Code less.
Create more.
Deploy everywhere.

Introduction to Qt Quick for C++ Developers

Today's consumers and enterprise users are tough to please. They grew up using slick UIs on their game consoles and seeing even fancier UIs at the movies. Specifications and feature lists alone are no longer selling points; they are means to an end. Visual impact and experience WOW are the selling points that matter today, and the consumer expects this visual delight whether they are using a powerful corporate notebook computer, a set-top box, or a mobile device.

Delivering this experience requires designers and developers to work together like never before. Gone are the days where designers can throw pixel maps over the wall and expect developers to implement their vision. So too are the days when developers could code purely for performance without regard to visual appeal. Design/development/test must become an iterative cycle, not a linear path.

Qt Quick is built for the way product teams work today. Core business logic is coded by developers and optimized for performance, the interface is crafted by designers working with visual tools, and integrated tooling supports round-trip iteration between the disciplines.

Qt Quick delivers performance because it builds on the Qt application and UI framework. The Qt framework is known for high runtime performance and small footprint, making it ideal for mobile, embedded, and netbook applications.

Qt Quick extends the Qt framework with QML, a declarative language that codes the way designers think. Each frame of a story board is declared as a branch in an element tree; each visual aspect of a frame is declared as a property of elements on the branch; each transition between frames can be decorated with a variety of animations and effects.

Qt Quick includes Qt Creator, a development environment built for collaboration between designers and developers. Designers work in a visual environment, developers work in a full-featured IDE, and Qt Creator supports round-trip iteration from design, to code, to test, and back to design.

Contents

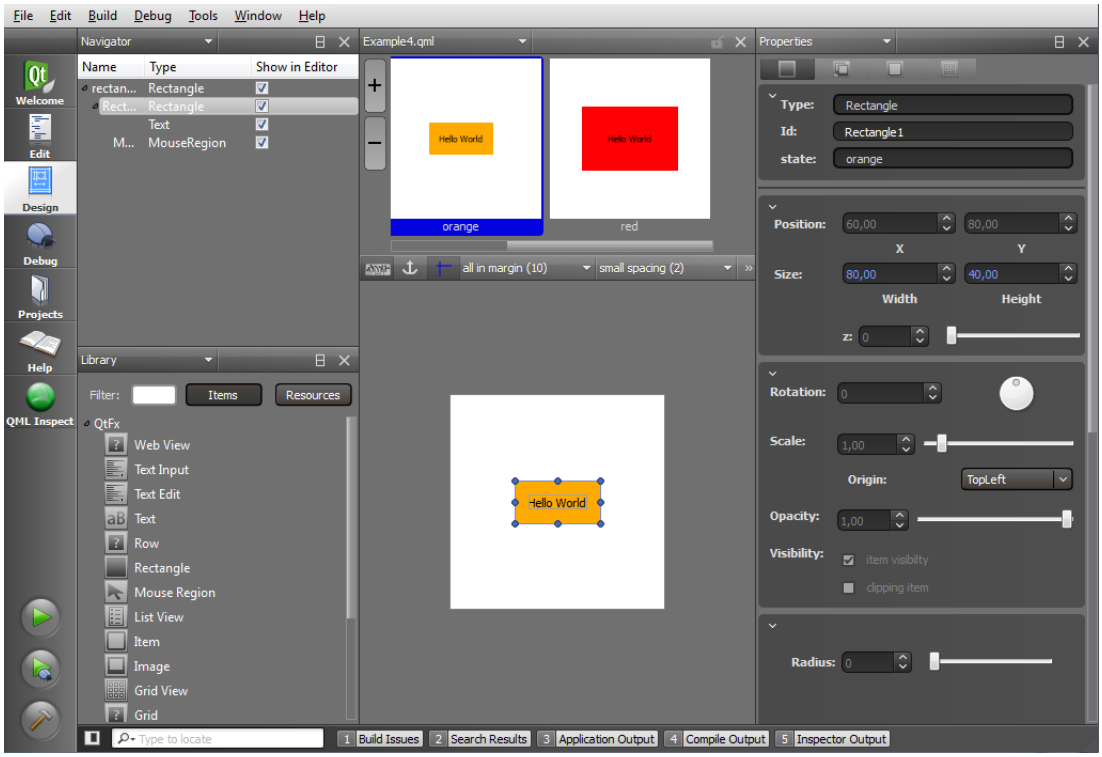
1. Qt Quick Overview

Qt Quick consists of the QML language, the QtDeclarative C++ module that integrates the QML language with C++ objects, and the Qt Creator tool that now includes extensions to support the environment. Qt Quick helps programmers and designers collaborate to build the fluid user interfaces that are becoming common in portable consumer devices, such as mobile phones, media players, set-top boxes and netbooks. Using the QtDeclarative C++ module, you can load and interact with QML files from your Qt application.

QML provides mechanisms to declaratively build an object tree using QML elements. QML improves the integration between JavaScript and Qt's existing QObject based type system, adds support for automatic property bindings and provides network transparency at the language level.

QML elements are a sophisticated set of graphical and behavioral building blocks. These different elements are combined together in QML documents to build components ranging in complexity from simple buttons and sliders, to complete internet-enabled applications like a Flickr photo browser.

Qt Quick builds on Qt's existing strengths. QML can be used to incrementally extend an existing application or to build completely new applications. QML is fully extensible from C++ through the QtDeclarative module.



Qt Creator's user interface for creating Qt Quick components

2. A short introduction to QML

QML is a rich language, and a full treatment is beyond the scope of this paper. This paper will instead provide an introduction to what QML can do and how to integrate QML with C++ to get the best of both worlds: high-performance business logic built in C++ and highly dynamic user interfaces using QML. A full treatment of QML is available in the online documentation.

Understanding QML begins with the concept of elements. An element is a template for a basic building block out of which a QML program will be built. QML supports for example visual elements of types Rectangle and Text, interaction elements of type `MouseArea` and `Flipable`, and animation elements of type `RotationAnimation` and `Transition`. There are also complex element types that allow the developer to work with data, implement views in model-view architectures, and other housekeeping element types that will just add confusion at this point.

All QML elements include one or more properties (for example `color`) that can be controlled by the developer and many elements include signals (for example `onClicked`) that can be used to react to events or changes in state.

2.1. Visual elements: Hello World

Enough text; it's time for the obligatory Hello World example. Here is the code required to place the text *Hello World* on top of a simple background rectangle:

```
import Qt 4.7

Rectangle {
    width: 300
    height: 200
    Text {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        text: "Hello World"
    }
}
```

Snippet 1: Hello World

Let's dissect this simple code. The *Hello World* example is a QML document, meaning it is a complete block of QML source code, ready to run. QML documents generally correspond to plain text files stored on a disk or network resource, but can also be constructed directly from text data.

A QML document always begins with one or more import statements. Here you see the import of Qt 4.7. To prevent elements introduced in later versions from affecting existing QML documents, the element types available within a document are controlled by the imported QML Modules. That is, QML is a *versioned* language.

Next you see the Rectangle element template used to create an active object. Objects can contain other objects, creating parent-child relationships. In the code above, the Rectangle object is parent to the Text object. The Rectangle element also defines a top-level window for managing the visual bounds and focus segmentation of the full UI.

Tech note: The `children` property of QML elements contains a list of all visual children of the element; the `resources` property contains an equivalent list of all non-visual objects. Both lists are populated implicitly by default, or you can explicitly populate them if convenient. A third property, `data`, is a list containing objects in either the `children` or `resources` lists. You cannot explicitly populate the data property but it may be useful if you need to iterate through a list of visual and non-visual objects.
So you can write:

```
Item {
    Text {}
    Rectangle {}
    Timer {}
}
```

instead of:

```
Item {
    children: [ //default property and implicitly assigned
        Text {},
        Rectangle {}
    ]
    resources: [ //default property and implicitly assigned
        Timer {}
    ]
}
```

Within objects, properties are bound to values using the *property : expression* statement. There are two aspects of this statement that bear explanation.

Firstly, expression is a JavaScript expression, which means you could set the properties based on a calculation, a condition, or other complex JavaScript manipulations. For example, you could set the aspect ratio of the rectangle based on the value of a variable orientation.

Secondly, binding is different from assignment. In an assignment, the value of a property is set when the assignment statement is executed and is fixed thereafter until and unless the statement is executed again. In a binding, a property is set when the binding statement is first executed, but will change if and when the result of the expression used to set the property changes. (If desired, you could assign a value to the property using property = expression inside a JavaScript block.)

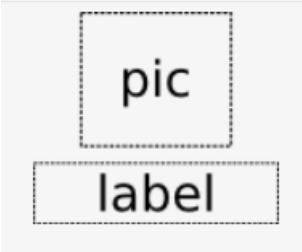
Consider what happens when orientation changes from portrait to landscape (possibly because of a sensor within a mobile device). Because of property bindings, the aspect ratio of the parent rectangle will change and the text element anchors will react to the change to re-center the text.

Tech note: Property binding is achieved by a NOTIFY signal being declared from within a Qt C++ object using the Q_PROPERTY() macro in a class that inherits QObject. If you do not know what that means, don't worry. If you plan to build your application purely in QML, you do not need to know this as it is merely part of the plumbing that Qt Quick handles for you. If plan to include some C++ in your application, the Qt C++ developers on the team will already know what this means.

The `anchors.horizontalCenter: parent.horizontalCenter` statement aligns the center of the text with the center of the parent rectangle. Anchors provide a way to position an item by

specifying its relationship with parent or sibling items. (Note: if you check the online documentation for the `Rectangle` element you will not see the `anchors.horizontalCenter` property listed. Look closely and you will see the `Rectangle` element inherits all the properties of the QML `Item` element; the `Item` element provides the `anchors.horizontalCenter` property.)

There are currently seventeen anchor properties available, allowing you to align, center, and fill elements relative to each other and to set margins and offsets. For example, Snippet 2 shows a `Text` element anchored to an `Image` element, horizontally centered and vertically below, with a margin.



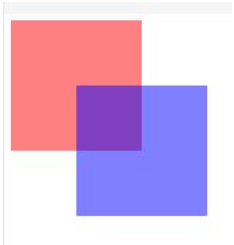

```
Text {
    id: label
    anchors.horizontalCenter: pic.horizontalCenter
    anchors.top: pic.bottom
    anchors.topMargin: 5
    ...
}
```

Snippet 2: Use anchors to align elements

2.2. Layering visual elements

QML visual elements can be layered on top of each other with transparency using `opacity : real` where real varies from 0 (transparent) to 1 (opaque). For performance reasons this should be used sparingly, especially in animations as each layer in the scene will need to be rendered at runtime for each frame of the animation. This may be fine for rapid prototyping, but before final deployment it is best to pre-render as much of the scene as possible, then simply load pixel maps at runtime.

Snippet 3 produces two offset and overlapping rectangles, one red and one blue, with transparency invoked such that the overlapping square will be purple. Notice how the child (blue) rectangle inherits the 50% opacity from its parent (red) rectangle.



```
Rectangle {
    opacity: 0.5
    color: "red"
    width: 100; height: 100
    Rectangle {
        color: "blue"
        x: 50; y: 50; width: 100; height: 100
    }
}
```

Snippet 3: Snippet: Use transparency sparingly

Tech note: Inherited elements always inherit their base element properties; e.g., a `Rectangle` is an `Item`, so all properties of an `Item` will always be present in a `Rectangle` element. Child objects can sometimes inherit the properties of their parents due to the way **QGraphicsView** works. This means that when you embed an element within another element certain properties have an effect on it's children. For example, in the above snippet, the `childRectangle` element is affected by the 50% opacity of the `parentRectangle` element.

2.3. Interaction elements: mouse and touch

To add mouse or touch interaction you need to add a `MouseArea` object. The `MouseArea` object lets the user click and drag the mouse (or touch point). Other available interaction elements include `Flickable`, `Flipable`, and `FocusScope`.

Note that the `MouseArea` object can be separate from any visually apparent objects, providing the designer flexibility. It is quite possible, for example, to create a visual representation of button for a user to click and then surround the visual representation with a larger mouse area that allows the user to "miss" the visible element by a few pixels.

To introduce a mouse region to the *Hello World* example, the rectangle containing the text is made a child of a new rectangle that will define the mouse area.

The `MouseArea` element includes signal handlers that allow you to write JavaScript expressions that will be called upon certain events or changes in state. Available handlers include `onClicked`, `onEntered`, `onExited`, `onPressed` and `onReleased`. In the example above, the `onClicked` signal handler toggles the color of the rectangle.

This example changes the color of the rectangle in response to any valid click. A click is defined as a press followed by a release, both inside the MouseArea (pressing, moving outside the `MouseArea`, and then moving back inside and releasing is also considered a click). The full syntax for the handler is `MouseArea::onClicked (mouse)` where the mouse parameter provides information about the click, including the x and y position of the release of the click, and whether the click was held. Our example does not care where the click occurred.

The *Mouse-touch interaction* snippet shows a simple case of visualizing state by changing one value in response to one event. The `onClicked` statement will quickly become ugly if you try to change multiple values in response to multiple states. That's where QML state declarations come in.

2.4. State Declarations

QML State declarations define a set of property value changes from the base state. The base state is the initial declaration of property values, and is expressed by using an empty string as the state name. After a state change you can always revert to the base state by assigning an empty string to the `state` property.

In the following snippet, states for the two colors are implemented. In the definition of the red rectangle, the id property is set. Named objects can be referenced by siblings or descendants. Two states are also defined: red and orange. The state property is assigned to give the element an initial state.

State elements include a `when` condition that can be used to determine when a state should be

```
import Qt 4.7

Rectangle {
    color: "#ff0000"
    width: 310
    height: 210
    MouseArea {
            anchors.fill: parent
            onClicked: {
                if (parent.color == "#ff0000") {
                    parent.color = "#ff9900";
                } else {
                    parent.color = "#ff0000";
                }
            }
    }
    Rectangle {
        width: 300
        height: 200
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter

        Text {
            anchors.horizontalCenter: parent.horizontalCenter
            anchors.verticalCenter: parent.verticalCenter
            text: "Hello World"
        }
    }
}
```

Snippet 4: Mouse-touch interaction

applied. Here you see the red state applied when the `MouseArea` is currently being pressed.

The defined state not only sets the color for each state, but also the width and height of the rectangle. The orange state provides a larger button. To utilize the states, the mouse region onClicked JavaScript is updated.

It is possible to define a set of states using code, as in this example, or using the graphical Qt Quick Designer component in Qt Creator.

To create animations between states, Transition elements are defined. Transition elements can use the information from the base state and the target state to interpolate the property changes using Animation elements. Animation elements in turn can use a number of different parametrized easing curves and grouping techniques, giving the developer and designer a high degree of control over how and when properties change during a state transition. This is discussed in further detail later on.

```
id: buttonRect;

state: "red"
states: [
    State {
        name: "red"
        when: mouseArea.pressed == true
        PropertyChanges {
            target: buttonRect;
            color: "red";
            width: 80; height: 40
        }
    },
    State {
        name: "orange"
        when: mouseArea.pressed == false
        PropertyChanges {
            target: buttonRect;
            color: "#ff9900";
            width: 120; height: 80
        }
    }
]
```

Snippet 5: Define states

2.5. QML components

The discussion of the *Hello World* snippet (page 3) described the contents of a QML document. How a QML document is named also matters. A QML document name that begins with an upper-case case letter defines a single, top-level QML component. A QML component is a template that is interpreted by the QML runtime to create an object with some predefined behaviour. As it is a template, a single QML component can be "run" multiple times to produce several objects, each of which are said to be instances of the component.

Once created, instances are not dependent on the component that created them, so they can operate on independent data. Here is an example of a simple `Button` component (defined in a `Button.qml` file) that is instantiated four times by `application.qml`. Each instance is created with a different value for its text property:

```
MouseArea {
    anchors.fill: parent
    onClicked: {
        if (parent.state == "red") {
            parent.state = "orange"
        } else {
            parent.state = "red";
        }
    }
}
```

Snippet 6: Simple state transitions


```
import Qt 4.7

Column {
    spacing: 10

    Button { text: "Apple" }
    Button { text: "Orange" }
    Button { text: "Pear" }
    Button { text: "Grape" }
}
```

Snippet 7: Use the button four times with different text properties

```
import Qt 4.7

Rectangle {
    property alias text: textItem.text

    width: 100; height: 30
    border.width: 1
    radius: 5
    smooth: true

    gradient: Gradient {
        GradientStop { position: 0.0; color: "darkGray" }
        GradientStop { position: 0.5; color: "black" }
        GradientStop { position: 1.0; color: "darkGray" }
    }

    Text {
        id: textItem
        anchors.centerIn: parent
        font.pointSize: 20
        color: "white"
    }

}
```

Snippet 8: The Button.qml file creates a button component

Note that QML documents may also include create components inline using the `Component` element.

2.6. Animation elements: fluid transitions

Animation effects are key to a fluid UI. In QML, animations are created by applying animation objects to object property values to gradually change them over time. Animation objects are created from the built-in set of animation elements, which can be used to animate various types of property values. In addition, animation objects can be applied in different ways depending

on the context in which they are required.

There is a detailed treatment of animation in QML in the online documentation. As an introduction, let's consider transitions.

Snippet 9 shows the code to animate the movement of a rectangle. The snippet creates a `Rectangle` object with two states: the default state, and an added moved state. In the moved state, the rectangle's position changes to (50, 50). The `Transition` object specifies that when the rectangle changes between the default and the moved state, any changes to the x and y properties should be animated, using an `Easing.InOutQuad`.

```
import Qt 4.7

Rectangle {
    id: rect
    width: 100; height: 100
    color: "red"

    states: State {
        name: "moved"
        PropertyChanges { target: rect; x: 50; y: 50 }
    }

    transitions: Transition {
        PropertyAnimation {
            properties: "x,y";
            easing.type: Easing.InOutQuad
        }
    }
}
```

Snippet 9: Animated state transitions

You can apply multiple transitions to an item as in Snippet 10. (Remember anything you can do to an item you can do to a `Rectangle`). By default a transition is applied to all state changes. For greater control you can set the from and to properties to apply a transition only when changing from a given state, to a given state, or between given states.

```
Item {
    ...
    transitions: [
        Transition { ... }
        Transition { ... }
    ]
}
```

Snippet 10: Multiple transitions

2.7. Model-View pattern in QML

Using QML in a model-view design pattern is a natural. QML can create fluid, visually appealing views into models whether the models are created in C++ or directly in QML.

QML currently provides three elements devoted to creating views into models. The `ListView` and `GridView` elements create list and grid views respectively. The `PathView` element lays out model-provided items on a path, for example a loop path that allows you to create a carousel interface into a list.

Let's create two different views into the same model – a basic contact list.

You can build models directly in QML using the ListModel element among others. Snippet 11 shows how to create a contacts model where each contact record includes a name, a phone number, and an icon. Each element in the list is defined by a `ListElement` element; each entry includes two data roles, name and icon. Save the document in the file `ContactModel.qml` for access later. (Notice the initial capital letter that makes this file an accessible component.)

```
import Qt 4.7

ListModel {
    ListElement {
        name: "Bill Jones"
        number: "+1 800 555 1212"
        icon: "pics/qtlogo.png"
    }
    ListElement {
        name: "Jane Doe"
        number: "+1 800 555 3434"
        icon: "pics/qtlogo.png"
    }
    ListElement {
        name: "John Smith"
        number: "+1 800 555 5656"
        icon: "pics/qtlogo.png"
    }
}
```

Snippet 11: Defining a List Model in QML

Snippet 12 uses the `ListView` element to lay out items horizontally or vertically. The snippet sets the model property to the `ContactModel` component just created. The delegate property provides a template defining each item instantiated by the view. In this case, the template shows the name and number roles using the built-in `Text` component. If you choose, you can define delegate components the same way you define any other QML component.

> **Tech note:** When using C++ models, the names used to refer to the different roles of the model are set using the QAbstractItemModel::setRoleNames() method. More on C++ models later in the Using Qt Quick in C++ applications section.

```
import Qt 4.7

ListView {
    width: 180; height: 200

    model: ContactModel {}
    delegate: Text {
        text: name + ": " + number
    }
}
```

Snippet 12: List view into contact model

Now let's get a little fancier and build a view into the contacts model that looks like a 3D carousel and allows the user to flick her way through the list. The resulting view and code is shown in Snippet 13. Note the creation of an inline component for use as the delegate property in the `PathView` element.


```
import Qt 4.7

Rectangle {
    width: 240; height: 200

    Component {
        id: delegate
        Column {
            Image { anchors.horizontalCenter:
                    name.horizontalCenter;
                    width: 64; height: 64;
                    source: icon
            }
            Text { text: name; font.pointSize: 16 }
        }
    }

    PathView {
        anchors.fill: parent
        model: ContactModel {}
        delegate: delegate
        path: Path {
           startX: 120; startY: 100
        PathQuad { x: 120; y: 25; controlX: 260; controlY: 75 }
        PathQuad { x: 120; y: 100; controlX: -20; controlY: 75 }
        }
    }
}
```

Snippet 13: Rotating carousel view into contact model

3. Using Qt Quick in C++ applications

Qt Quick comes with its own run-time and enables loading new functionality via modules, making it possible to develop applications built entirely with QML. However, the real strength of Qt Quick is the ability to integrate it into a C++ application.

For the most basic needs, for example integrating a QML view into a C++ project, the **QDeclarativeView** widget can be used. It is derived from a **QGraphicsView** but also includes the required components to host a QML application. Alternatively, you can simply make new C++ types available to the QML runtime through plugins, and these types can do anything your C++ application can do.

For more complex situations, how you proceed depends on from how the C++ code is built.

If you are starting with an application based on C++ widgets, you can reuse all your graphics assets and re-factor the **QWidgets** to QML. Since you already have the full interaction and design work done, developing and coding the QML is relatively easy.

If you are instead starting with an application based on a QGraphicsView, the conversion process is even easier and can proceed in stages. The entire QML application can be inserted into an existing graphics view by instantiating a QML engine. If desired, the QML interface can co-exist with the existing user interface and allow the conversion process to proceed in steps.

Snippet 14 shows the three steps required to add a QML engine and context to an existing **QGraphicsView**. First create an environment for instantiating QML components using the QDeclarativeEngine class, then encapsulate a QML component definition using QDeclarativeComponent. Finally, the resulting QGraphicsObject can be added to the existing scene and co-exist with the rest of the user interface.

```
QGraphicsScene *scene = ...;

QDeclarativeEngine *engine = new QDeclarativeEngine;
QDeclarativeComponent component(engine, QUrl::fromLocalFile(...));
QGraphicsObject *object =
    qobject_cast<QGraphicsObject*>(component.create());

scene->addItem(object);
```

Snippet 14: Adding a QML engine to QGraphicsView

If the component fails to load the QML file, the `error` property will be set to true. To output the error messages, the following statement can be placed just after the call to `create()`.

```
qWarning() << component.errors();
```

> **Tech note:** The syntax above is only available if you include `<QtDebug>` in your project. See documentation about qWarning() for details.

To align the user interfaces it is possible to transform the **QGraphicsObject** and adjust the z-value to place it at the right depth in the scene. In order to achieve the optimal performance for the QML part of the user interface, the following options are recommended to be set.

```
QGraphicsView *view = ...;

view->setOptimizationFlags(QGraphicsView::DontSavePainterState);
view->setViewportUpdateMode(
    QGraphicsView::BoundingRectViewportUpdate);
view->setItemIndexMethod(QGraphicsScene::NoIndex);
```

Snippet 15: Optimizing QML interface performance

Although combining an existing graphics view-based user interface with QML is possible, it is recommended to convert the entire experience to Qt Quick.

3.1. Sharing data between C++ and QML

Qt Quick provides numerous ways to share data between C++ and QML with or without implementing a formal model-view design pattern. It is also possible to trigger calls to QML functions from C++ and vice-versa. In general, exposing a **QObject** will make all of its signals, slots and properties available to the QML environment.

All QML code executes within a context. The context keeps track of what data is available to different leaves and nodes in a QML object tree. Data is shared as context properties or context objects. A context property is simply a way to expose a given **QObject** through a given name. For example, to expose a **QColor** property named `frameColor` to QML, simply use the following snippet.

```
QDeclarativeContext *context = ...;
context->setContextProperty("frameColor", QColor(Qt::red));
```

This property can then be accessed from within the QML context as a global property, as shown below. Remember property values are bound, not assigned, in QML. This means you can alter the `frameColor` property from C++ and the change will be reflected in QML.

```
Rectangle {
    border.color: frameColor
}
```

It is possible to add multiple context properties to a **QDeclarativeContext** object, but as the list of properties climbs the readability of the code crumbles. Instead of setting each property individually, it is cleaner to gather all context properties into one **QObject** and set the single object as the context object instead.

Snippet 16 shows how to define the interface object `MyInterface` using the `setContextProperty()` method. The `Q_PROPERTY` macro defines the properties available within `MyInterface` to the Qt property system and sets notification signals, allowing subsequent bindings to work.

Note that all properties added explicitly by QDeclarativeContext::setContextProperty() take precedence over the context object's default properties.

```
class MyInterface : ... {
    ...
    Q_PROPERTY(QAbstractItemModel *myModel READ model NOTIFY modelChanged)
    ...
};

MyInterface *myInterface = new MyInterface;
QDeclarativeEngine engine;
QDeclarativeContext *context = new
   QDeclarativeContext(engine.rootContext());
context->setContextObject(myDataSet);
QDeclarativeComponent component(&engine);
component.setData("import Qt 4.7\nListView { model: myModel }", QUrl());
component.create(context);
```

Snippet 16: Defining an interface using setContextProperty()

3.2. QML views into C++ models

Object properties work well when providing a limited set of values to QML, but are difficult to manage when large data sets are involved. In these cases formal models are visualized with formal views. This model/view design pattern allows developers to separate the implementation of user interface from business logic, supporting the model-view design pattern. The model can be implemented in C++ while the view is coded in QML.

QML can create views into C++ models that are exposed using the QAbstractItemModel interface.

To expose a **QAbstractItemModel** to QML a context property is used:

```
QAbstractItemModel *model = ...;
context->setContextProperty("dataModel", model);
```

> **Tech note:** Within Qt, the **QAbstractItemModel** class provides the abstract interface for item model classes. The **QAbstractItemModel** class defines the standard interface that item models must use to be able to interoperate with other components in the model/view architecture. It is not supposed to be instantiated directly. Instead, you should subclass it to create new models.
>
> The **QAbstractItemModel** class is one of the Model/View Classes and is part of Qt's model/view framework.

3.3. QML / C++ program flow

Qt Quick allows QML to call C++ methods and allows C++ signals to be handled by JavaScript expressions within the QML context.

3.3.1. Calling C++ methods from QML

In order to feedback data from the user to the business logic, QML must be able to call C++ methods. This is achieved through slots or `Q_INVOKABLE` methods. By providing QML access to a **QObject** as a context property, the slots and invokable methods of that class can be called from QML.

For example, the following **QObject** derived class is added to the QML context.

```
class CallableClass : public QObject
{
    Q_OBJECT
    ...
public slots:
    void cppMethod() { qDebug("C++ method called!"); }
};

...

context->setContextProperty("cppObject", new CallableClass);
```

The QML code then can refer to the cppMethod method using the cppObject global object. In this example the method in question returns no value nor accepts any arguments, but that is not a limitation of QML. Both return values and arguments of the types supported by QML are supported.

```
MouseArea {
    ...
    onClicked: {
        cppObject.cppMethod();
    }
}
```

3.3.2. Qt signal to QML handler

Qt C++ signals can be handled by JavaScript executing in a QML context. For instance, the `CallableClass` class from the previous example also declares a signal, `cppSignal()`.

```
class CallableClass : public QObject
{
    Q_OBJECT
    ...
signals:
    void cppSignal();
};
```

Using a `Connections` QML element, a signal handler can be implemented in QML. The connections element can be used to handle signals for any target object, including other QML elements. The signal handler is called `onSignalName`, where the first letter of the signal name is capitalized.

```
Connections {
    target: cppObject
    onCppSignal: { console.log("QML function called!"); }
}
```

3.4. Extending QML from C++

QML has built-in support for an extensive set of element types, but when application-specific needs pop up it is possible to extend QML with custom element types built in C++. For example, let's say you have a burning desire for a QML element called `Person` with properties `name` and `shoeSize`.

All QML elements map to C++ types. Snippet 17 declares a basic C++ class `Person` with the two properties we want accessible on the QML type – `name` and `shoeSize`. Although in this example we use the same name for the C++ class as the QML element, the C++ class can be named differently, or appear in a namespace.

```
class Person : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString name READ name WRITE setName)
    Q_PROPERTY(int shoeSize READ shoeSize WRITE setShoeSize)

public:
    Person(QObject *parent = 0);

    QString name() const;
    void setName(const QString &);

    int shoeSize() const;
    void setShoeSize(int);

private:
    QString m_name;
    int m_shoeSize;
};
```

Snippet 17: Declare a Person class

```
Person::Person(QObject *parent)
: QObject(parent), m_shoeSize(0)
{
}

QString Person::name() const
{
    return m_name;
}

void Person::setName(const QString &n)
{
    m_name = n;
}

int Person::shoeSize() const
{
    return m_shoeSize;
}

void Person::setShoeSize(int s)
{
    m_shoeSize = s;
}
```

Snippet 18: Define the Person class

The `Person` class implementation is quite basic. The property accessors simply return members of the object instance.

The `main.cpp` file also calls the qmlRegisterType() function to register the `Person` type with QML as a type in the `People` library version 1.0, and defines the mapping between the C++ and QML class names.

The `Person` type can now be used from QML:

```
import People 1.0

Person {
    name: "Bob Jones"
    shoeSize: 12
}
```

4. Getting started

This paper provided a brief introduction to Qt Quick. There is plenty more information, tutorials, and code examples for you to explore.

For additional information about Qt Quick:

- Try the QML Tutorial at http://doc.qt.nokia.com/4.7/qml-tutorial.html
- Full documentation of QML elements is online at http://doc.qt.nokia.com/4.7/qdeclarativeelements.html

To begin working with Qt Quick:

- Download and install snapshots of the latest Qt Creator IDE 2.1, which previews a QML Text Editor with code completion, syntax highlighting and context-sensitive help; a QML Visual Editor that was built from the ground up using QML; and a QML Debugger that allows you to inspect the QML item tree and its properties at runtime, to check frame rates, to evaluate JavaScript expressions and so on inside Qt Creator. Qt Creator 2.1 is scheduled for release later in 2010.
- Once you install Qt Creator, check out the included examples installed in the `YourInstalledRoot/examples/declarative` directory and exposed through the IDE.
- You will find an online discussions and wikis covering Qt Quick at http://www.forum.nokia.com/Community and at http://developer.qt.nokia.com/

About Qt

Qt is a cross-platform application framework. Using Qt, you can develop applications and user interfaces once, and deploy them across many desktop and embedded operating systems without rewriting the source code. Qt Development Frameworks, formerly Trolltech, was acquired by Nokia in June 2008. For more details about Qt please visit http://qt.nokia.com.

About Nokia

Nokia is the world leader in mobility, driving the transformation and growth of the converging Internet and communications industries. We make a wide range of mobile devices with services and software that enable people to experience music, navigation, video, television, imaging, games, business mobility and more. Developing and growing our offering of consumer Internet services, as well as our enterprise solutions and software, is a key area of focus. We also provide equipment, solutions and services for communications networks through Nokia Siemens Networks.

Code less.
Create more.
Deploy everywhere.

NOKIA

Oslo, Norway +47 21 60 48 00 • Redwood City, CA, USA +1 650-551-1676 • qt-info@nokia.com www.qt.nokia.com