Qt Quick for Qt Developers

Integrating QML with C++

The Qt Company

Based on Qt 5.4 (QtQuick 2.4)

Contents

- 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

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

Declarative Environment

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 QtQuick module

Setting up a QtQuick Application

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/animation.qml")));
    return app.exec();
}
```

Demo: qml-cpp-integration/ex-simpleviewer

Setting up QtQuick

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

```
import QtQuick 2.0
import QtQuick.Window 2.2

Window {
    visible: true
    width: 400; height: 300
```

Demo: qml-cpp-integration/ex-simpleviewer

Exporting C++ Objects to QML

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

Exporting C++ Objects to QML

- Class `QQmlContext` exports the instance to QML.

```
int main(int argc, char ** argv) {
    QGuiApplication app(argc, argv);

    AnimalModel model; model.addAnimal(Animal("Wolf", "Medium"));
    model.addAnimal(Animal("Polar bear", "Large"));
    model.addAnimal(Animal("Quoll", "Small"));

    QQmlApplicationEngine engine;
    QQmlContext *ctxt = engine.rootContext();
    ctxt->setContextProperty(”animalModel", &model);

    engine.load(QUrl(QStringLiteral("qrc:/view.qml")));

    return app.exec();
}
```

Using the Object in QML

- Use the instances like any other QML object

```
Window {
    visible: true
    width: 200; height: 250

    ListView {
        width: 200; height: 250
        model: animalModel

        delegate: Text { text: "Animal: " + type + ", " + size }
    }
}
```

What Is Exported?

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

```
class IntervalSettings : public QObject
{
    Q_OBJECT
    Q_PROPERTY(int duration READ duration WRITE setDuration
                            NOTIFY durationChanged)
    Q_ENUMS(Unit)
    Q_PROPERTY(Unit unit READ unit WRITE setUnit NOTIFY unitChanged)
public:
    enum Unit { Minutes, Seconds, MilliSeconds };
```

Exporting Classes to QML

Overview

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:
    explicit Timer( QObject* parent = 0 );

private:
    QTimer* m_timer;
}
```

Implementing the Class

- Element `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 "timer.h"
#include <QGuiApplication>
#include <qqml.h> // for qmlRegisterType
#include <QQmlApplicationEngine>

int main(int argc, char **argv) {
    QGuiApplication app( argc, argv );
    // Expose the Timer class
    qmlRegisterType<Timer>( "CustomComponents", 1, 0, "Timer" );

    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
    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
    - version1.0 (first number is major; second Is minor)
- Available as the `Timer` element
    - The `Timer` element is an non-visual item
    - A subclass of `QObject`

Step 3+4 Importing and Using the Class

- In the *main.qml* file:

```
import CustomComponents 1.0

Window {
    visible: true; width: 500; height: 360
    Rectangle { anchors.fill: parent
        Timer { id: timer }
    }
    …
}
```

Demo: qml-cpp-integration/ex_simple_timer

Adding Properties

The Qt Company

- In the *`main.qml`* file:

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

- A new `interval` property

Demo: qml-cpp-integration/ex_timer_properties

Declaring a Property

- In the *timer.h* file:

```
class Timer : public QObject
{
    Q_OBJECT
    Q_PROPERTY(int interval READ interval WRITE setInterval
                            NOTIFY intervalChanged) // Or use MEMBER
    ….
```

- 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

Implementing Getter and Setter

- 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();
    Q_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

Adding Signals

- In the *main.qml* file:

```
Rectangle {
     …
     Timer {
          id: 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

Declaring a Signal

- In the *timer.h* file:

```
Q_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

Emitting the Signal

- In the `timer.cpp` file:

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

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

Handling the Signal

- In the `main.qml` file:

```
Timer {
    id: 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:

- Create C++ slots
  - Automatically exposed to QML
  - Useful for methods that do not return values

- Mark regular C++ functions as invokable
  - Allows values to be returned

Adding Slots

- In the `main.qml` file:

```
Timer {
    id: timer
    interval: 1000
    onTimeout: {
        console.log( "Timer fired!" );
    }
}
MouseArea {
    anchors.fill: parent
    onClicked: {
        if (timer.active == false) {
            timer.start();
        } else {
            timer.stop();
        }
    }
}
```

Adding Slots

- Element 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

Declaring Slots

- In the *`timer.h`* file:

```
public Q_SLOTS:
    void start();
    void stop();
```

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

Implementing Slots

- In the *timer.cpp* file:

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

- Remember to emit notifier signal for any changing properties

Adding Methods

- In the *`main.qml`* file:

```
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

Declaring a Method

- 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

Implementing a Method

- 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

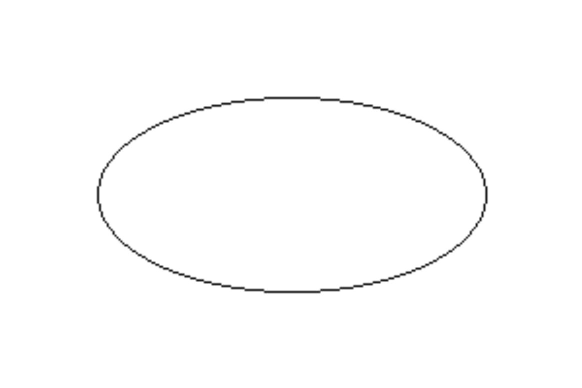
- Define QML-callable methods
  - Reuse slots as QML-callable methods
  - Methods that return values are marked using `Q_INVOKABLE`

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);
}
```

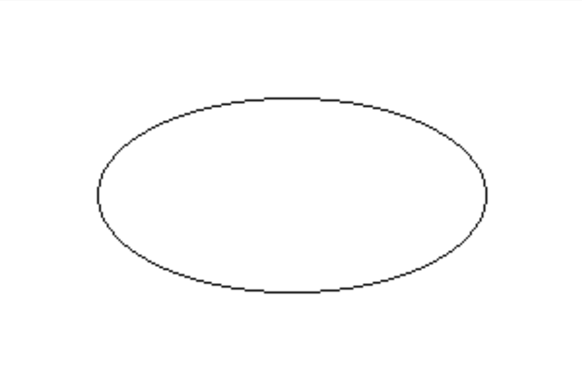
Exporting a QPainter based GUI Class cont'd.

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

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

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

    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/ellipse1.qml")));
    return app.exec();
}
```

Exporting a QPainter based GUI Class cont'd.

- In the *`ellipse1.qml`* file:

```
import Shapes 1.0

Window {
    visible: true
    width: 300; height: 200
    Item {
        anchors.fill: parent
        Ellipse {
            x: 50; y: 50
            width: 200; height: 100
        }
    }
}
```

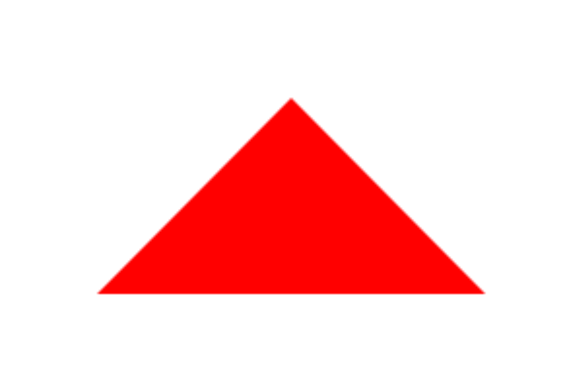


Demo: qml-cpp-integration/ex-simple-item

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 non GUI 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 "triangleitem.h"
#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

Using Custom Types

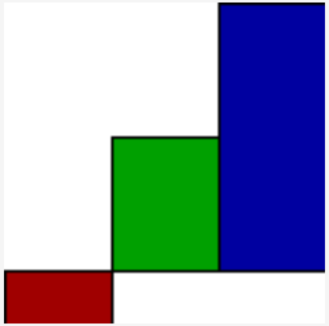
Defining Custom Property Types

- Enums
- Custom types as property values

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

- Collection of custom types

```
Chart {
    anchors.fill: parent
    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_PROPERTY( int duration READ duration WRITE setDuration
                             NOTIFY durationChanged )
    Q_ENUMS( Unit )
    Q_PROPERTY( Unit unit READ unit WRITE setUnit NOTIFY unitChanged )
public:
    enum Unit { Minutes, Seconds, MilliSeconds };
```

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

Custom Classes as Property Types

- Use the subtype 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:
    QTimer *m_timer;
    IntervalSettings *m_settings;
}
```

Custom Classes as Property Types cont'd.

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

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

Custom Classes as Property Types cont'd.

- Instantiating allow you this syntax:

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

- Alternatively you would need this syntax:

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

Custom Classes as Property Types cont'd.

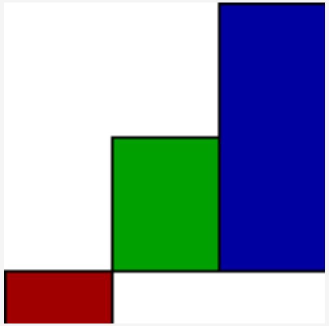
- Both classes must be exported to QML

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

Demo: qml-cpp-integration/ex_timer_custom_types

Collections of Custom Types

```
Chart {
    anchors.fill: parent
    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

Declaring the List Property

- In the *`chartitem.h`* file:

```
class ChartItem : public QQuickPaintedItem
{
    Q_OBJECT
    Q_PROPERTY(QQmlListProperty<BarItem> bars READ bars NOTIFY barsChanged)

public:
    ChartItem(QQuickItem *parent = 0);
    void paint(QPainter *painter);
    QQmlListProperty<BarItem> bars();
    …
}
```

- 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:

```
    QQmlListProperty<BarItem> bars();
    …
Q_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,
                                           0, 0, 0);
}
```

- Defines and returns a list of `BarItem` objects
  - With an append function
- Possible to define count, at and clear functions as well

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 }
}
```

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 re-implement

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

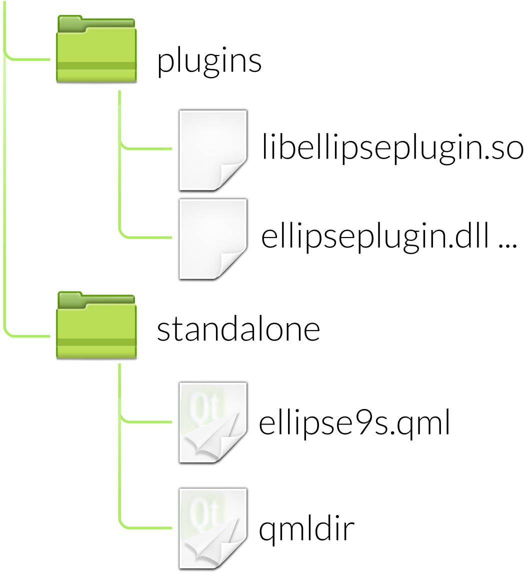
```
TEMPLATE = lib
CONFIG += qt plugin
QT += quick

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 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
  - File `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:

```
Item {
    anchors.fill: parent
    Ellipse {
        x: 50; y: 50
        width: 200;
        height: 100
    }
}
```

- Use the custom item directly
- No need to import any custom modules
  - Files `qmldir` and `ellipse9s.qml` are in the same project directory
  - Element `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 `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

Using an Extension Plugin

- In the *`ellipse9s.qml`* file:

```
import Shapes 9.0

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

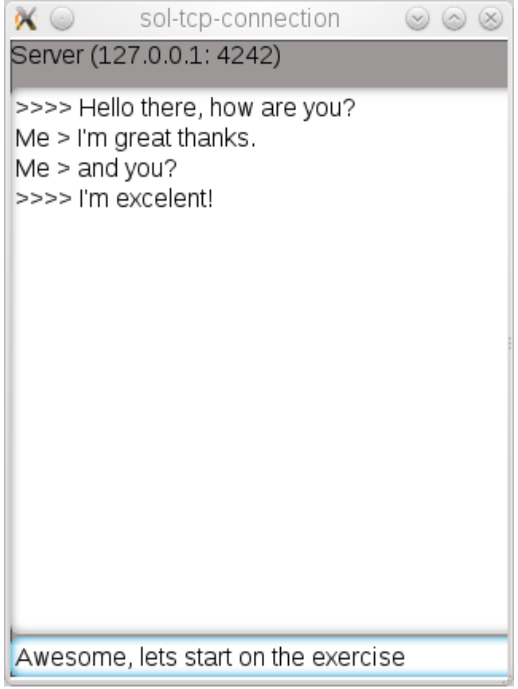
- The `Ellipse` item is part of the `Shapes` module
- A different URI makes a different import necessary; e.g.,
  - `plugin->registerTypes(“com.theqtcompany.examples.Shapes”);`
  - corresponds to `import com.theqtcompany.examples.Shapes 9.0`

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 within 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++ extensions
  - 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 connection is already implemented in C++
- The GUI is implemented in QML
- Missing: The glue which makes the two parts work together
- STEPS aree available in the file `readme.txt`



```
Server (127.0.0.1: 4242)
>>>> Hello there, how are you?
Me > I'm great thanks.
Me > and you?
>>>> I'm excelent!
```

Awesome, lets start on the exercise

Lab: qml-cpp-integration/lab-tcp-conection