Qt Jambi™ Whitepaper
Technology Preview 3

Trolltech®

www.trolltech.com

Abstract

This whitepaper describes Qt Jambi, a technology that integrates the Qt® cross-platform C++ framework for GUI applications with the Java® programming language. This technology enables Java developers to take advantage of the features provided by Qt, from within Java Standard Edition 5.0 and Java Enterprise Edition 5.0 as well as later versions. In addition to a unique inter-object communication mechanism, Qt Jambi has excellent cross-platform support for 2D and 3D graphics, internationalization, SQL databases, XML and TCP/IP network protocols. Qt Jambi produces high-performance applications that run using a Java virtual machine and runtime environment.

Qt Jambi™ Whitepaper
Technology Preview 3

Trolltech®

www.trolltech.com

Contents

1. Introduction 2
    1.1. What Qt Jambi Offers to Java Programmers 2
    1.2. What Qt Jambi Offers to C++ Programmers 4
2. Main Features 5
    2.1. Signals and Slots 5
    2.2. Layout Management 6
    2.3. Qt Designer 7
    2.4. Application Resource Management 8
    2.5. Graphics System 9
3. Bridging the Conceptual Differences Between Java and C++ 10
    3.1. Value Types and Object Types 10
    3.2. Memory Management 11
    3.3. Fundamental Qt Classes 11
    3.4. Enums 12
    3.5. Multiple Inheritance 12
    3.6. Public and Protected Member Variables 12
4. How Qt Jambi Works under the Hood 13
    4.1. The Qt Jambi Generator 13
    4.2. Type Mapping 14
    4.3. Polymorphic Shell Classes 14
    4.4. Signal Wrapper Classes 15
5. The Qt Development Community 15
6. Summary 16

1. Introduction

Qt Jambi is the Java version of the popular Qt C++ cross-platform framework, opening a world of possibilities for Java and C++ programmers alike. It is an officially supported technology aimed at Java programmers who want to create great-looking GUI applications using a first-rate GUI framework.

With the release of Qt Jambi, Java programmers are invited to join the Qt development community and write software in Qt. The technology also enables C++ programmers to integrate their Qt code with Java.

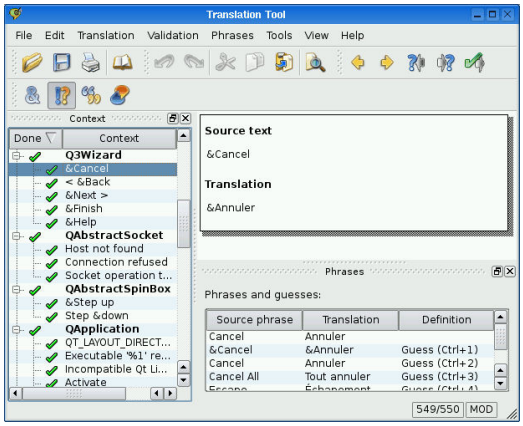
1.1. What Qt Jambi Offers to Java Programmers

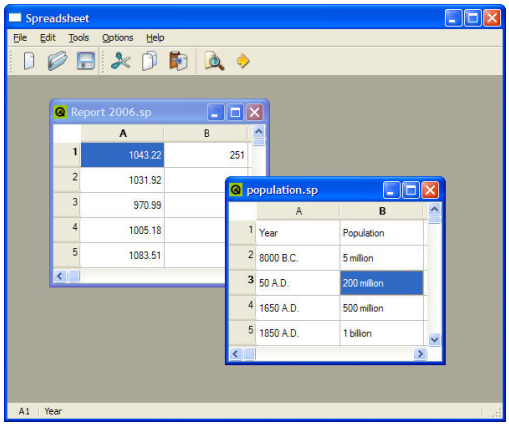
Compared to Java frameworks such as Swing and SWT, Qt Jambi's main advantage is its intuitive and powerful API, which it inherits from Qt. Code written in Qt Jambi tends to be concise and highly readable.

One reason for this is Qt Jambi's "signals and slots" mechanism, which is similar to Microsoft®'s delegates. For inter-object communication, Swing uses the listener pattern, forcing developers to implement a listener interface to receive events. This requires several lines of glue code each time. In contrast, connecting a Qt Jambi signal to a slot requires only one line of code.

Another reason is Qt Jambi's layout management classes. Conceptually, these classes are similar to Swing's `BoxLayout` and `GridBagLayout`, but they are much easier to use and consistently produce better results.

Qt Jambi also introduces a full-blown main window architecture with support for dock windows, something that neither Swing nor SWT offers. Dock windows are windows that the user can move inside a dock area on the side of an application's main window. The user can undock a dock window and make it float on top of the application or minimize it.

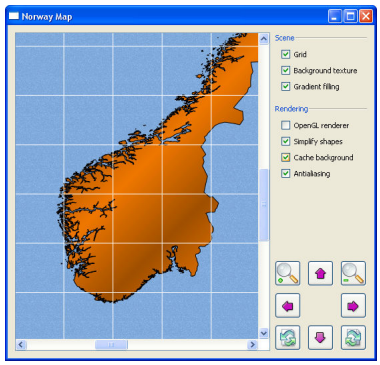




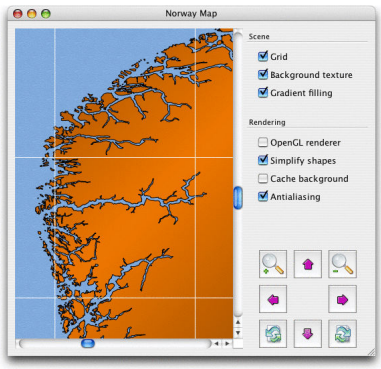
No matter on which platform they run, Qt Jambi applications automatically have the native look and feel. They honor any Windows® or Mac OS X® theme, and respect user preferences for colors, fonts, sounds, etc. In addition, Qt Jambi fully takes advantage of platform-specific extensions such as ActiveX®, ClearType®, XRender and OpenGL®. Qt Jambi also supports desktop integration, e.g., making it possible for applications to launch

a web browser, mail composer, and other external resources using the facilities provided by the user's desktop environment.

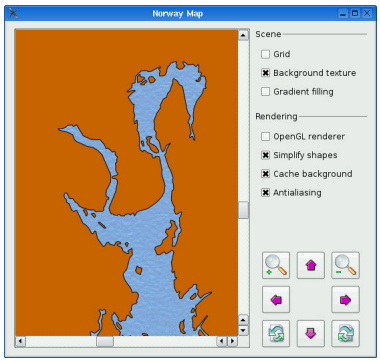
While Qt Jambi applications look native by default, it is entirely possible to give them a distinctive look. One approach is to subclass QStyle or any of the built-in styles (e.g., QWindowsStyle). This works because Qt, like Swing, emulates the look and feel of the different platforms it supports, rather than being a thin wrapper over single-platform toolkits. In addition, Qt Jambi supports widget style sheets following a syntax similar to that used by HTML Cascading Style Sheets (CSS), letting you easily perform minor customizations or create your own look. For the more adventurous, a third approach is to subclass Qt's widgets directly and reimplement their paint, mouse, and key event handlers to gain full control over their look and feel.



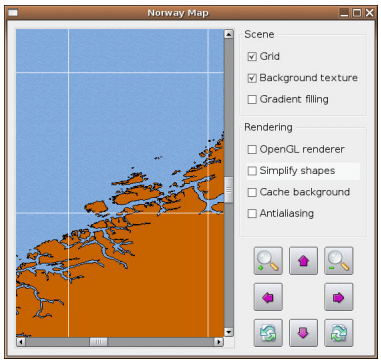
Windows XP



Mac OS X



KDE (X11)



GNOME (X11)

Qt Jambi's unique Graphics View framework provides responsive handling of large numbers of canvas items. The framework's event propagation architecture enables precise double-precision interaction with the items on the scene. The items can handle key events, mouse press, move, release and double click events, and they can also track mouse movement. Graphics View provides very fast item discovery using a BSP (Binary Space Partitioning) tree, and as a result of this, it can visualize large scenes in real-time, even with millions of items.

With Qt Jambi, applications can be written entirely as source code, or using *Qt Designer* to speed up development. *Qt Designer* is Trolltech's tool for designing and building graphical user interfaces from Qt components using a convenient drag and drop approach. Once the interfaces are created, functionality can easily be added by writing the required code separately. The main benefit of *Qt Designer* over other form builders (e.g., the IntelliJ® IDEA

and Borland®'s JBuilder®) is *Qt Designer*'s intuitive handling of layouts and signal–slot connections.

1.2. What Qt Jambi Offers to C++ Programmers

In addition to integrating the complete Qt API with the benefits that Java provides (notably garbage collection), Qt Jambi enables C++ programmers to easily integrate their Qt code with Java. This is done using the Qt Jambi Generator, which reads the code of an existing C++ library and generates a Java API for it.

The Qt Jambi Generator can ease the transition from a C++ environment to a Java environment, and it can greatly improve the degree and efficiency of collaboration between C++ and Java programmers within a project team. See "Bridging the Conceptual Differences Between Java and C++" (page 10) for details.

2. Main Features

Qt is the de facto standard C++ framework for high-performance cross-platform software development. Qt Jambi is implemented as a thin layer around Qt's C++ libraries. Each method call made to a Qt Jambi class is redirected to the corresponding C++ class.

In this section, we will present some of Qt Jambi's core features and the advantages they provide to Java programmers compared to other frameworks.

2.1. Signals and Slots

The traditional approach for inter-component communication in Java is using the listener pattern. This approach has the disadvantage that the components must know about all the interfaces they intend to support at the time they are written, meaning that you need more code and the code will be less flexible.

Qt Jambi's signals and slots mechanism is different. Qt widgets emit signals when events occur. For example, a button will emit a "clicked" signal when it is clicked. The programmer can connect this signal to one or more "slots" (i.e., methods) that are then invoked whenever the signal is emitted.

Consider a small inter-object communication application with a slider and a spin box. To synchronize the two components, the following code is required when writing the application in Swing:

```
slider.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
        spinner.setValue(slider.getValue());
    }
});
spinner.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
        slider.setValue((Integer)spinner.getValue());
    }
});
```

With Qt Jambi, connecting the slider and spin box requires only these two lines:

```
slider.valueChanged.connect(spinBox, "setValue(int)");
spinBox.valueChanged.connect(slider, "setValue(int)");
```

Qt Jambi's widgets already provide signals that cater for most situations. When writing custom components (widgets or other objects), we can define our own signals and emit them at appropriate times. For example, let's suppose that we are developing a `MyBankAccount` component and want to emit a `balanceChanged()` signal whenever the minimum and/or maximum value changes. We would declare the signal as follows:

```
public Signal1<Integer> balanceChanged;
```

Qt Jambi uses Java generics to make signals type-safe. The number "1" in the signal declaration specifies that it is a signal that has one argument (the new balance).

In `setBalance()`, we would emit the `balanceChanged()` signal using `emit()`:

```
void setBalance(int newBalance) {
    if (balance != newBalance) {
        balance = newBalance;
        balanceChanged.emit(balance);
    }
}
```

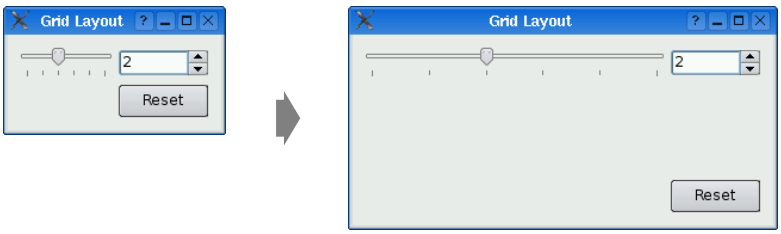
In addition, Qt Jambi supports a "connect slots by name" mechanism that makes it possible to automatically connect to a child's signal by following a specific naming convention. For example, to respond to the `clicked()` and `valueChanged(int)` signals of the `closeButton` and `slider` child widgets, we can simply implement the following slots:

```
public void on_closeButton_clicked() { ... }
public void on_slider_valueChanged(int newValue) { ... }
```

and call `QtJavaUtils.connectSlotsByName()` in the class's constructor.

2.2. Layout Management

Qt Jambi provides a simple and powerful way of specifying the layout. Consider the dialog shown below at two different sizes:



Using Qt Jambi and `QGridLayout`, all you have to do is to add the widgets to the layout specifying their position in the grid as row–column pairs:

```
QGridLayout layout = new QGridLayout(this);
layout.addWidget(slider, 0, 0);
layout.addWidget(spinBox, 0, 1);
layout.addWidget(resetButton, 2, 1);
layout.setRowStretch(1, 1);
```

When the dialog is resized, the extra horizontal space goes to the slider widget, because by default sliders have an "expanding" size policy, meaning that they grow to take any available space. Vertically, the extra space is taken by the empty row between the spin box and the Reset button, because it has a non-zero stretch factor.

To create the same layout pattern using Swing, more code is required:

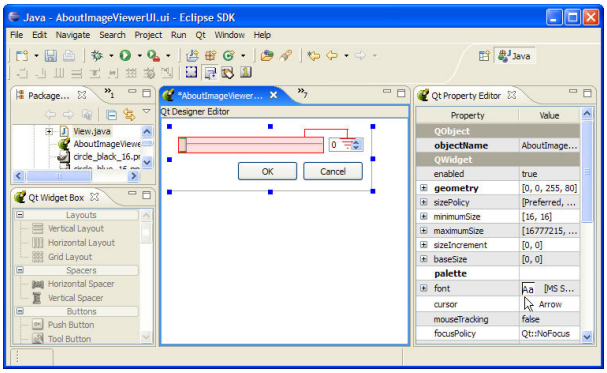
```
GridBagLayout layout = new GridBagLayout();
GridBagConstraints constraint = new GridBagConstraints();

constraint.fill = GridBagConstraints.HORIZONTAL;
constraint.insets = new Insets(10, 10, 10, 0);
```

```
constraint.weightx = 1;

layout.setConstraints(slider, constraint);

constraint.gridwidth = GridBagConstraints.REMAINDER;
constraint.insets = new Insets(10, 5, 10, 10);
constraint.weightx = 0;

layout.setConstraints(spinner, constraint);

constraint.anchor = GridBagConstraints.SOUTHEAST;
constraint.fill = GridBagConstraints.REMAINDER;
constraint.insets = new Insets(10, 10, 10, 10);
constraint.weighty = 1;

layout.setConstraints(resetButton, constraint);

JPanel panel = new JPanel(layout);
panel.add(slider);
panel.add(spinner);
panel.add(resetButton);
```

2.3. Qt Designer

Qt Jambi's layout managers are pleasant and intuitive to program with, and it is not unthinkable for programmers to develop entire Qt Jambi applications purely by writing source code. Still, many developers can benefit from a visual approach for designing forms, because it makes it easy to correct or change designs. At the same time, developers want their dialogs to scale properly to fit whatever window size the end-user prefers.

This is where *Qt Designer* comes to the rescue. *Qt Designer* is a user interface design tool that provides a convenient drag and drop approach to dialog creation and that generates the layout code for you. Once the interfaces are created, functionality can easily be added by writing the required code separately.



Qt Designer stores the user interfaces as `.ui` files, a straightforward XML-based file format. These files are converted to Java source files by the Java User Interface Compiler (JUIC) command-line tool. Most developers would integrate JUIC into their favorite IDE (e.g., Eclipse) so that it is run automatically whenever the `.ui` file changes.

The generated class is called `Ui_MyDialog` and contains the code to set up the widgets and layouts as specified in the `.ui` file. To use it, one approach is to create a `QDialog` subclass with a `Ui_MyDialog` member variable and to call `setupUi()` in the constructor:

```
package com.mycompany.myproduct;

import com.trolltech.qt.gui.*;
```

```
public class MyDialog extends QDialog
{
    private Ui_MyDialog ui = new Ui_MyDialog();

    public MyDialog() { ui.setupUi(this); }
}
```

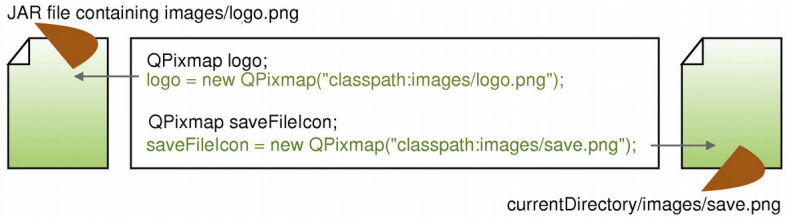
Any custom functionality goes in our `QDialog` subclass. For example, to make the Reset button do something, we would implement an `on_resetButton_clicked()` slot as follows:

```
private void on_resetButton_clicked() {
    ui.slider.setValue(0);
}
```

The `setupUi()` function automatically calls `QtJavaUtils.connectSlotsByName()`, which connects the Reset button's `clicked()` signal to the `on_resetButton_clicked()` slot, ensuring that our slot gets called whenever the user clicks the button.

2.4. Application Resource Management

Qt Jambi provides a uniform syntax for accessing resources in the Java classpath, no matter whether they are located directly on the disk or in a JAR bundle. While the standard Java API only supports accessing resources in an undocumented subset of its file I/O operations (which does *not* include the `java.io.File` class), Qt Jambi allows resources to be used wherever a file name is expected. Resources are identified by a `classpath:` prefix.



For example, here's how to access an image from the classpath using Qt Jambi:

```
QImage img = new QImage("classpath:images/logo.png");
```

The traditional Java approach is somewhat more cumbersome:

```
Image img = Toolkit.getDefaultToolkit().createImage(
                    ClassLoader.getSystemResource(”images/logo.png“));
```

Similarly, to print out the file size of a resource file using the traditional approach, the following code is required:

```
int fileSize;
try {
    fileSize = ClassLoader.getSystemResource("images/logo.png")
                          .openConnection()
                          .getContentLength();
} catch (Exception exception) {
    ...
}
```

Using Qt Jambi you only need one line of code:

```
int size = new QFile("classpath:images/logo.png").size();
```

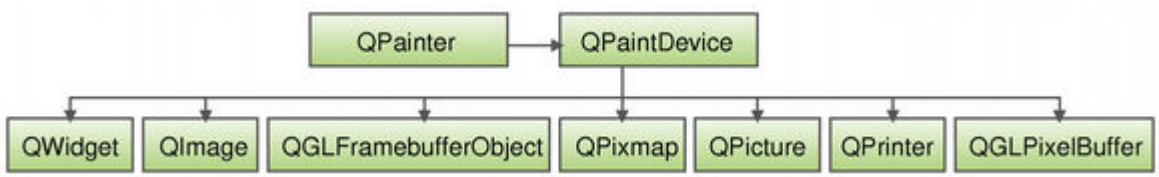
In addition, Qt Jambi can list the contents of directories in the classpath. For example:

```
List<String> fileNames = new QDir("classpath:images").entryList();
```

In contrast, using the standard Java API, there is no easy way of accessing the contents of directories in the classpath.

2.5. Graphics System

Qt Jambi implements its own separate rendering pipeline based on Qt's paint system. Qt Jambi provides painting on both screen and print devices using the same API. It is primarily based on the `QPainter` and `QPaintDevice` classes. `QPainter` is used to perform drawing operations and `QPaintDevice` is an abstraction of a two-dimensional space that can be painted on.



Compared to other toolkits and frameworks, Qt Jambi's paint system is easier and more intuitive to use. For example, consider a component with some graphical contents and a print button. Using Swing, the programmer must implement the `Printable` and `ActionListener` interfaces and the following two functions to pop up a print dialog enabling the user to print the contents:

```
public int print(Graphics g, PageFormat pf, int pi)
        throws PrinterException {
    if (pi >= 1)
        return Printable.NO_SUCH_PAGE;

    // perform drawing

    return Printable.PAGE_EXISTS;
}
public void actionPerformed(ActionEvent e) {
    if (e.getSource() instanceof JButton) {
        PrinterJob printJob = PrinterJob.getPrinterJob();
        printJob.setPrintable(this);
        if (printJob.printDialog()) {
            try {
                printJob.print();
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

Using Qt Jambi, all the programmer has to do is to connect the print button's `clicked()` signal to a custom `print()` slot:

```
void print() {
    QPrinter printer = new QPrinter();
    QPainter painter = new QPainter();
    QPrintDialog printDialog = new QPrintDialog(printer, this);
```

```
        if (printDialog.exec()) {
            painter.begin(printer);

            // perform drawing

            painter.end();
        }
    }
```

In addition, Qt Jambi's text handling is based on native font support, providing access to platform features like ClearType anti-aliasing on Windows and Mac OS X, and sub-pixel anti-aliasing on X11™. `QPainter` can even be used to generate PDF documents, without requiring a third-party component.

Qt Jambi's rendering pipeline co-exists with Java2D™ rendering, meaning that it is possible to use Java2D for advanced imaging and Qt Jambi's graphic system for painting widgets.

3. Bridging the Conceptual Differences Between Java and C++

Qt Jambi is the Java version of the Qt API. When adapting Qt's C++ API to Java, some issues arose with certain C++ constructs that have no direct equivalents in Java. In this section, we will see how Qt Jambi handles memory management, how it converts C++ types into Java types, and how it copes with multiple inheritance, operator overloading, and member variables.

3.1. Value Types and Object Types

Most of Qt's classes can be described either as "value types" or as "object types". When mapping Qt's API to Java, value types and object types are treated differently.

- *Value types*: These are classes such as `QImage` and `QString` that provide a copy constructor and an assignment operator. These behave like built-in types and are usually passed to functions by value or by constant reference (e.g., `const QString &`).
- *Object types*: These are classes such as `QObject`, `QWidget`, and `QEvent`. These have virtual functions and cannot be copied. In C++, these classes are typically instantiated using the `new` operator and are passed by pointer (e.g., `QWidget *`).

In the Qt Jambi API, object types are passed as references. For example, a C++ method with the signature

```
bool eventFilter(QObject *object, QEvent *event)
```

is mapped to the following Qt Jambi method:

```
bool eventFilter(QObject object, QEvent event)
```

Qt value types are handled a bit differently. In the Qt Jambi API, value types appear as references. Thus, a C++ method with the signature

```
QRect mapToDevice(const QRect &rect, const QSize &size) const
```

has the following signature in Qt Jambi:

```
QRect mapToDevice(QRect rect, QSize size)
```

When we call a Qt Jambi method that takes value types, we can safely assume that the method will not alter the arguments behind our back. In addition, the return value is always an independent copy that we can alter without producing undesirable side-effects.

For Qt methods that take a pointer or a non-const reference of a value type, or returns such a construct, the mapping to Java is handled on a per-case basis. If the pointer represents an array, it is usually mapped to a Java array. For example,

```
void drawRects(const QRect *rects, int length)
```

is mapped to

```
void drawRects(QRect rects[])
```

In this example, the `length` parameter is omitted, because Java arrays know their own length.

3.2. Memory Management

Java provides a garbage collector that automatically detects dynamically allocated objects that are no longer accessible and reclaims the memory, simplifying memory management significantly. Traditional approaches for mapping C++ APIs to Java typically result in crude Java APIs that don't take advantage of Java's garbage collector.

With Qt Jambi, C++ value types (i.e., types that have a copy constructor and an assignment operator, such as `QPoint` and `QRect`) are garbage collected just like any other normal Java object, meaning that there is no need to call `dispose()` explicitly to release the memory allocated by these objects. Classes like `QObject` and `QWidget` fall outside the garbage collector's scope. Because Qt Jambi objects normally are created as children of the window in which they live, we only need to call `dispose()` on application windows, like in Swing and SWT.

3.3. Fundamental Qt Classes

There is some overlap between Qt's C++ classes and fundamental Java classes. Qt Jambi maps the overlapping Qt classes to the corresponding Java classes. This ensures that Java programmers can use familiar Java construct (such as `String`'s `+` operator and the foreach loop) while still accessing the power of Qt. It also enables them to combine Qt Jambi APIs and other Java APIs seamlessly. The table below lists the Qt C++ classes that are mapped to equivalent Java classes:

Qt (C++)	Qt Jambi (Java)
`QChar`	`char` and `java.lang.Character`
`QHash`	`java.util.HashMap`
`QList`	`java.util.List`
`QMap`	`java.util.SortedMap`
`QString`	`java.lang.String`
`QThread`	`java.lang.Thread`
`QVector`	`java.util.List`

In addition to these classes, the Qt API also provides an abstract value type, `QVariant`, that can hold values of many C++ and Qt types. Variants are used extensively by the item view classes and the database module.

In Java, there is no real need for such a type, because the language already provides `Object` as an abstract type. This works because all Java classes inherit `Object`. For that reason, the Qt Jambi API uses `Object` where Qt uses `QVariant`. Extra features provided by `QVariant` are mapped to static methods in `com.trolltech.qt.QVariant`.

3.4. Enums

Starting with version 5.0, the Java language supports enums as a means of providing type-safe constant values. Although the Java syntax is very similar to that of C++ enums, regular Java enums are implemented as objects, which means that they cannot be used as integer constants or as bit flags. Qt Jambi works around this limitation and maps the C++ enums to Java enums, guaranteeing type safety for both enums and flags (the Qt Jambi flags are implemented using the generics based `com.trolltech.qt.QFlags` class).

3.5. Multiple Inheritance

A few classes in Qt have more than one base class. In Java, multiple inheritance is possible only with interfaces. To work around this limitation, Qt Jambi uses interfaces that wrap the classes used in multiple inheritance contexts.

Consider the following C++ class definition:

```
class QWidget : public QObject, public QPaintDevice
{
    ...
};
```

The `QWidget` class inherits both `QObject` and `QPaintDevice`. In Qt Jambi, the class is defined as follows:

```
public class QWidget extends QObject implements QPaintDeviceInterface
{
    ...
}
```

In this example, `QPaintDeviceInterface` is an interface that declares the same methods as `QPaintDevice` without implementing them. Any member variable and method implementation found in the original `QPaintDevice` class is put in `QWidget` instead. As a result, the Qt Jambi version of `QWidget` has exactly the same API and functionality as the C++ version.

3.6. Public and Protected Member Variables

The C++-based Qt API is primarily based on methods, but there are some classes that give direct access to the member variables of an object; e.g., `QStyleOption`. Since JNI only provides access to native resources through methods, each of these member variables is implemented as a set and get method pair. For example, the C++ member variable

```
QString text;
```

is available through

```
String text();
void setText(String text);
```

in Java.

4. How Qt Jambi Works under the Hood

Qt Jambi is a static layer of code that transmits Java calls made to the Qt Jambi API into the natively compiled, C++-based Qt libraries, and vice versa. This layer relies on the J2SE Runtime Environment (JRE®) and the Java Native Interface (JNI).

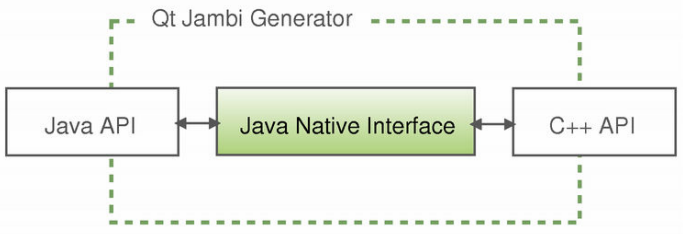
Qt Jambi uses JNI to communicate between the C++-based Qt API and Java. JNI is supported by most Java virtual machines, and is a standard programming interface for writing Java native methods and embedding the Java virtual machine into native applications.

Qt Jambi is partly implemented in Java and partly in C++. Most of the code is automatically generated by a special tool, the Qt Jambi Generator, but some parts are handwritten. The handwritten parts include the foundation for the Qt Jambi library, an implementation of the signals and slots mechanism, and a few convenience Java methods.

4.1. The Qt Jambi Generator

The Qt Jambi Generator is a command-line tool that reads class definitions written in C++ and generates code that maps a C++ library onto an equivalent Java API. The generated code ensures that method calls made in Java are redirected to the corresponding method in the C++ library.

The generator supports a selected subset of C++, covering the most common constructs such as multiple inheritance and operator overloading. In this section, we will describe how the generator handles some of these. For more information, refer to the generator's online documentation, which is available at http://doc.trolltech.com/qtjambi.



The generator uses the Java Native Interface (JNI) to communicate between the C++-based Qt API and Java. For example, consider a C++ class `MyClass` containing a method called `myMethod(const QRect &)`. To enable communication between Java and the C++ code, the generator creates a Java class definition for `MyClass` containing a declaration of `myMethod(QRect)`. It also generates C++ code to map the Java version of the method to the version implemented in the C++ library.

When invoking the generator, we can pass an XML file as command-line argument to customize the output. Among other things, the XML file makes it possible to add or remove methods in generated classes, to modify the signature of methods, and to specify C++-to-Java type conversions.

4.2. Type Mapping

When converting C++ method signatures to Java, the generator requires that we specify which classes are value types and which classes are object types (p. 10). For object types (such as `QObject`, `QWidget`, and `QEvent`), the generator requires that such objects never are passed by value in any of the C++ methods. (This requirement is rarely a problem in practice since most object types have a disabled copy constructor and assignment operator.)

When mapping object types from Java to C++, the generator looks up the Java object to fetch a pointer to its C++ equivalent; in other words, Java objects of these types maintain connections to the corresponding C++ objects at all times. When mapping from C++ to Java, the objects must be handled differently depending on whether they were originally constructed in Java or inside the C++ library: Objects created on the Java side always contain a pointer linking to its Java object, and the mapping is a simple pointer lookup. When mapping objects created within the C++-based library, on the other hand, the pointers to the corresponding Java objects are kept in a separate table and the mapping is potentially a table lookup; e.g., for `QObject`s this is only required the first time a reference to the corresponding Java object is requested since the reference then will be stored in the object.

In the Java part of Qt Jambi, the value type classes are identical to any other class, and their objects are passed as references like any other Java object. Since the C++-based Qt library passes these types by value or constant reference, such objects will be copied by the C++ implementation (using Qt's meta-object system) when mapping from Java to C++. Similarly, return values will be copied before they are passed back into Java.

Although most of the C++-based Qt API is mapped directly onto the equivalent Java-based API (e.g., `int` and `QObject`), the C++ concepts of pointers and non-const references, as well as pointers to pointers, do not have direct Java equivalents. For example, a C++ pointer sometimes corresponds to a Java reference, sometimes to a Java array. To ensure that the Java version of the API is as powerful as its C++ equivalent, the generator encapsulates C++ pointers (and non-const references) using `QNativePointer`. For example, the Qt Jambi Generator maps

```
QApplication(int &argc, char **argv)
```

from the C++-based Qt library onto

```
QApplication(QNativePointer argc, QNativePointer argv)
```

in Java. `QNativePointer` can check the data type to prevent incorrect accesses (e.g., accessing a `char **` as an `int *`). In practice, `QNativePointer` seldom occurs when using Qt Jambi because it normally provides alternative signatures for methods that use pointers to value types (e.g., `QApplication(String args[])`).

4.3. Polymorphic Shell Classes

Qt's classes rely heavily on polymorphism. When mapping Qt to Java, it is necessary to be able to reimplement virtual functions, such as `QWidget`'s `mousePressEvent()` and `keyPressEvent()` functions.

In C++, each polymorphic object has a pointer to a table listing the virtual functions that should be used for this object. The actual implementation of this table is part of the C++ instance and is not accessible to the programmer; e.g., a `QWidget` object created in Java cannot modify this table.

To make the Qt Jambi library as powerful as the C++-based Qt library, Qt Jambi has built-in functionality to forward virtual method calls made on the C++ side to the Java side. For example, when `QWidget`'s `mouseMoveEvent()` method is called, the reimplemented Java method is called instead. Since these calls require some CPU resources, Qt Jambi only calls the Java implementation of a virtual method if it has been reimplemented on the Java side (i.e., if no virtual methods are reimplemented, no calls will be made into Java).

To enable programmers to reimplement virtual functions and to call protected functions in the C++ library, the generator creates a shell subclass for each class in the C++ API. For example:

```
class QtJambiShell_QWidget : public QWidget {
    ...
};
```

The shell class inherits the C++ class and, whenever an instance of a class is constructed in Java, a corresponding native object of the shell class is constructed by the generated native code. If a class has been extended by a user's custom Java class, and one or more of the virtual methods in the class have been reimplemented, the shell class ensures that it is the reimplemented Java implementations that are called instead of the C++ implementations.

4.4. Signal Wrapper Classes

In Qt Jambi, the signals and slots mechanism is implemented independently of the native C++-based Qt mechanism. As a result, the generator must provide a way of linking the two together to ensure that a signal emission inside the C++ library causes the corresponding Java signal to be emitted as well.

This is done by implementing signal wrapper classes. A signal wrapper class is a `QObject` subclass that serves as a link between each signal in the C++ library and its twin in the generated Java code. The first time some Java code creates a connection, an initialization method is called on the C++ object owning the signal. This method prepares the object so that each of the signals in its class is connected to the corresponding slot in the signal wrapper class, which in turn emits the appropriate Java signal when executed.

5. The Qt Development Community

> *Companies and independent developers from around the world are joining the Qt development community every day. They have recognized that Qt's architecture lends itself to rapid application development. These developers, whether they are targeting one or many platforms, benefit from Qt's consistent and straightforward API, powerful build system, and supporting tools such as Qt Designer. With Qt Jambi, all this is also available to Java developers.*

Qt has an active and helpful user community who communicate using the qt-interest mailing list, the Qt Centre web site at http://www.qtcentre.org, and a number of other community web sites and weblogs. In addition, many Qt developers are active members of the KDE® community. Qt customers receive our quarterly developers' newsletter, *Qt Quarterly*. A growing number of commercial and open-source add-ons from third parties are also available; see http://www.trolltech.com for the most up-to-date information.

Qt's extensive documentation is available online at http://doc.trolltech.com. The documentation for the Qt Jambi technology is not ready at the time of this release, but the Qt Jambi API is sufficiently close to the C++ Qt API, so it is possible to use the Qt reference documentation to develop Qt Jambi source code. See "Using Qt's Documentation for Qt Jambi Development" (available at http://doc.trolltech.com/qtjambi) for more information. A complete set of documentation for Qt Jambi is expected to be available as part of the final release.

There are also a number of books in English, French, German, Russian, and Japanese that present and explain Qt programming, including Trolltech's *C++ GUI Programming with Qt 4* (ISBN 0131872494).

Online References:

http://doc.trolltech.com/qtjambi
http://partners.trolltech.com/
http://lists.trolltech.com/qt-jambi-interest/
http://lists.trolltech.com/qt-interest/
http://doc.trolltech.com/qq/

6. Summary

With the release of Qt Jambi, writing commercial software in Qt is no longer a privilege limited to C++ developers. Now, Java developers are invited to take the advantage of Qt's features from within Java's Standard Edition as well as the Enterprise Edition.

Qt is the de facto standard C++ framework for high performance cross-platform software development. Qt Jambi is implemented as a thin layer around the C++-based Qt library. In this whitepaper we have described some of Qt's core features and the advantages they present to Java programmers. In addition to providing the complete Qt API inside the Java environment, Qt Jambi also enables C++ programmers to easily integrate their Qt code with Java. We have given a brief description of the technology, as well as a description of how Qt Jambi resolves the conceptual differences between C++ and Java.

It is Trolltech's goal to expand the Qt framework to support customer projects that require, or may benefit from, implementing code in other languages (e.g., C#). Qt Jambi can be considered proof of concept that the goal can be reached for similar languages in the future, encompassing technology and valuable experience that can be reused.