# Integrating QML with C++

## Qt Essentials - Training Course

Produced by Nokia, Qt Development Frameworks

*Material based on Qt 4.7, created on January 18, 2011*

http://qt.nokia.com

# Module: Integrating QML with C++

- Declarative Environment
- Custom Items and Properties
- Signals & Slots and Methods
- Using Custom Types

# Objectives

- The QML runtime environment
  - understanding of the basic architecture
  - ability to set up QML in a C++ application
- Exposing C++ objects to QML
  - knowledge of the Qt features that can be exposed
  - familiarity with the mechanisms used to expose objects

# Module: Integrating QML with C++

- Declarative Environment
- Custom Items and Properties
- Signals & Slots and Methods
- Using Custom Types

# Overview

Qt Quick is a combination of technologies:

- A set of components, some graphical
- A declarative language: QML
  - based on JavaScript
  - running on a virtual machine
- A C++ API for managing and interacting with components
  - the **QtDeclarative** module

# Setting up a Declarative View

```
#include <QApplication>
#include <QDeclarativeView>
#include <QUrl>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QDeclarativeView view;
    view.setSource(QUrl("qrc:files/animation.qml"));
    view.show();
    return app.exec();
}
```

Demo qml-cpp-integration/ex-simpleviewer

# Setting up QtDeclarative

```
QT       += declarative
RESOURCES = simpleviewer.qrc
SOURCES   = main.cpp
```

# Module: Integrating QML with C++

- Declarative Environment
- Custom Items and Properties
- Signals & Slots and Methods
- Using Custom Types

# Overview

Interaction between C++ and QML occurs via objects exposed to the QML environment as new types.

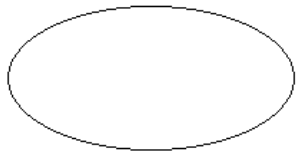
- Non-visual types are `QObject` subclasses
- Visual types (items) are `QDeclarativeItem` subclasses
  - `QDeclarativeItem` is the C++ equivalent of `Item`

To define a new type:

- In C++: Subclass either `QObject` or `QDeclarativeItem`
- In C++: Register the type with the QML environment
- In QML: Import the module containing the new item
- In QML: Use the item like any other standard item

# A Custom Item

In the *ellipse1.qml* file:



```
import QtQuick 1.0
import Shapes 1.0

Item {
    width: 300; height: 200

    Ellipse {
        x: 50; y: 50
        width: 200; height: 100
    }
}
```

- A custom ellipse item
  - instantiated using the `Ellipse` element
  - from the custom `Shapes` module
- Draws an ellipse with the specified geometry

# Declaring the Item

In the *ellipseitem.h* file:

```
#include <QDeclarativeItem>

class EllipseItem : public QDeclarativeItem
{
    Q_OBJECT
public:
    EllipseItem(QDeclarativeItem *parent = 0);
    void paint(QPainter *painter,
               const QStyleOptionGraphicsItem *option,
               QWidget *widget = 0);
};
```

- `EllipseItem` is a `QDeclarativeItem` subclass
- As with Qt widgets, each item can have a parent
- Each custom item needs to paint itself

# Implementing an Item

In the *ellipseitem.cpp* file:

```
#include <QtGui>
#include "ellipseitem.h"

EllipseItem::EllipseItem(QDeclarativeItem *parent)
    : QDeclarativeItem(parent)
{
    setFlag(QGraphicsItem::ItemHasNoContents, false);
}
...
```

- As usual, call the base class's constructor
- It is necessary to clear the `ItemHasNoContents` flag

# Implementing an Item (Continued)

In the *ellipseitem.cpp* file:

```
...
void EllipseItem::paint(QPainter *painter,
    const QStyleOptionGraphicsItem *option, QWidget *widget)
{
    painter->drawEllipse(option->rect);
}
```

- A simple `paint()` function implementation
- Ignore style information and use the default pen

# Registering the Item

```
#include <QApplication>
#include <QDeclarativeView>
#include "ellipseitem.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    qmlRegisterType<EllipseItem>("Shapes", 1, 0, "Ellipse");

    QDeclarativeView view;
    view.setSource(QUrl("qrc:files/ellipse1.qml"));
    view.show();
    return app.exec();
}
```

- Custom item registered as a module and element
- Automatically available to the `ellipse1.qml` file

# Reviewing the Registration

```
qmlRegisterType<EllipseItem>("Shapes", 1, 0, "Ellipse");
```

- This registers the `EllipseItem` C++ class
- Available from the `Shapes` QML module
  - version 1.0 (first number is major; second is minor)
- Available as the `Ellipse` element
  - the `Ellipse` element is an visual item
  - a subtype of `Item`

Demo qml-cpp-integration/ex-simple-item

# Adding Properties

In the *ellipse2.qml* file:

```
import QtQuick 1.0
import Shapes 2.0

Item {
    width: 300; height: 200

    Ellipse {
        x: 50; y: 50
        width: 200; height: 100
        color: "blue"
    }
}
```

- A new `color` property

Demo qml-cpp-integration/ex-properties

# Declaring a Property

In the *ellipseitem.h* file:

```
class EllipseItem : public QDeclarativeItem
{
    Q_OBJECT
    Q_PROPERTY(QColor color READ color WRITE setColor
               NOTIFY colorChanged)
    ...
```

- Use a `Q_PROPERTY` macro to define a new property
  - named `color` with `QColor` type
  - with getter and setter, `color()` and `setColor()`
  - emits the `colorChanged()` signal when the value changes
- The signal is just a notification
  - it contains no value
  - we must emit it to make property bindings work

# Declaring Getter and Setter

In the *ellipseitem.h* file:

```
public:
    ...
    const QColor &color() const;
    void setColor(const QColor &newColor);

signals:
    void colorChanged();

private:
    QColor m_color;
};
```

- Declare the getter and setter
- Declare the private variable containing the color

# Implementing Getter and Setter

In the *ellipseitem.cpp* file:

```
const QColor &EllipseItem::color() const {
    return m_color;
}

void EllipseItem::setColor(const QColor &newColor) {
  // always check if new value differs from current
  if (m_color != newColor) {
    m_color = newColor; // set the new value
    update(); // update painting
    emit colorChanged(); // notify about changes
  }
}
```

# Updated Paint

In the *ellipseitem.cpp* file:

```
void EllipseItem::paint(QPainter *painter,
    const QStyleOptionGraphicsItem *option, QWidget *widget) {
    painter->save(); // save painter state
    painter->setPen(Qt::NoPen); // we use no pen
    // setup brush as described by property 'color'
    painter->setBrush(m_color);
    painter->drawEllipse(option->rect);
    painter->restore(); // restore painter state
}
```

# Summary of Items and Properties

- Register new QML types using `qmlRegisterType`
  - new types are subclasses of `QDeclarativeItem`
- Add QML properties
  - define C++ properties with `NOTIFY` signals
  - notifications are used to maintain the bindings between items

# Module: Integrating QML with C++

- Declarative Environment
- Custom Items and Properties
- Signals & Slots and Methods
- Using Custom Types

# Adding Signals

In the *ellipse3.qml* file:


```
import QtQuick 1.0
import Shapes 3.0

Item {
    width: 300; height: 200

    Ellipse {
        x: 50; y: 35; width: 200; height: 100
        color: "blue"
        onReady: label.text = "Ready"
    }

    ...
```

- A new `onReady` signal handler
  - changes the `text` property of the `label` item

# Declaring a Signal

In the *ellipseitem.h* file:

```
...
signals:
    void colorChanged();
    void ready();
...
```

- Add a `ready()` signal
  - this will have a corresponding `onReady` handler in QML
  - we will emit this 2 seconds after the item is created

# Emitting the Signal

In the *ellipseitem.cpp* file:

```
EllipseItem::EllipseItem(QDeclarativeItem *parent)
    : QDeclarativeItem(parent)
{
    setFlag(QGraphicsItem::ItemHasNoContents, false);
    QTimer::singleShot(2000, this, SIGNAL(ready()));
}
```

- Change the constructor
  - start a 2 second single-shot timer
  - this emits the `ready()` signal when it times out

# Handling the Signal

In the *ellipse3.qml* file:

```
Ellipse {
    x: 50; y: 35; width: 200; height: 100
    color: "blue"
    onReady: label.text = "Ready"
}

Text {
    ...
    text: "Not ready"
}
```


- In C++:
  - the `EllipseItem::ready()` signal is emitted
- In QML:
  - the `Ellipse` item's `onReady` handler is called
  - sets the text of the `Text` item

# Adding Methods to Items

Two ways to add methods that can be called from QML:

1. Create C++ slots
   - automatically exposed to QML
   - useful for methods that do not return values
2. Mark regular C++ functions as invokable
   - allows values to be returned

# Adding Slots

In the *ellipse4.qml* file:

```
Ellipse {
    x: 50; y: 35; width: 200; height: 100
    color: "blue"
    onReady: label.text = "Ready"

    MouseArea {
        anchors.fill: parent
        onClicked: parent.setColor("darkgreen");
    }
}
```


- `Ellipse` now has a `setColor()` method
  - accepts a color
  - also accepts strings containing colors
- Normally, could just use properties to change colors...

# Declaring a Slot

In the *ellipseitem.h* file:

```
...
    const QColor &color() const;

signals:
    void colorChanged();
    void ready();

public slots:
    void setColor(const QColor &newColor);
...
```

- Moved the `setColor()` setter function
  - now in the `public slots` section of the class
- Accepts `QColor` values
  - the string passed from QML is converted to a color
  - for custom types, make sure that type conversion is supported

# Adding Methods

In the *ellipse5.qml* file:

```
Ellipse {
    x: 50; y: 35; width: 200; height: 100
    color: "blue"
    onReady: label.text = "Ready"

    MouseArea {
        anchors.fill: parent
        onClicked: parent.color = parent.randomColor()
    }
}
```


- `Ellipse` now has a `randomColor()` method
  - obtain a random color using this method
  - set the color using the `color` property

# Declaring a Method

In the *ellipseitem.h* file:

```
...
public:
    EllipseItem(QDeclarativeItem *parent = 0);
    void paint(QPainter *painter,
      const QStyleOptionGraphicsItem *option, QWidget *widget = 0);

    const QColor &color() const;
    void setColor(const QColor &newColor);

    Q_INVOKABLE QColor randomColor() const;
...
```

- Define the `randomColor()` function
  - add the `Q_INVOKABLE` macro before the declaration
  - returns a `QColor` value
  - *cannot* return a const reference

# Implementing a Method

In the *ellipseitem.cpp* file:

```
QColor EllipseItem::randomColor() const
{
    return QColor(qrand() & 0xff, qrand() & 0xff, qrand() & 0xff);
}
```

- Define the new `randomColor()` function
  - the pseudo-random number generator has already been seeded
  - simply return a color
  - do not use the `Q_INVOKABLE` macro in the source file

# Summary of Signals, Slots and Methods

- Define signals
  - connect to Qt signals with the `onSignal` syntax
- Define QML-callable methods
  - reuse slots as QML-callable methods
  - methods that return values are marked using `Q_INVOKABLE`

# Module: Integrating QML with C++

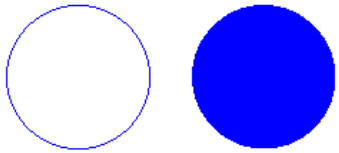
- Declarative Environment
- Custom Items and Properties
- Signals & Slots and Methods
- Using Custom Types

# Defining Custom Property Types

- Items can be used as property types
  - allows rich description of properties
  - subclass `QDeclarativeItem` (as before)
  - requires registration of types (as before)
- A simpler way to define custom property types:
  - use simple enums and flags
  - easy to declare and use
- Collections of custom types:
  - define a new custom item
  - use with a `QDeclarativeListProperty` template type

# Enums as Property Types

```
import QtQuick 1.0
import Shapes 6.0

Item {
    width: 300; height: 150

    Ellipse {
        x: 35; y: 25; width: 100; height: 100
        color: "blue"
        style: Ellipse.Outline
    }

    Ellipse {
        x: 165; y: 25; width: 100; height: 100
        color: "blue"
        style: Ellipse.Filled
    }
}
```



- A new `style` property with a custom value type

# Declaring Enums as Property Types

In the *ellipseitem.h* file:

```
class EllipseItem : public QDeclarativeItem
{
    Q_OBJECT
    Q_PROPERTY(QColor color READ color WRITE setColor
               NOTIFY colorChanged)
    Q_PROPERTY(Style style READ style WRITE setStyle
               NOTIFY styleChanged)
    Q_ENUMS(Style)

public:
    enum Style { Outline, Filled };
...
```

- Declare the `style` property using the `Q_PROPERTY` macro
- Declare the `Style` enum using the `Q_ENUMS` macro
- Define the `Style` enum

# Using Enums in Getters and Setters

In the *ellipseitem.h* file:

```
...
    const Style &style() const;
    void setStyle(const Style &newStyle);

signals:
    void colorChanged();
    void styleChanged();

private:
    QColor m_color;
    Style m_style;
...
```

- Define the usual getter, setter, signal and private member for the `style` property

# Using Enums in Getters and Setters

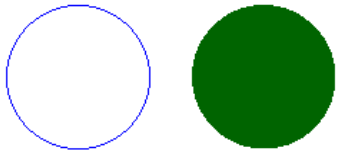
In the *ellipseitem.cpp* file:

```
const EllipseItem::Style &EllipseItem::style() const
{
    return m_style;
}

void EllipseItem::setStyle(const Style &newStyle)
{
    if (m_style != newStyle) {
        m_style = newStyle;
        update();
        emit styleChanged();
    }
}
```

- Similar getter and setter as used for the `color` property

# Custom Items as Property Types

```
import QtQuick 1.0
import Shapes 7.0

Item {
    width: 300; height: 150

    Ellipse {
        x: 35; y: 25; width: 100; height: 100
        style: Style { color: "blue"
                       filled: false }
    }

    Ellipse {
        x: 165; y: 25; width: 100; height: 100
        style: Style { color: "darkgreen"
                       filled: true }
    }
}
```



- A new `style` property with a custom item type

# Declaring the Style Item

In the *style.h* file:

```
class Style : public QDeclarativeItem
{
    Q_OBJECT
    Q_PROPERTY(QColor color READ color WRITE setColor
               NOTIFY colorChanged)
    Q_PROPERTY(bool filled READ filled WRITE setFilled
               NOTIFY filledChanged)

public:
    Style(QDeclarativeItem *parent = 0);
...
```

- `Style` is a `QDeclarativeItem` subclass
- Defined like the `EllipseItem` class
  - implemented in the same way

# Declaring the Style Property

In the *ellipseitem.h* file:

```
class EllipseItem : public QDeclarativeItem
{
    Q_OBJECT
    Q_PROPERTY(Style *style READ style WRITE setStyle
               NOTIFY styleChanged)
public:
...
    Style *style() const;
    void setStyle(Style *newStyle);

signals:
    void styleChanged();
...
```

- Declare the `style` property
  - with the `Style` pointer as its type
  - declare the associated getter, setter and signal

# Using the Style Property

In the *ellipseitem.cpp* file:

```
void EllipseItem::paint(QPainter *painter,
    const QStyleOptionGraphicsItem *option, QWidget *widget)
{
    painter->save();
    if (!m_style->filled()) {
        painter->setPen(m_style->color());
        painter->setBrush(Qt::NoBrush);
    } else {
        painter->setPen(Qt::NoPen);
        painter->setBrush(m_style->color());
    }
    painter->drawEllipse(option->rect);
    painter->restore();
}
```

- Use the internal `Style` object to decide what to paint
  - using the getters to read its `color` and `filled` properties

# Registering the Style Property

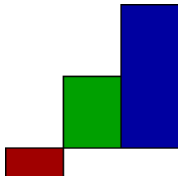
In the *ellipseitem.cpp* file:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    qmlRegisterType<EllipseItem>("Shapes", 7, 0, "Ellipse");
    qmlRegisterType<Style>("Shapes", 7, 0, "Style");
...
```

- Register the `Style` class
  - in the same way as the `EllipseItem` class
  - in the same module: `Shapes` 7.0

# Collections of Custom Types

```
import QtQuick 1.0
import Shapes 8.0

Chart {
    width: 120; height: 120
    bars: [
        Bar { color: "#a00000"
              value: -20 },
        Bar { color: "#00a000"
              value: 50 },
        Bar { color: "#0000a0"
              value: 100 }
    ]
}
```



- A `Chart` item
  - with a `bars` list property
  - accepting custom `Bar` items

# Declaring the List Property

In the *chartitem.h* file:

```
class BarItem;

class ChartItem : public QDeclarativeItem
{
    Q_OBJECT
    Q_PROPERTY(QDeclarativeListProperty<BarItem> bars READ bars
               NOTIFY barsChanged)
public:
    ChartItem(QDeclarativeItem *parent = 0);
    void paint(QPainter *painter,
      const QStyleOptionGraphicsItem *option, QWidget *widget = 0);
...
```

- Define the `bars` property
  - in theory, read-only but with a notification signal
  - in reality, writable as well as readable

# Declaring the List Property

In the *chartitem.h* file:

```
...
    QDeclarativeListProperty<BarItem> bars();

signals:
    void barsChanged();

private:
    static void append_bar(QDeclarativeListProperty<BarItem> *list,
                           BarItem *bar);
    QList<BarItem*> m_bars;
};
```

- Define the getter function and notification signal
- Define an append function for the list property

# Defining the Getter Function

In the *chartitem.cpp* file:

```
QDeclarativeListProperty<BarItem> ChartItem::bars()
{
    return QDeclarativeListProperty<BarItem>(this, 0,
        &ChartItem::append_bar);
}
```

- Defines and returns a list of `BarItem` objects
  - with an append function

# Defining the Append Function

```
void ChartItem::append_bar(QDeclarativeListProperty<BarItem> *list,
                           BarItem *bar)
{
    ChartItem *chart = qobject_cast<ChartItem *>(list->object);
    if (chart) {
        bar->setParentItem(chart);
        chart->m_bars.append(bar);
        chart->barsChanged();
    }
}
```

- Static function, accepts
  - the list to operate on
  - each `BarItem` to append
- When a `BarItem` is appended
  - emits the `barsChanged()` signal

# Summary of Custom Property Types

- Define items as property types:
  - declare and implement a new `QDeclarativeItem` subclass
  - declare properties to use a pointer to the new type
  - register the item with `qmlRegisterType`
- Use enums as simple custom property types:
  - use `Q_ENUMS` to declare a new enum type
  - declare properties as usual
- Define collections of custom types:
  - using a custom item that has been declared and registered
  - declare properties with `QDeclarativeListProperty`
  - implement a getter and an append function for each property
  - read-only properties, but read-write containers
  - read-only containers define append functions that simply return

# Creating Extension Plugins

- Declarative extensions can be deployed as plugins
  - using source and header files for a working custom type
  - developed separately then deployed with an application
  - write QML-only components then rewrite in C++
  - use placeholders for C++ components until they are ready
- Plugins can be loaded by the `qmlviewer` tool
  - with an appropriate `qmldir` file
- Plugins can be loaded by C++ applications
  - some work is required to load and initialize them

# Defining an Extension Plugin

```
#include <QDeclarativeExtensionPlugin>

class EllipsePlugin : public QDeclarativeExtensionPlugin
{
    Q_OBJECT

public:
    void registerTypes(const char *uri);
};
```

- Create a `QDeclarativeExtensionPlugin` subclass
  - only one function to reimplement

# Implementing an Extension Plugin

```
#include <qdeclarative.h>
#include "ellipseplugin.h"
#include "ellipseitem.h"

void EllipsePlugin::registerTypes(const char *uri)
{
    qmlRegisterType<EllipseItem>(uri, 9, 0, "Ellipse");
}

Q_EXPORT_PLUGIN2(ellipseplugin, EllipsePlugin);
```

- Register the custom type using the `uri` supplied
  - the same custom type we started with
- Export the plugin using
  - the name of the plugin library (`ellipseplugin`)
  - the name of the plugin class (`EllipsePlugin`)

# Building an Extension Plugin

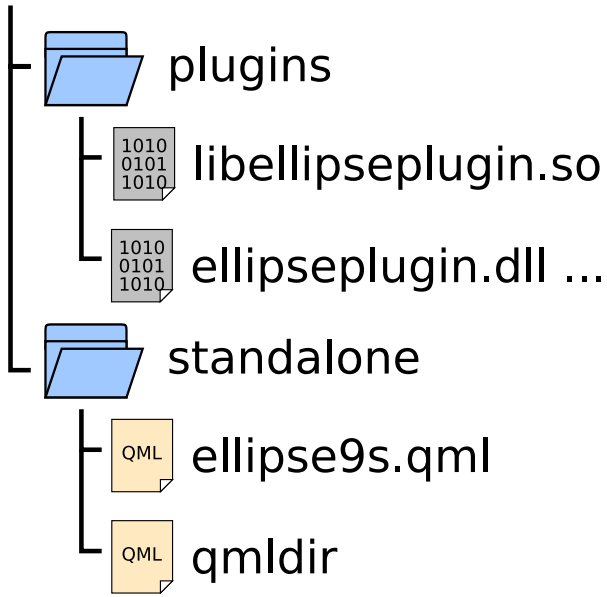
```
TEMPLATE  = lib
CONFIG   += qt plugin
QT       += declarative

HEADERS  += ellipseitem.h \
            ellipseplugin.h

SOURCES  += ellipseitem.cpp \
            ellipseplugin.cpp

DESTDIR   = ../plugins
```

- Ensure that the project is built as a Qt plugin
- Declarative module is added to the Qt configuration
- Plugin is written to a `plugins` directory

# Using an Extension Plugin

To use the plugin with the `qmlviewer` tool:

- Write a `qmldir` file
  - include a line to describe the plugin
  - stored in the `standalone` directory
- Write a QML file to show the item
  - `ellipse9s.qml`




The `qmldir` file contains a declaration:

```
plugin ellipseplugin ../plugins
```

- **`plugin`** followed by
  - the plugin name: **`ellipseplugin`**
  - the plugin path relative to the `qmldir` file: **`../plugins`**

# Using an Extension Plugin

In the `ellipse9s.qml` file:

```
import QtQuick 1.0

Item {
    width: 300; height: 200

    Ellipse {
        x: 50; y: 50
        width: 200; height: 100
    }
}
```

- Use the custom item directly
- No need to import any custom modules
  - `qmldir` and `ellipse9s.qml` are in the same project directory
  - `Ellipse` is automatically imported into the global namespace

# Loading an Extension Plugin

To load the plugin in a C++ application:

- Locate the plugin
  - (perhaps scan the files in the `plugins` directory)
- Load the plugin with `QPluginLoader`

```
QPluginLoader loader(pluginsDir.absoluteFilePath(fileName));
```

- Cast the plugin object to a `QDeclarativeExtensionPlugin`

```
QDeclarativeExtensionPlugin *plugin =
    qobject_cast<QDeclarativeExtensionPlugin *>(loader.instance());
```

- Register the extension with a URI

```
if (plugin)
    plugin->registerTypes("Shapes");
```

  - in this example, `Shapes` is used as a URI

# Using an Extension Plugin

In the `ellipse9s.qml` file:

```
import QtQuick 1.0
import Shapes 9.0

Item {
    width: 300; height: 200

    Ellipse {
        x: 50; y: 50
        width: 200; height: 100
    }
}
```

- The `Ellipse` item is part of the `Shapes` module
- A different URI makes a different import necessary; e.g.,

```
plugin->registerTypes("com.nokia.qt.examples.Shapes");
```

- corresponds to

```
import com.nokia.qt.examples.Shapes 9.0
```

# Summary of Extension Plugins

- Extensions can be compiled as plugins
  - define and implement a `QDeclarativeExtensionPlugin` subclass
  - define the version of the plugin in the extension
  - build a Qt plugin project with the `declarative` option enabled
- Plugins can be loaded by the `qmlviewer` tool
  - write a `qmldir` file
  - declare the plugin's name and location relative to the file
  - no need to import the plugin in QML
- Plugins can be loaded by C++ applications
  - use `QPluginLoader` to load the plugin
  - register the custom types with a specific URI
  - import the same URI and plugin version number in QML