Integrating QML with C++

Training Course

Visit us at `http://qt.digia.com`

Produced by Digia Plc.

Material based on Qt 5.0, created on September 27, 2012

Digia Plc.

Module: Integrating QML with C++

- Declarative Environment
- Exporting C++ objects to QML
- Exporting Classes to QML
  - Exporting Non-GUI Classes
  - Exporting QPainter based GUI Classes
  - Exporting Scene Graph based GUI Classes
- Using Custom Types
- Plug-ins

digia

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

Demo qml-cpp-integration/ex-clock

digia

- Declarative Environment
- Exporting C++ objects to QML
- Exporting Classes to QML
  - Exporting Non-GUI Classes
  - Exporting QPainter based GUI Classes
  - Exporting Scene Graph based GUI Classes
- Using Custom Types
- Plug-ins

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 **QtQuick** module

Setting up a QtQuick View

```
#include <QGuiApplication>
#include <QQuickView>
#include <QUrl>

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

Demo qml-cpp-integration/ex-simpleviewer

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

- Declarative Environment
- Exporting C++ objects to QML
- Exporting Classes to QML
  - Exporting Non-GUI Classes
  - Exporting QPainter based GUI Classes
  - Exporting Scene Graph based GUI Classes
- Using Custom Types
- Plug-ins

Exporting C++ objects to QML

- C++ objects can be exported to QML

```
class User : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString name READ name WRITE setName
               NOTIFY nameChanged)
    Q_PROPERTY(int age READ age WRITE setAge NOTIFY ageChanged)

public:
    User(const QString &name, int age, QObject *parent = 0);
    ...
}
```

- The notify signal is needed for correct property bindings!
- Q_PROPERTY must be at top of class

- `QQmlContext` exports the instance to QML.

```
void main( int argc, char* argv[] ) {
    ...
    User *currentUser = new User("Alice", 29);

    QAbstractItemModel *thingsModel = createModel();

    QQuickView *view = new QQuickView;
    QQmlContext *context = view->engine()->rootContext();

    context->setContextProperty("_currentUser", currentUser);
    context->setContextProperty("_favoriteThings", thingsModel);

    ...
}
```

- Use the instances like any other QML object

```
Text {
    text : _currentUser.name
    ...
}
ListView {
    model : _favoriteThings
    ...
}
```

Accessible from QML:

- Properties
- Signals
- Slots
- Methods marked with Q_INVOKABLE
- Enums registered with Q_ENUMS

```
class Circle {
  Q_ENUMS(Style)

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

- Declarative Environment
- Exporting C++ objects to QML
- Exporting Classes to QML
  - Exporting Non-GUI Classes
  - Exporting QPainter based GUI Classes
  - Exporting Scene Graph based GUI Classes
- Using Custom Types
- Plug-ins

Steps to define a new type in QML:

- In C++: Subclass either `QObject` or `QQuickItem`
- 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

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

Step 1: Implementing the Class

```
#include <QObject>

class QTimer;

class Timer : public QObject
{
    Q_OBJECT
public:
    Timer(QObject *parent = 0);

private:
    QTimer* m_timer;
};
```

Implementing the Class

- `Timer` is a `QObject` subclass
- As with all `QObjects`, each item can have a parent
- Non-GUI custom items do not need to worry about any painting

Step 1: Implementing the Class

```
#include "timer.h"
#include <QTimer>

Timer::Timer(QObject *parent)
    : QObject(parent),
      m_timer(new QTimer(this))
{
    m_timer->setInterval( 1000 );
    m_timer->start();
}
```

Step 2: Registering the Class

```
#include <QGuiApplication>
#include <QQuickView>
#include "timer.h"

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    qmlRegisterType<Timer>("CustomComponents", 1, 0, "Timer");

    QQuickView view;
    view.setSource(QUrl("qrc:///main.qml"));
    view.show();
    return app.exec();
}
```

- Timer registered as an element in module "CustomComponents"
- Automatically available to the `main.qml` file

Reviewing the Registration

```
qmlRegisterType<Timer>("CustomComponents", 1, 0, "Timer");
```

- This registers the `Timer` C++ class
- Available from the `CustomComponents` QML module
  - version 1.0 (first number is major; second is minor)
- Available as the `Timer` element
  - the `Timer` element is an non-visual item
  - a subclass of `QObject`

In the *main.qml* file:

```
import QtQuick 2.0
import CustomComponents 1.0

Rectangle {
    width: 500
    height: 360

    Timer {
        id: timer
        ...

    }
}
```

Demo qml-cpp-integration/ex_simple_timer

In the *main.qml* file:

```
Rectangle {
    ...
    Timer {
        id: timer
        interval: 3000
    }
}
```

- A new `interval` property

Demo qml-cpp-integration/ex_timer_properties

In the *timer.h* file:

```
class Timer : public QObject
{
    Q_OBJECT
    Q_PROPERTY( int interval READ interval
                WRITE setInterval NOTIFY intervalChanged )
    ...
```

- Use a `Q_PROPERTY` macro to define a new property
  - named `interval` with `int` type
  - with getter and setter, `interval()` and `setInterval()`
  - emits the `intervalChanged()` 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, Setter and Signal

In the *timer.h* file:

```
public:
    ...
    void setInterval(int msec);
    int interval();

signals:
    void intervalChanged();
    ...

private:
    QTimer *m_timer;
};
```

- Declare the getter and setter
- Declare the notifier signal
- Contained QTimer object holds actual value

In the *timer.cpp* file:

```
void Timer::setInterval( int msec )
{
    if ( m_timer->interval() == msec )
        return;
    m_timer->stop();
    m_timer->setInterval( msec );
    m_timer->start();
    emit intervalChanged();
}

int Timer::interval()
{
    return m_timer->interval();
}
```

- Do not emit notifier signal if value does not actually change
- Important to break cyclic dependencies in property bindings

Summary of Items and Properties

- Register new QML types using `qmlRegisterType`
  - new non-GUI types are subclasses of `QObject`
- Add QML properties
  - define C++ properties with `NOTIFY` signals
  - notifications are used to maintain the bindings between items
  - *only* emit notifier signals if value actually changes

In the *main.qml* file:

```
Rectangle {
    ...
    Timer {
        interval: 3000
        onTimeout: {
            console.log( "Timer fired!" );
        }
    }
}
```

- A new `onTimeout` signal handler
  - outputs a message to stderr.

Demo qml-cpp-integration/ex_timer_signals

In the *timer.h* file:

```
...
signals:
    void timeout();
    void intervalChanged();
...
```

- Add a `timeout()` signal
  - this will have a corresponding `onTimeout` handler in QML
  - we will emit this whenever the contained QTimer object fires

In the *timer.cpp* file:

```
Timer::Timer(QObject *parent)
    : QObject(parent),
      m_timer(new QTimer(this))
{
    connect(m_timer, SIGNAL(timeout()),
            this, SIGNAL(timeout()));
}
```

- Change the constructor
- connect `QTimer::timeout()` signal to `Timer::timeout()` signal

In the *main.qml* file:

```
Rectangle {
    ...
    Timer {
        interval: 3000
        onTimeout: {
            console.log( "Timer fired!" );
        }
    }
}
```

- In C++:
  - the `QTimer::timeout()` signal is emitted
  - connection means `Timer::timeout()` is emitted
- In QML:
  - the `Timer` item's `onTimeout` handler is called
  - outputs message to stderr

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

In the *main.qml* file:

```
Rectangle {

    Timer {
        id: timer
        onTimeout: {
            console.log( "Timer fired!" );
        }
    }

    MouseArea {

        onClicked: {
            if ( timer.active == false ) {
                timer.start();
            } else {
                timer.stop();
            }
        }
    }
}
```

Adding Slots

- `Timer` now has `start()` and `stop()` methods
- Normally, could just use properties to change state...
- For example a `running` property

Demo qml-cpp-integration/ex_timer_slots

In the *timer.h* file:

```
...
public slots:
    void start();
    void stop();
...
```

- Added `start()` and `stop()` slots to public slots section
- No difference to declaring slots in pure C++ application

In the *timer.cpp* file:

```
void Timer::start() {
    if ( m_timer->isActive() )
        return;
    m_timer->start();
    emit activeChanged();
}

void Timer::stop() {
    if ( !m_timer->isActive() )
        return;
    m_timer->stop();
    emit activeChanged();
}
```

- Remember to emit notifier signal for any changing properties

In the *main.qml* file:

```
Rectangle {

    Timer {
        id: timer
        interval: timer.randomInterval( 500, 1500 )

        onTimeout: {
            console.log( "Timer fired!" );
        }
    }
}
```

- `Timer` now has a `randomInterval()` method
  - obtain a random interval using this method
  - accepts arguments for min and max intervals
  - set the interval using the `interval` property

Demo qml-cpp-integration/ex-methods

In the *timer.h* file:

```
...
public:
    explicit Timer( QObject* parent = 0 );
...
    Q_INVOKABLE int randomInterval( int min, int max ) const;
...
```

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

In the *timer.cpp* file:

```
int Timer::randomInterval( int min, int max ) const
{
    int range = max - min;
    int msec = min + qrand() % range;
    qDebug() << "Random interval =" << msec << "msecs";
    return msec;
}
```

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

Summary of Signals, Slots and Methods

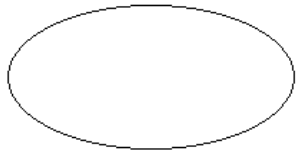
- Define signals
  - connect to Qt signals with the `onSignal` syntax
- Define QML-callable methods
  - reuse slots as QML-callable methods
  - methods that return values are marked using `Q_INVOKABLE`

Exporting a QPainter based GUI class

- Derive from `QQuickPaintedItem`
- Implement `paint(…)`
- Similar to non GUI classes:
  - Export object from C++
  - Import and use in QML
  - properties, signals/slots, Q_INVOKABLE

Exporting a QPainter based GUI class cont'd.

```
#include <QQuickPaintedItem>

class EllipseItem : public QQuickPaintedItem
{
    Q_OBJECT
public:
    EllipseItem(QQuickItem *parent = 0);
    void paint(QPainter *painter);
};
```



Exporting a QPainter based GUI class cont'd.

```
EllipseItem::EllipseItem(QQuickItem *parent)
    : QQuickPaintedItem(parent)
{
}

void EllipseItem::paint(QPainter *painter)
{
    const qreal halfPenWidth =
      qMax(painter->pen().width() / 2.0, 1.0);

    QRectF rect = boundingRect();
    rect.adjust(halfPenWidth, halfPenWidth,
                -halfPenWidth, -halfPenWidth);

    painter->drawEllipse(rect);
}
```

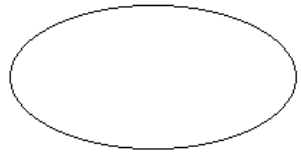
Exporting a QPainter based GUI class cont'd.

```
#include <QGuiApplication>
#include <QQuickView>
#include "ellipseitem.h"

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

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

Exporting a QPainter based GUI class cont'd.

In the *ellipse1.qml* file:

```
import QtQuick 2.0
import Shapes 1.0

Item {
    width: 300; height: 200

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



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

Exporting a Scene Graph based GUI class

- Derive from `QQuickItem`
- Implement `updatePaintNode(…)`
- Create and initialize a `QSGNode` subclass (e.g. `QSGGeometryNode`)
  - `QSGGeometry` to specify the mesh
  - `QSGMaterial` to specify the texture
- Similar to other classes:
  - Export object from C++
  - Import and use in QML
  - properties, signals/slots, Q_INVOKABLE

Exporting a Scene Graph based GUI class cont'd.

```
#include <QQuickItem>
#include <QSGGeometry>
#include <QSGFlatColorMaterial>

class TriangleItem : public QQuickItem
{
    Q_OBJECT
public:
    TriangleItem(QQuickItem *parent = 0);

protected:
    QSGNode *updatePaintNode(QSGNode *node, UpdatePaintNodeData *data);

private:
    QSGGeometry m_geometry;
    QSGFlatColorMaterial m_material;
};
```

Exporting a Scene Graph based GUI class cont'd.

```
#include <QSGGeometryNode>

TriangleItem::TriangleItem(QQuickItem *parent)
    : QQuickItem(parent),
      m_geometry(QSGGeometry::defaultAttributes_Point2D(), 3)
{
    setFlag(ItemHasContents);
    m_material.setColor(Qt::red);
}
```

Exporting a Scene Graph based GUI class cont'd.

```
QSGNode *TriangleItem::updatePaintNode(QSGNode *n, UpdatePaintNodeData *)
{
    QSGGeometryNode *node = static_cast<QSGGeometryNode *>(n);
    if (!node) node = new QSGGeometryNode();

    QSGGeometry::Point2D *v = m_geometry.vertexDataAsPoint2D();
    const QRectF rect = boundingRect();
    v[0].x = rect.left();
    v[0].y = rect.bottom();
    v[1].x = rect.left() + rect.width()/2;
    v[1].y = rect.top();
    v[2].x = rect.right();
    v[2].y = rect.bottom();
    node->setGeometry(&m_geometry);
    node->setMaterial(&m_material);
    return node;
}
```

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

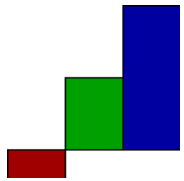
- Declarative Environment
- Exporting C++ objects to QML
- Exporting Classes to QML
  - Exporting Non-GUI Classes
  - Exporting QPainter based GUI Classes
  - Exporting Scene Graph based GUI Classes
- Using Custom Types
- Plug-ins

- Enums
- Custom types as property values

```
Timer {
    interval {
        duration: 2
        unit: IntervalSettings.Seconds
    }
}
```

- Collection of custom types

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



Defining Custom Property Types

- Custom classes can be used as property types
  - allows rich description of properties
  - subclass `QObject` or `QQuickItem` (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 `QQmlListProperty` template type

Using Enums

```
class IntervalSettings :public QObject
{
    Q_OBJECT

    Q_ENUMS( Unit )
    Q_PROPERTY( Unit unit READ unit ...)

public:
    enum Unit { Minutes, Seconds, MilliSeconds };
    ...
}
```

```
Timer {
    interval {
        duration: 2
        unit: IntervalSettings.Seconds
    }
}
```

Custom classes as Property Types

- Use the sub type as a pointer

```
class Timer : public QObject
{
    Q_OBJECT
    Q_PROPERTY( IntervalSettings* interval READ interval
                WRITE setInterval NOTIFY intervalChanged)

public:
    IntervalSettings* interval() const;
    void setInterval( IntervalSettings* );
    ...

private:
    IntervalSettings* m_settings;
}
```

Custom classes as Property Types cont'd.

- Instantiate `m_settings` to an instance rather than just a null pointer:

```
Timer::Timer(...)
    : m_settings(new IntervalSettings)
{
  ...
}
```

Custom classes as Property Types cont'd.

- Instantiating allow you this syntax:

```
Timer {
    interval {
        duration: 2
        unit: IntervalSettings.Seconds
    }
}
```

- Alternatively you would need this syntax:

```
Timer {
    interval: IntervalSettings {
        duration: 2
        unit: IntervalSettings.Seconds
    }
}
```

Custom classes as Property Types cont'd.

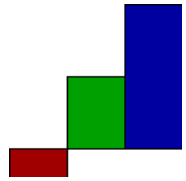
- Both classes must be exported to QML

```
qmlRegisterType<Timer>( "CustomComponents", 1, 0, "Timer" );
qmlRegisterType<IntervalSettings>( "CustomComponents",
                                   1, 0, "IntervalSettings");
```

Demo qml-cpp-integration/ex_timer_custom_types

Collections of Custom Types

```
import QtQuick 2.0
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

Demo qml-cpp-integration/ex-custom-collection-types

In the *chartitem.h* file:

```
class BarItem;

class ChartItem : public QQuickPaintedItem
{
    Q_OBJECT
    Q_PROPERTY(QQmlListProperty<BarItem> bars READ bars
               NOTIFY barsChanged)
public:
    ChartItem(QQuickItem *parent = 0);
    void paint(QPainter *painter);
...
```

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

In the *chartitem.h* file:

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

signals:
    void barsChanged();

private:
    static void append_bar(QQmlListProperty<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:

```
QQmlListProperty<BarItem> ChartItem::bars()
{
    return QQmlListProperty<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(QQmlListProperty<BarItem> *list,
                           BarItem *bar)
{
    ChartItem *chart = qobject_cast<ChartItem *>(list->object);
    if (chart) {
        bar->setParent(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 classes as property types:
  - declare and implement a new `QObject` or `QQuickItem` 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 `QQmlListProperty`
  - 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

Default Property

- One property can be marked as the default

```
class ChartItem : public QQuickPaintedItem
{
    Q_OBJECT
    Q_PROPERTY(QQmlListProperty<BarItem> bars READ bars
               NOTIFY barsChanged)
    Q_CLASSINFO("DefaultProperty", "bars")
...
```

- Allows child-item like syntax for assignment

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

- Declarative Environment
- Exporting C++ objects to QML
- Exporting Classes to QML
  - Exporting Non-GUI Classes
  - Exporting QPainter based GUI Classes
  - Exporting Scene Graph based GUI Classes
- Using Custom Types
- Plug-ins

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 `qmlscene` 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 <QQmlExtensionPlugin>

class EllipsePlugin : public QQmlExtensionPlugin
{
    Q_OBJECT
    Q_PLUGIN_METADATA(IID
      "org.qt-project.Qt.QQmlExtensionInterface/1.0")

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

- Create a `QQmlExtensionPlugin` subclass
  - add type information for Qt's plugin system
  - only one function to reimplement

Implementing an Extension Plugin

```
#include "ellipseplugin.h"
#include "ellipseitem.h"

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

- Register the custom type using the `uri` supplied
  - the same custom type we started with

Building an Extension Plugin

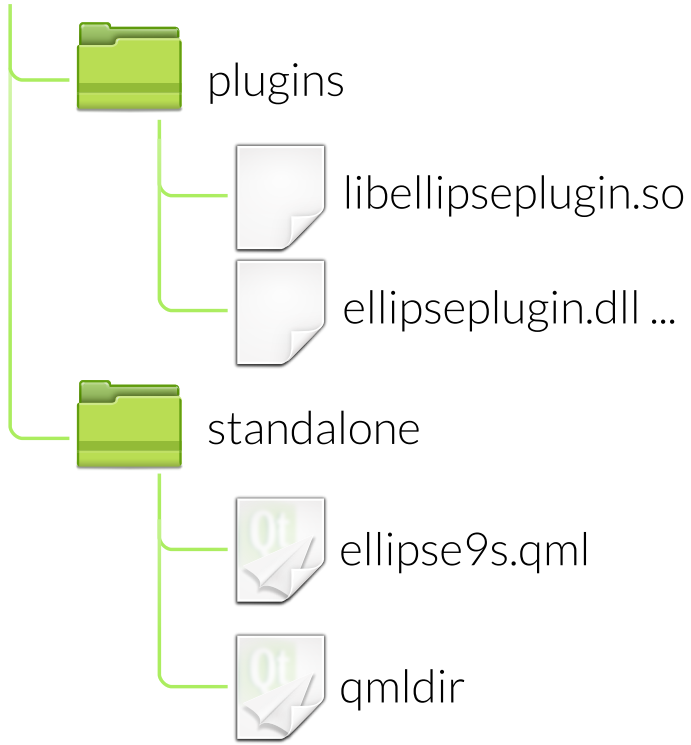
```
TEMPLATE  = lib
CONFIG   += qt plugin
QT       += quick

HEADERS  += ellipseitem.h \
            ellipseplugin.h

SOURCES  += ellipseitem.cpp \
            ellipseplugin.cpp

DESTDIR   = ../plugins
```

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

Using an Extension Plugin

To use the plugin with the `qmlscene` 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`**

In the `ellipse9s.qml` file:

```
import QtQuick 2.0

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

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 `QQmlExtensionPlugin`

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

- Register the extension with a URI

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

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

In the `ellipse9s.qml` file:

```
import QtQuick 2.0
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");
```

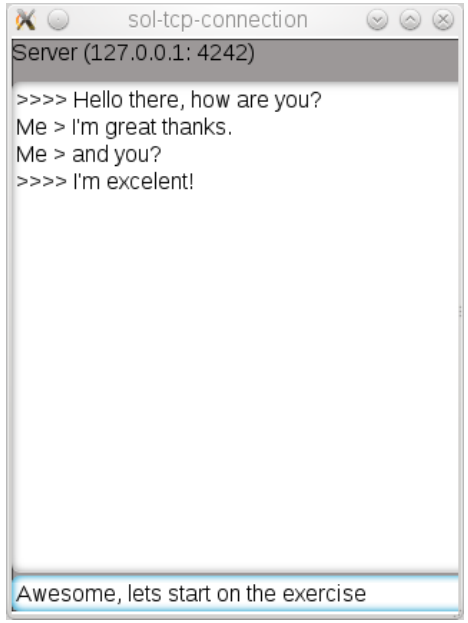
- corresponds to

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

Summary of Extension Plugins

- Extensions can be compiled as plugins
  - define and implement a `QQmlExtensionPlugin` subclass
  - define the version of the plugin in the extension
  - build a Qt plugin project with the `quick` option enabled
- Plugins can be loaded by the `qmlscene` 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

LAB: Chat Program



- The handout contains a partial solution for a small chat program.
- One side of the chat will be a server (using QTcpServer) and the other end connect to it.
- The TCP communication is already implemented in C++
- The GUI is implemented in QML
- Missing: Is the glue which makes the two parts work together.
- **STEPS** are available in the file readme.txt.

Lab qml-cpp-integration/lab-tcp-connection

© Digia Plc.

Digia, Qt and the Digia and Qt logos are the registered trademarks of Digia Plc. in Finland and other countries worldwide.