ICS

QML Programming Fundamentals and Beyond

QML C++ Integration

Material based on Qt 5.12

Course Outline

Session 1: April 28, Introduction to QML
- About QML
- Properties
- Basic Types

Session 2: May 1, QML Item Placement
- How to correctly size and place items
- When to use Anchors, Layouts and Positioners

Session 3: May 5, Touch Interaction
- QML Signals
- Touch Events
- Single and Multi-Touch
- Swipe and Pinch Gestures

Session 4: May 8, States & Transitions
- Creating and defining states
- Using Transitions

Session 5: May 15, Custom Items & Components
- Creating your own Components
- Creating a Module

Session 6: May 19, Model / View
- Model / View
- QML Models
- QML Views

Session 7: May 22, C++ Integration
- Why expose C++ to QML
- Exposing C++ Objects
- Exposing C++ Classes

About ICS

ICS Designs User Experiences and Develops Software for Connected Devices

- Largest source of independent Qt expertise in North America since 2002
- Headquartered in Waltham, MA with offices in California, Canada, Europe
- Includes Boston UX, ICS' UX design division

- Embedded, touchscreen, mobile and desktop applications
- Exclusive Open Enrollment Training Partner in North America

Qt Service Partner

BOSTON UX

UX/UI Design and Development for Connected Devices Across Many Industries

Agenda

- Benefits of C++ Integration
- Exposing C++ Objects to QML
- Exporting C++ Classes to QML

Benefits of QML and C++ Integration

- QML engine has tight coupling with Qt's meta object system (MOC)
  - Allows access to C++ functionality directly in the QML code
- Separation between UI code and application logic
  - User Interface in QML and Javascript (use sparingly!)
  - Application logic in C++
- Custom QML object types can be defined in C++
  - Any `QObject`-derived C++ class can be registered as a QML object type
- Access the functionality of Qt Quick C++ API
  - E.g. to dynamically generate images using `QQuickImageProvider`

The Integration Method

C++ classes can be exposed to QML in a variety of ways:

- Register it as an instantiable QML type
  - It can be instantiated and used like any ordinary QML object type
- Register it as a Singleton Type
  - A single instance of the class can be imported from QML code
- An instance can be embedded into QML code as a context property
  - Allows the instance's properties, methods, and signals to be accessed from QML

Exposing C++ Objects: What Is Exposed?

- **Properties**
- **Signals**
- **Slots**
- Enums declared with the `Q_ENUM` macro
- Methods declared with the `Q_INVOKABLE` macro

```
class IntervalSettings : public QObject {
    Q_OBJECT
    Q_PROPERTY(int duration READ duration WRITE setDuration NOTIFY durationChanged)
    Q_PROPERTY(Unit timeUnit READ timeUnit WRITE setTimeUnit NOTIFY timeUnitChanged)

public:
    enum Unit { Minutes, Seconds, MilliSeconds };
     Q_ENUM(Unit)
     Q_INVOKABLE void start();
     ...
};
```

Exposing C++ Objects: setContextProperty

- Class **QQmlContext** exports the instance to QML
- setContextProperty will expose our object using the provided name.

```
#include <QQmlContext>

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

    MyObject backendData;

    QQmlApplicationEngine engine;
    QQmlContext *ctxt = engine.rootContext();
    ctxt->setContextProperty("appData", &backendData);

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

Using the Exported Object in QML

Use the instances like any other QML object

```
Window {
    visible: true
    width: appData.appWidth
    height: appData.appHeight
    ...
}
```

Exporting C++ Classes: Overview

To define a new type in QML, follow the four steps:

1. C++: Subclass either `QObject` or `QQuickItem`
2. C++: Register the type with the QML engine
3. QML: Import the module containing the new item
4. QML: Use the item like any other standard item

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

Step 1: Defining the Class

```
#include <QObject>

class QTimer;

class MyTimer : public QObject {
    Q_OBJECT

public:
    explicit MyTimer(QObject *parent = nullptr);

private:
    QTimer *m_timer;
}
```

Step 1: Implementing the Class

- QML type **Timer** is a **QObject** subclass
- As with all **QObject**s, each item can have a parent
- Non-GUI custom items do not need to paint (more later)

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

MyTimer::MyTimer(QObject* parent)
    : QObject(parent), m_timer(new QTimer(this)) {

    m_timer->setInterval(1000);
    m_timer->start();
}
```

Step 2: Registering the Class

- **`MyTimer`** is registered as a QML type in our new module *CustomComponents*
- Automatically available to the main.qml file

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

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

    qmlRegisterType<MyTimer>("CustomComponents", 1, 0, "MyTimer");

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

Step 2: Reviewing the Registration

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

- Registers **`MyTimer`** C++ class with the **`QQMLEngine`**
- Available from the *CustomComponents* QML module
- Version 1.0
  - The first number is major, and the second is minor
- Available as the **`MyTimer`** QML type
  - The **`MyTimer`** type is an non-visual item
  - It also subclasses QObject

Step 3 & 4: Importing and Using the Class

In the *main.qml* file

```
import CustomComponents 1.0 // Newly defined module

Window {
    visible: true;
    width: 500; height: 360

    Rectangle {
        anchors.fill: parent
        MyTimer { // our custom QML type
            id: timer
        }
    }
}
```

Declaring a Property in C++

In the my*timer.h* file:

```
class MyTimer : 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 update property bindings

Declaring a Getter, Setter, and Signal

In the my*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 a Getter and Setter

In the *mytimer.cpp* file:

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

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


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

Declaring the Signal in C++

In the my*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

Emitting the Signal

In the my*timer.cpp* file:

```
MyTimer::MyTimer(QObject *parent)
    : QObject(parent), m_timer(new QTimer(this)) {

    connect(m_timer, &QTimer::timeout, this, &MyTimer::timeout);
}
```


- Update the constructor to add the connection
- Connect **`QTimer::timeout()`** signal to our **`MyTimer::timeout()`** signal

Handling the Signal

In the *main.qml* file:

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


In C++:

- The **`QTimer::timeout()`** signal is emitted
- Because we connected the signals in C++, **`MyTimer::timeout()`** is emitted

In QML:

- The **`MyTimer`** item's **`onTimeout`** handler is called
- Outputs message to *stderr*

Adding Methods to QML Items

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

1. Create Functions as C++ slots
    a. Automatically exposed to QML
    b. Useful for methods that do not return values
2. Mark regular C++ functions with the `Q_INVOKABLE` macro
    a. Allows values to be returned

Declaring a Method in C++

In the my*timer.h* file:

```
public:
    explicit MyTimer(QObject* parent = nullptr);

    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 the Method in C++

In the my*timer.cpp* file:

```
int MyTimer::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
- Returns an int representing the number of milliseconds
- Do not use the **`Q_INVOKABLE`** macro in the source file

Adding Methods in QML

In the *main.qml* file:

```
MyTimer {
    id: timer
    interval: timer.randomInterval(500, 1500)
    onTimeout: {
        console.log("Timer fired!");
    }
}
```


- `MyTimer` now has a **`randomInterval()`** method
- The method obtains a random interval value
- Accepts arguments for min and max intervals
- The interval is set using the interval property


TIME FOR A LIVE DEMO
38
Copyright (c) ICS, Inc.
All Rights Reserved.
WHAT COULD GO WRONG?
memegenerator.net

Q&A Session

If you have additional questions or feedback,
please contact us at QtTraining@ics.com

COMING SOON!

Hands-on Virtual Training:
Building an Embedded Device Application with Qt

Course begins July 14

More details and registration available early June