "Qt Jambi Generator", which automatically generates bindings between Java and C++ types and functions. In this article, we would like to share some of the experiences we have made from porting a C++ API to Java.

The Challenge

When we port Qt to Java, we build Qt and then implement a layer of code that communicates between Java and the native C++ library. This layer uses JNI, the Java Native Interface, which is specificially written to allow communication between Java and natively compiled code.

In the porting process, the Qt Jambi Generator reads the public header files from Qt and generates the following:

- A set of Java classes that map to the C++ API classes. These classes contain Java methods that call corresponding C++ functions using JNI, and a C++ void pointer (represented as a long numerical value) which addresses a special book-keeping class that contains necessary information about the binding between the Java and C++ objects.
- Implementations of said JNI functions which route function calls from Java into C++, and which, in the case of constructors, create the C++ objects whenever a user of the API constructs the corresponding Java object.
- Subclasses of all polymorphic types which route virtual C++ calls to their Java counterparts.
- Special `QObject` subclasses that route C++ signal emissions to Java.

Despite their many similarities, the conceptual gap between Java and C++ is large and, at times, unforgiving. For an undramatic example of such a gap, consider the concept of implicit casts in C++. In the Qt framework, implicit casts are used to make code shorter and less verbose. In Java, however, implicit casts are non-existent. For instance, take the following construction of a `QPen` object from a `QColor` in C++:

```
QPen pen(QColor::fromRgb(255, 255, 255));
```

The `QPen` constructor actually takes `QBrush` as its argument, but the `QBrush` object can be implicitly constructed from the `QColor` object behind the scenes. Now consider the equivalent Java code in which the C++ imeplicit casts are replaced with their Java explicit equivalents.

```
QPen pen = new QPen(new QBrush(QColor.fromRgb(255, 255,
                                              255)));
```

While Java is naturally more verbose than C++, the extra step of constructing the `QBrush` object here seems unintuitive and is not how the designers of the API wanted it to look. The solution for such a problem is to manually extend the automatically generated code by adding a handwritten overload to the `QPen` constructor that takes a `QColor` object and creates the `QBrush` object behind the scenes.

And this is precisely the challenge: bridging the gap between the two languages while still retaining as much as possible of the original intent of the API. If the direct port of the API is significantly less convenient than its original, work needs to be done to provide the same level of usability, even if this means providing something that is no longer a word-for-word translation.

For instance, Java's container classes and strings are nestled deep in the language specification with special rules related to them, and they even have their own keywords. In fact, these classes cannot actually be implemented using Java. Directly mapping the equivalent classes in Qt and using these in the API would be a direct translation, but it would not add any value, and it would make the resulting API much less convenient than both its original and its potential. So the Qt Jambi Generator is written to recognize the classes in question and, rather than port them, replace any mention of them in the API with their Java counterparts.

Both the issues brought up so far are easily fixed and their solutions are almost obvious. In the rest of this article, we will focus on more ominous differences in the mechanisms and ideals of Java and C++. These are issues we discovered along the way, and on some level forced us to take pauses and rethink earlier ideas, or even make certain compromises to provide the best possible API when the perfect solution to a problem just doesn't exist.

Before we get into that, however, we will briefly describe Qt Jambi's type systems and how they can be used to add customizations to the generated code.

Type Systems

In addition to the C++ header files, the Qt Jambi Generator bases its porting work on a set of XML files, collectively known as "the type system". These files contain information about the mapped classes that cannot be inferred by the automated process. The most basic of such information is whether a given class is to be considered a value type or an object type.

Value types are typically passed by value from function to function, and, as a user, you would not expect side effects of any alterations made to a value type object. Take for instance a rectangle class such as `QRect` and a `setRect()` method such as the one in `QGraphicsRectItem`. If you initialize your rectangle and pass it to the method, you would expect the method to make a copy of the original and store the copy rather than a reference to your instance. This means that if you alter the rectangle after it has been passed to `setRect()` you would not expect the `QGraphicsRectItem` object to change. This convention can be seen in Qt and also in other Java frameworks such as Swing.

Object types, on the other hand, are usually passed by pointer, and in fact the Qt Jambi Generator only supports passing object types this way. Rather than represent a value which can be copied, they represent a single, uniquely identifiable instance of a type. If a property in a class is of an object type, you would expect its setter to keep a reference to the original object, and that any subsequent changes to the object would also affect the property referencing it. Object types are often polymorphic types such as `QObject`, but can also be non-polymorphic such as `QPainter`.

In addition to these basic rules, the type system also provides a way for its user to customize several parts of the porting process in cases where the automated process gives unwanted results. As an example, you are able to inject handwritten code into generated classes and methods, and you can also customize the actual type conversion code which is generated for a particular argument of a particular function. This evolved quite organically. Required features in the Qt Jambi Generator were added as the need arose, which is why we usually state that it supports the subset of C++ used in Qt. In some cases where users have asked, we have expanded on this, but we do not believe we will ever be able to support every possible accent of C++.

Below is a small excerpt from one of the type system files used when porting Qt to Java:

```
<object-type name="QAbstractTextDocumentLayout">
    <modify-function
     signature="setPaintDevice(QPaintDevice*)">
      <modify-argument index="1">
        <reference-count action="set"
         variable-name="__rcPaintDevice"/>
      </modify-argument>
    </modify-function>

    <modify-function
      signature="draw(QPainter*,QAbstractTextDocumentLayout::PaintContext)">
        <modify-argument index="1"
            invalidate-after-use="yes" />
    </modify-function>

    <modify-function
     signature="drawInlineObject(QPainter*,QRectF,QTextInlineObject,int,QTextFormat)">
        <modify-argument index="1"
            invalidate-after-use="yes" />
    </modify-function>
  </object-type>
```

For more information on the Qt Jambi type system, please refer to the online documentation at http://doc.trolltech.com/.

Bridging the Gap

As mentioned, our main challenge in porting a C++ API to Java was to provide the best possible mapping of the original API's intent despite the conceptual differences between the two languages. We have selected a number of concepts that we considered particularly challenging or interesting, and will now provide examples and descriptions of how these difficulties were solved for Qt Jambi in the cases where they could in fact be solved.

Multiple inheritance

One of the most significant conceptual differences between Java and C++ is the mechanism of multiple inheritance. C++ supports a pure form of multiple inheritance, where any class can inherit from a theoretically unlimited number of other classes. There are certain pitfalls related to this practice (e.g., the diamond problem), but C++ provides its user with a set of tools to work around the obstacles and expects the programmer to use these tools to write correct and safe code.

When Java was designed, multiple inheritance was deemed too complicated for a language aiming to become "simple, object oriented, and familiar". Instead, a special case of the mechanism was implemented: the ability to multiple inherit completely abstract classes known as interfaces. The interface design pattern covers an arguably clean and unambiguous usage area for multiple inheritance, and is in fact how multiple inheritance is mostly used in the Qt framework.

An example of this is the `QPaintDevice` interface which is implemented by both `QWidget` and `QPrinter`. In Qt, painting code can easily be generalized to work on both printers and widgets on the screen simultaneously, as it deals with paint devices and not with the classes directly. The `QPaintDevice` class is an abstract class with no superclass that contains some pure virtual functions, and also a few convenience functions that have actual implementations.

The general mapping of multiple inheritance into Java is to use the interface mechanism. The type system is expected to provide information about which types are used as interfaces in the API, and when code is generated for these classes, the Generator will create both a Java interface with all method declarations, and also a regular Java class which inherits the interface and which contains any method implementations that may reside in the C++ version of the interface. Whenever a class in the C++ hierarchy inherits from the interface, this class will also get a copy of these implementations in Java.

Whenever a Java instance of a Qt Jambi class is passed into a C++ function, we need to convert it into a C++ object. As mentioned, the Java object itself contains an untyped pointer to the corresponding C++ object, and this needs to be cast into the correct type before passing it into the target function. The caveat is this: whenever this conversion happens for an interface type, we will need to know how to cast it to the correct subobject in C++, or it will not be possible for us to pass the correct pointer value into the corresponding C++ function. This requires knowing the full structure of the object in C++. We have to know which classes it is typed with and in what order. In Qt Jambi, this is done using a special method in the Java interface that we guarantee will always return the correct C++ pointer for the object and interface type.

The consequence of this, of course, is that it is impossible for a user of the generated API to make his or her own implementations of the interface, as they will have no way of implementing such a method. Unfortunately, that's an unsolvable inconvenience in the generated API, as we need to be able to statically construct a C++ instance that corresponds to any Java instance that the user of the API creates, and there is such a high number of combinations of superclasses and interfaces to consider that covering them all could be viewed as effectively impossible.

When using Qt Jambi, you will rather subclass the main implementation of an interface and not have the convenience of the interfaces in your own code.

Pointers and Arrays

Where C++ has pointers to objects, Java has references. For many purposes, the two are semantically and conceptually identical, but pointers have a greater range of use cases which are not supported by Java references. In particular, when given a pointer to a location in memory in C++, there is no way to programmatically determine whether it points to a single instance of its type, or to an array of instances. The person writing the API will need to communicate the contract to the person using the API, either by adhering to a specific convention or by documenting the intended usage.

In Qt, pointers to object types are by convention always considered a pointer to a single instance of the type. In order to provide an array of object type instances, one would use double indirection in the API. In the Qt Jambi Generator, pointers to instances of object types will always be handled as pointers to single instances.

Whenever a pointer to a value type instance occurs in the API, the usage will most likely be documented, unless it is apparent from the function signature how it is intended to be used. Consider the following C++ function:

```
void readData(char *buffer, int bufferSize);
```

This is not at all an uncommon usage of pointers, so we wanted to find a general solution for such cases that would provide a match for the API in all instances. This meant making an API for both creating, reading and manipulating C++ pointers that references an arbitrary number of instances of a given type. The class `QNativePointer` was written for this purpose, and any API that uses a pointer to a value type in C++ will be replaced by `QNativePointer` in Java.

```
public void readData(QNativePointer buffer,
                     int bufferSize);
```

The `QNativePointer` class provides us with a similar power as the original API in C++, but also imposes on us the same amount of responsibility. As a result, it is equally easy (or perhaps even easier) to mess up and have your application crash when using this class than when dealing with pointers. Since it introduces a potential instability that is unheard of in Java, we decided early on that we would like to minimize its usage as much as possible. This is done by manually overriding the generated code in the type system. As humans, we can read the documentation and handwrite code which provides the same level of convenience to Java users of the API, for example by using Java arrays in place of the C++ arrays.

Templates and Generic Classes

In Java 5, a mechanism for generic programming was introduced. The syntax chosen is similar to that of templates in C++, but as generic mechanisms go, the two reside in completely different ballparks. Templates in C++ are implemented in a so-called heterogeneous way, meaning that the code is only partially compiled prior to parameterization. The code referencing the generic type is left unevaluated until the type is replaced by an actual type, and the mechanism is powerful and highly optimizable for speed.

In Java, a homogeneous approach was selected instead, meaning that the generic code is compiled before the instantiation happens. This provides greater optimizability for space, and, perhaps most importantly, statically verifiable generic code.

Since there is no way of dynamically parameterizing a C++ template, there is no general way of mapping a generic API from C++ to Java. Thankfully, such API is uncommon in the Qt framework and mostly constrained to container classes which, as has been mentioned, are not ported by the Generator.

The requirement for template support in Qt Jambi first arose when we set out to port QtConcurrent, and it turns out that there exists a special case of templates which could be supported in a general way: whenever the generic code makes absolutely no assumptions about its argument type, and thus can be completely compiled without specialization. This would, for instance, be the case for a generic container written to store objects of any type. In such cases, the Qt Jambi Generator provides some tools to help its user map the equivalent API to Java.

Since an unbounded generic parameter in Java becomes the common superclass `java.lang.Object` when the code is compiled, we simply needed to add generic syntax to the Java class, and then expect the `Object` class in our JNI code. The template can in turn be instantiated with the special type, `JObjectWrapper`, which is the binding type corresponding to `Object`.

Memory Management

Another large conceptual difference between Java and C++ is how memory is managed. Java has implicit memory management, meaning that objects are subject to garbage collection and potentially removed by the runtime when there are no longer any references to them. In C++, memory management is explicit, meaning that the programmer must know how many references there are to a particular piece of memory and manually deallocate the memory once it is no longer needed.

In Qt Jambi, we wanted programmers to be able to depend on the same memory management conditions as they do when using regular Java classes. This is a very hard problem, however, as there really isn't any way of counting the C++ references to a given object.

The risk is that an object is garbage collected while still being used by C++ code, causing dangling pointers and, subsequently, crashes. Under certain conditions, we also cannot know when the object has been deleted by a C++ statement. Specifically, we can detect the deletion of a C++ object if it is a `QObject` (we are told when the object is destroyed) or when it is created in Java and its class has a virtual destructor (we get a call to our generated reimplemented destructor). In other cases, we risk having dangling pointers in our binding layer if C++ deletes an object and we still have Java references to it.

Our solution to this is to assume ownership contracts for any Qt Jambi object and make sure that we change this contract whenever the rules of ownership are changed by a function call.

By default, any object created in Java will be owned by Java and can therefore be deleted by Java's garbage collector. Objects created in C++ and converted into Java will have so-called split ownership, meaning that Java is allowed to garbage collect the Java object but must leave the C++ object alone.

In addition to these two contracts, there is also C++ ownership, which prohibits the garbage collector from interfering until the C++ object has been deleted. Once this happens, the Java object is released and can be collected.

Keeping the ownership contracts valid has to be done manually in the type system. Whenever a function assumes ownership of an object, we need to set a special attribute for the function, causing the Generator to produce code that changes the object's owner to C++. In cases where the function does not assume ownership, but does retain a pointer to the object, we also need to identify this in the type system, and have the Generator produce code that retains references to the Java object for as long as the C++ code is using its pointer.

There is no way of automating this, as the process requires a complex understanding of the underlying code. It is, however, the same potential problem that must be considered by a C++ user of the API, so well-formed APIs will most likely have a clear and documented policy on how this is handled.

A second potential problem related to memory management occurs when functions are called the other way around; i.e., when C++ calls a virtual function which is reimplemented in Java. Consider the case of a virtual function called `paint()` which takes a pointer to a `QPainter` object. The implicit contract is that the implementor of `paint()` should use the painter object to issue draw commands, but that once the pointer goes out of scope, there are no guarantees concerning the lifetime of the object. Hence, the programmer should not retain any reference to the object, lest he may experience crashes when accessing it later. This of course also applies when the reimplementation is in Java.

```
private QPainter painter;

@Override
public void paint(QPainter painter)
{
    this.painter = painter; // Uh-oh
}
```

In Java, we would like to avoid crashing the application whenever it's possible, as hard crashes make it difficult for Java programmers to debug their code. Therefore, we try to identify code which implies this contract, so that we can detach the Java objects from their C++ counterparts once a virtual call has ended. This means that if you do write code like the one above and subsequently try to access the painter object, we can easily detect that the object has been detached and throw an exception, which is much easier to debug.

Types and Identities of Objects

As we have seen, objects are frequently passed back and forth over the language barrier. In such cases our binding layer will need to take the object from the source language and magically turn it into an object in the target language. We would like these two objects to correspond as much as possible.

One such case is when the binding layer is asked to pass a Java object over to a C++ function. Finding a corresponding C++ object to the Java object is easy. We control the Java object completely, and as mentioned we can store the C++ pointer inside. When the conversion is taking place, we can simply retrieve the pointer and cast it to the correct type.

When the emigration is in the opposite direction, though, it's not so trivial. When the binding layer gets an arbitrary C++ pointer and is asked to convert it to a Java object, there are three cases to consider:

1. The object was created in Java.
2. The object was created in C++ and it is the first time it has been converted.
3. The C++ object has also been created in C++, but it has been converted to Java before.

When the object was created in Java, a Java object corresponding to it already exists. In this case we would obviously like to convert the pointer into this specific Java object. In the second case, the binding layer will have to construct a new Java object of the correct type which has bindings to the C++ object. In the third case and in an ideal world, we would like to convert the same C++ pointer into the same Java object every time. This is where it gets tricky, and while I hate being the bearer of bad news, our world turns out to be somewhat non-ideal.

Consider the virtual `paint()` method from the previous section. In the C++ code for the library we are porting, there is a call to this function from a handler for the paint event:

```
void MyWidget::paintEvent(QEvent *)
{
    QPainter p(this);
    paint(&p);
}
```

Now imagine that the `paint()` function is reimplemented in Java. We would like the call to `paint()` to be transmitted into Java and the `QPainter` object to be converted into a Java object and then passed into the reimplementation of the function. In any call to `paintEvent()` the object, `p`, will be allocated on the stack, very likely in the same location in memory every time. Hence, it's likely that the pointer to `p` will be equal with every subsequent call to `paint()` even though the object itself is different. From the binding layer's perspective, there is no way of detecting whether the pointer we are seeing is a pointer to the same object as last time, or a pointer to a new object in the same location. Thus we cannot in general guarantee that there is always a one-to-one relationship between a C++ object and its corresponding Java object. The consequence of this in Qt Jambi is that you cannot always rely on the identity comparison operator for comparing two objects. Luckily, as we will see, this is only relevant for some rare cases in Qt.

In most cases we can support this relationship. Specifically, we can support it whenever we can detect whether the C++ object has been deleted. This is true when its type is a `QObject`, it has a virtual destructor and is constructed by Java code, or is owned by Java as described in a previous section. The case of `QObject` is specially handled using the object's user data, and if the object is not a `QObject`, we maintain an internal cache of pointers that we can use to look up the corresponding Java object, which is updated when we see that the object has been destroyed.

In other cases, we need to construct a new Java object every time we want to pass a C++ object over the language barrier. Whenever the binding layer needs to construct a Java object, another tricky question poses itself: how do we know which class to construct for our new Java object? The statically generated code only knows the type in the signature of the function it is calling, but in the case of polymorphic types, for instance, it is their very nature to be passed into functions expecting a supertype instead, and thus the type in the signature may not be specific enough to fully represent the C++ object in question.

Take `QEvent` as an example. The general event handler, `event()`, will receive objects of subclasses of QEvent that are identified by a type property. The subclasses may contain data which is particular to the specific type, and we may want to cast the object into its actual class to access this data.

```
public boolean event(QEvent event)
{
    if (event instanceof QMouseEvent) {
        QMouseEvent mouseEvent = (QMouseEvent) event;
        System.err.println("Please stop hitting the"
                           + "following button: "
                           + mouseEvent.button());
    }

    return false;
}
```

In this Java code, we are using the instanceof operator and a type cast with an event to first check if it is of the QMouseEvent type, and then, if it is, access the data particular to QMouseEvent.

When the Qt Jambi Generator produces code that converts the event object from C++ to Java, and it has to construct a new Java object to make the conversion, all it knows about the source is that it is of a subclass of QEvent. In order for it to construct the correct subclass, we need to provide the Generator with information on how to determine the actual type. This is also done manually in the type system. In the case of QEvent we will instruct the Generator to produce code that calls the type() function in the event object and creates a Java object with the class corresponding to the type. In cases where the correct type cannot be found (the object may be of an internal subclass which is not mapped in the type system) then the general supertype QEvent is used instead.

Enums and Extensibility

The final topic we will cover is the case of porting enums from C++ to Java. In C++, an enum is a type consisting of a set of named integer values. This is useful for making type-safe enumerated types or flags. In Java, an enum is also a list of named constants, but there's no value related to the enum constants other than their ordinal value. Rather than being castable to a plain old integer like in C++, they are actual objects, and the enum type is a class type which has been special-cased to be used in relation with the switch keyword and which can never be instantiated at run-time. That means that Java enums are not strictly designed to be used as flags, or as identifiers for values.

To work around these limitations, the Qt Jambi Generator takes advantage of the fact that the enum is a class and adds a value() method to it. The enum constants are then statically constructed with their value set to the same as the integer value of the corresponding enum constants in C++. This is then the value used to convert the enum back and forth between Java and C++, and it can be used in place of casting the enum to an integer. The value() method is also used when enum types are made the basis for a flags type, which is a pattern borrowed from the C++ version of Qt.

Another concept made easy in C++ by the fact that enum constants are represented as integers is extensible enums. The concept is used regularly in Qt, specificially to allow extending of types such as QEvent with custom subclasses, and giving these their own Type enum constants for identification. In such enum types, there will be some sort of User constant which represents the base of all custom types, and new enums can be constructed by incrementing this and casting the result to the enum type:

```
QEvent::Type type1 = QEvent::Type(QEvent::User + 1);
QEvent::Type type2 = QEvent::Type(QEvent::User + 2);
```

As was mentioned above, Java's enums do not support construction at run-time, because the very reason for having enums in the language is to have an enumerated type which is statically type safe, something which is no longer true once you make dynamic extensions to the type. In our specific cases, the code is written to interpret the enum type in a more lenient manner, so we needed to be able to construct new objects of certain enum types at run-time.

All enum types in Java automatically become subclasses of the Enum class, and this class does in fact have a constructor, but it cannot be accessed from user code. In order to work around this, we bypass the accessibility check by constructing the object in the unpoliced environment of JNI. In order to guarantee the identity of enum constants, we also cache each constructed enum constant so that it can be retrieved directly later on.

The C++ code above can be written in Java as follows:

```
QEvent.Type type1 = QEvent.Type.
    resolve(QEvent.Type.User.value() + 1);
QEvent.Type type2 = QEvent.Type.
    resolve(QEvent.Type.User.value() + 2);
```

The resolve() method will then check if the given value already corresponds to an enum constant, either one created statically or one created dynamically. If it doesn't, it will construct a new constant in JNI and add it to its cache.

Conclusion

Though producing a completely automatic port of an API to a different language might be viable by always accepting the greatest common divisor as "good enough", the intent of the API will quite possibly not be intact in the resulting, generated code.

For Qt Jambi, our main goal was to expose the ideas of Qt to Java programmers, not just the names of classes and functions. Doing this required manual intervention in many, many cases, and still requires manual intervention whenever new API is added to Qt.

On the other hand, we also wanted the Qt Jambi Generator to produce usable code out-of-the-box, so that a user could quickly reach a point of actually having compilable Java classes to test. In the end, the way we structured our work was usually to try to solve a complex issue in a general and automatable way at first, and then consider whether the resulting API required manual tweaking only when we had something which could already be considered as working.

__Eskil Abrahamsen Blomfeldt__ is a software engineer at Nokia, Qt Software in Oslo, Norway. He's one of the main architects of Qt Jambi and currently spends his time working on state machines and graphics.

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

Authors: Eskil Abrahamsen Blomfeldt, Karsten Heimrich, Geir Vattekar.
Contributors: Alexandra Leisse, Kavindra Devi Palaraja.
Production: Katherine Barrios.
Editors: David Boddie, Geir Vattekar.

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

New Wiki on Qt Centre

Qt Centre, one of the largest Qt Community sites on the Web, keeps a Wiki containing examples, guides, and information about configuration and installation. It contains in-depth articles about Qt classes and concepts, in the form of tutorials and how-tos, complementing Qt's official documentation.

Qt Centre also has a FAQ with the common questions, ranging in complexity from, "Why don't my signals and slots work?" to how to cope with build issues for the MySQL integration.

The Wiki currently contains about 150 pages, and will likely grow steadily in the future. It is worth a look and is found here:

http://wiki.qtcentre.org.

Qt Jambi to Become Community Project

With the Qt 4.5 release, Qt Software announced that it will discontinue internal development of features for Qt Jambi, handing over the future development of the technology to interested Qt and Java developers. The last Qt Software-driven Qt Jambi release will be built against the Qt 4.5 libraries.

Qt Software will support and maintain Jambi for one year after the 4.5.0_01 release; releases of Qt Jambi will be available for download under the LGPL license. The Qt Jambi Generator, the tool that automates much of the porting process, will likewise be up for grabs. The developers of Qt Jambi hope that, as well as taking Qt Jambi forward, interested parties in the community will pick up the generator and have some fun with it—perhaps helping with the effort to keep the Qt dream alive for Java users.

KDE Reaches Milestone with "The Answer"

On 27 January, KDE reached another milestone with the release of version 4.2 ("The Answer") of the popular desktop and development platform. With this release, the community has provided the first implementation of the new technologies ready for end users. Various applications made a huge leap forward in stability and usability and scripting support for Plasma has seen large improvement. Technology previews of KDE applications are available for both Windows and Mac OS X.

Right after the release of Qt 4.5, on 4 March, KDE also made their libraries available under the LGPL license along with the 4.2.1 maintenance release.

http://kde.org/announcements/announce-4.2.1.php

New version of Qt Kinetic on Labs

A new preview of the Qt Kinetic project is now available on Qt Labs. Kinetic's main focus is the creation of a new animation framework, enabling declarative UI design and styling. It aims to make it easier to make dynamic, animated and smooth GUIs. Behind the scenes, the kinetic project also houses a state machine, which powers the animation framework. The machine can also be used for other purposes.

Kinetic can be tested by cloning its git repository git://labs.trolltech.com/qt/kinetic. There are examples in the repository located under examples/animation and examples/statemachine.

The project page for Qt Kinetic can be found here:

http://labs.trolltech.com/page/Projects/Graphics/Kinetic.

Qt Extended Migrates into the Qt Framework

On 3 March, Qt Software announced that Qt Extended will cease to exist as a standalone product. Selected features will be transferred into the Qt framework—making Qt an even richer cross-platform application framework.

The final release of Qt Extended will be version 4.4.3. Qt Software will honor all existing support agreements, and for customers who need continued access to support beyond the term of their current agreement, Qt Software is offering the possibility of purchasing supplemental support.

Winners of Pimp My Widgets

Winners of the *Pimp My Widgets* developer contest have been found. In the contest Qt developers "pimped" widgets and simple applications from several categories for a chance to win the grand prize of a Segway® i2 Personal Transporter, or one of three Nokia N810 Internet Tablets.

Entries were judged based on five criteria:

- Best Use of Qt
- Usefulness
- Coding Creativity
- Portability
- Bling Factor

Developers from all around the word sent in all kinds of pimped out widgets, and the competition was fierce. In the end, the winner turned out to be Jukka-Pekka Maakelaa with his `QuickCalendarView`. This intuitive widget provides three ways to determine visible timespan, toggles quickly between day and month view, and provides intuitive ways to view information on appointments.

You can read more about the competition on its web pages. Here you can also download the `QuickCalendarView` widget, and also the widgets of the three runners up. The web page is found here:

http://www.qtsoftware.com/developer/pimp-my-widgets-developer-contest

Qt @ Nokia Developer Summit

28–29 April 2009, Monaco

Join the first annual Nokia Developer Summit and make new connections, learn new skills and develop new ideas to create tomorrow's technologies. Qt experts will be presenting their ideas on how Qt can help you develop a mobile application that gives a truly compelling user experience. In addition, our Qt experts will highlight the power of the Qt cross-platform strategy and how easy it is to take your desktop application to the mobile environment.

Presentations at the event include "Nokia Keynote: Power of cross platform development using Qt" by Benoit Schillings, Chief Technologist at Qt Software, "Qt on Nokia platforms (S60 and maemo)" by Mika Rytkönen, Director, Program Management, Nokia, and "Qt overview – Enabling cross platform app development from desktop to mobile Nokia".

To register for the Nokia Developer Summit, please visit:

http://www.developersummit2009.com/