FREE eBook

LEARNING

qml

Free unaffiliated eBook created from
Stack Overflow contributors.

#qml

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: qml

It is an unofficial and free qml ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official qml.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with qml

Remarks

QML is an acronym that stands for **Q**t **M**eta-object **L**anguage. It is a declarative programming language that is part of the Qt framework. QML's main purpose is fast and easy creation of user interfaces for desktop, mobile and embedded systems. QML allows seamless integration of JavaScript, either directly in the QML code or by including JavaScript files.

Versions

Qt Version	QtQuick Version	Release Date
4.7	1.0	2010-09-21
4.8	1.1	2011-12-15
5.0	2.0	2012-12-19
5.1	2.1	2013-06-03
5.2	2.2	2013-12-12
5.3	2.3	2014-05-20
5.4	2.4	2014-12-10
5.5	2.5	2015-07-01
5.6	2.6	2016-03-15
5.7	2.7	2016-06-16
5.8	2.7	2017-01-23

Examples

Installation

QML comes with newer Version of the cross-platform application framework Qt. You can find the newest Version of Qt in the Downloads section.

To create a new QML Project in the Qt Creator IDE, select "File -> New ..." and under "Applications" select "Qt Quick-Application". After clicking "select" you can now name and set the path for this project. After hitting "next" you can select which components you want to use, if

unsure just leave the default and click on "next". The two next steps will allow you to setup up a Kit and Source Control if you want to, otherwise keep the default settings.

You now have created a simple and ready to use QML application.

Hello World

A simple application showing the text "Hello World" in the center of the window.

```
import QtQuick 2.3
import QtQuick.Window 2.0

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World") //The method qsTr() is used for translations from one language
to other.

    Text {
        text: qsTr("Hello World")
        anchors.centerIn: parent
    }
}
```

Creating a simple button

You can easily transform every component in a clickable button using the MouseArea component. The code below displays a 360x360 window with a button and a text in the center; pressing the button will change the text:

```
import QtQuick 2.0

Rectangle {
    width: 360
    height: 360

    Rectangle {
        id: button

        width: 100
        height: 30
        color: "red"
        radius: 5     // Let's round the rectangle's corner a bit, so it resembles more a
button
        anchors.centerIn: parent

        Text {
            id: buttonText
            text: qsTr("Button")
            color: "white"
            anchors.centerIn: parent
        }

        MouseArea {
            // We make the MouseArea as big as its parent, i.e. the rectangle. So pressing
```

```
anywhere on the button will trigger the event
            anchors.fill: parent

            // Exploit the built-in "clicked" signal of the MouseArea component to do
something when the MouseArea is clicked.
            // Note that the code associated to the signal is plain JavaScript. We can
reference any QML objects by using their IDs
            onClicked: {
                buttonText.text = qsTr("Clicked");
                buttonText.color = "black";
            }
        }
    }
}
```

Display an image

This example shows the simplest usage of the Image component to display an image.

The Image `source` property is a url type that can be either a file with an absolute or relative path, an internet URL (`http://`) or a Qt resource (`qrc:/`)

```
import QtQuick 2.3

Rectangle {
    width: 640
    height: 480

    Image {
         source: "image.png"
    }
}
```

Mouse Event

This example shows how mouse event is used in QML.

```
import QtQuick 2.7
import QtQuick.Window 2.2

Window {
    visible: true
    Rectangle {
        anchors.fill: parent
        width: 120; height: 240
        color: "#4B7A4A"

        MouseArea {
            anchors.fill: parent // set mouse area (i.e. covering the entire rectangle.)
            acceptedButtons:  Qt.AllButtons
            onClicked: {
                // print to console mouse location
                console.log("Mouse Clicked.")
                console.log("Mouse Location: <",mouseX,",",mouseY,">")

                //change Rectangle color
```

```
                if ( mouse.button === Qt.RightButton )
                    parent.color = 'blue'
                if ( mouse.button === Qt.LeftButton )
                    parent.color = 'red'
                if ( mouse.button === Qt.MiddleButton )
                    parent.color = 'yellow'
            }
            onReleased: {
                // print to console
                console.log("Mouse Released.")
            }
            onDoubleClicked: {
                // print to console
                console.log("Mouse Double Clicked.")
            }

        }
    }


}
```

Read Getting started with qml online: https://riptutorial.com/qml/topic/653/getting-started-with-qml

Chapter 2: Animation

Examples

Simple number animation

One of the very basic animations that you could come across is the `NumberAnimation`. This animation works by changing the numeric value of a property of an item from an initial state to a final state. Consider the following complete example:

```
import QtQuick 2.7
import QtQuick.Controls 2.0

ApplicationWindow {
    visible: true
    width: 400
    height: 640

    Rectangle{
        id: rect
        anchors.centerIn: parent
        height: 100
        width: 100
        color: "blue"
        MouseArea{
            anchors.fill: parent
            onClicked: na.running = true
        }

        NumberAnimation {
            id: na    //ID of the QML Animation type
            target: rect    //The target item on which the animation should run
            property: "height"    //The property of the target item which should be changed by
the animator to show effect
            duration: 200    //The duration for which the animation should run
            from: rect.height    //The initial numeric value of the property declared in
'property'
            to: 200    //The final numeric value of the property declared in 'property'
        }
    }
}
```

Behavior based animation

A behavior based animation allows you to specify that when a property changes the change should be animated over time.

```
ProgressBar {
    id: progressBar
    from: 0
    to: 100
    Behavior on value {
        NumberAnimation {
```

```
            duration: 250
        }
    }
}
```

In this example if anything changes the progress bar value the change will be animated over 250ms

Read Animation online: https://riptutorial.com/qml/topic/1281/animation

Chapter 3: Creating custom elements in C++

Examples

Creating custom elements in C++

QML came with rich set of visual elements. Using only QML we can build complex applications with these elements. Also it's very easy to build your own element based on set of standard items like Rectangle, Button, Image etc. Moreover, we can use items like Canvas to build element with custom painting. It would seem that we can build a variety of applications in QML only, without touching the capabilities of C++. And it's actually true but still sometimes we would like to make our application faster or we want to extend it with power of Qt or to add some opportunity which are not available in QML. And certainly there is such possibility in *QML*. Basically *QtQuick* uses *Scene Graph* to paint its content a high-performance rendering engine based on *OpenGL*. To implement our own visual element we can use 2 ways:

1. The traditional for Qt way using QPainter (QQuickPaintedItem).
2. The common QML way using QQuickItem and OpenGL functionality.

It is possible that the first method seems easier but it's worth considering that it is also slower than the first one since QtQuick paints the item's content on a surface and then insert it into scene graph so the rendering is a two-step operation. So using scene graph API directly is always significantly faster.

In order to explore both methods closer let's create our own element which definitely doesn't exist in QML, for example a triangle.

Class declaration

```
class QQuickCustomItem : public QQuickItem
{
    Q_OBJECT
    Q_PROPERTY(QColor color READ color WRITE setColor NOTIFY colorChanged)
public:
    QQuickCustomItem(QQuickItem *parent = Q_NULLPTR);

protected:
    QSGNode *updatePaintNode(QSGNode *oldNode, UpdatePaintNodeData *updatePaintNodeData);

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

private:
    QColor m_color;
    bool m_needUpdate;

signals:
    void colorChanged();
};
```

We add Q_OBJECT macro to work with signals. Also we add custom property to specify color of our Rectangle. To make it works all we need is reimplement virtual function QQuiclItem::updatePaintNode().

Class implementation.

Firstly we define a constructor.

```
QQuickCustomItem::QQuickCustomItem(QQuickItem *parent) :
    QQuickItem(parent),
    m_color(Qt::red),
    m_needUpdate(true)
{
    setFlag(QQuickItem::ItemHasContents);
}
```

Please note that the setFlag() function call is mandatory otherwise your object will not be added to the scene graph. Next, we define a function for the paining.

```
QSGNode *QQuickCustomItem::updatePaintNode(QSGNode *oldNode, QQuickItem::UpdatePaintNodeData
*updatePaintNodeData)
{
    Q_UNUSED(updatePaintNodeData)
    QSGGeometryNode *root = static_cast<QSGGeometryNode *>(oldNode);

    if(!root) {
        root = new QSGGeometryNode;
        QSGGeometry *geometry = new QSGGeometry(QSGGeometry::defaultAttributes_Point2D(), 3);
        geometry->setDrawingMode(GL_TRIANGLE_FAN);
        geometry->vertexDataAsPoint2D()[0].set(width() / 2, 0);
        geometry->vertexDataAsPoint2D()[1].set(width(), height());
        geometry->vertexDataAsPoint2D()[2].set(0, height());

        root->setGeometry(geometry);
        root->setFlag(QSGNode::OwnsGeometry);
        root->setFlag(QSGNode::OwnsMaterial);
    }

    if(m_needUpdate) {
        QSGFlatColorMaterial *material = new QSGFlatColorMaterial;
        material->setColor(m_color);
        root->setMaterial(material);
        m_needUpdate = false;
    }

    return root;
}
```

At the first call to the function our node isn't created yet so `oldNode` will be NULL. So we create the node and assign geometry and material to it. Here we use *GL_TRIANGLE_FAN* for our geometry to paint solid rectangle. This point is the same as in OpenGL. For example to draw triangle frame we can change the code to:

```
geometry->setDrawingMode(GL_LINE_LOOP);
geometry->setLineWidth(5);
```

You can refer to *OpenGL* manual to check for other shapes. So, all that remains is to define setter/getter for our property:

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

void QQuickCustomItem::setColor(const QColor &color)
{
    if(m_color != color) {
        m_color = color;
        m_needUpdate = true;
        update();
        colorChanged();
    }
}
```

Now there is only one small detail to make it works. We need to notify *QtQuick* of the new item. For example, you can add this code to your main.cpp:

```
qmlRegisterType<QQuickCustomItem>("stackoverflow.qml", 1, 0, "Triangle");
```

And here is our QML test file:

```
import QtQuick 2.7
import QtQuick.Window 2.0
import stackoverflow.qml 1.0

Window {
    width: 800
    height: 800
    visible: true

    Rectangle {
        width: 200
        height: 200
        anchors.centerIn: parent
        color: "lightgrey"

        Triangle {
            id: rect
            width: 200
            height: 200
            transformOrigin: Item.Top
            color: "green"
            onColorChanged: console.log("color was changed");
            PropertyAnimation on rotation {
                from: 0
                to: 360
                duration: 5000
                loops: Animation.Infinite
            }
        }
    }
    Timer {
        interval: 1000
```

```
        repeat: true
        running: true
        onTriggered: rect.color = Qt.rgba(Math.random(),Math.random(),Math.random(),1);
    }
}
```

As you see our item behaves like all other QML items. Now let's create the same item using QPainter:

All we need is to replace

```
QSGNode *updatePaintNode(QSGNode *oldNode, UpdatePaintNodeData *updatePaintNodeData);
```

with

```
void paint(QPainter *painter);
```

and, of cource inherit our class from `QQuickPaintedItem` instead of `QQuickItem`. Here is our painting function:

```
void QQuickCustomItem::paint(QPainter *painter)
{
    QPainterPath path;
    path.moveTo(width() / 2, 0);
    path.lineTo(width(), height());
    path.lineTo(0, height());
    path.lineTo(width() / 2, 0);
    painter->fillPath(path, m_color);
}
```

Everything else remains unchanged.

Read Creating custom elements in C++ online: https://riptutorial.com/qml/topic/6509/creating-custom-elements-in-cplusplus

Chapter 4: Integration with C++

Examples

Creating a QtQuick view from C++

It is possible to create a QtQuick view directly from C++ and to expose to QML C++ defined properties. In the code below the C++ program creates a QtQuick view and exposes to QML the height and width of the view as properties.

main.cpp

```
#include <QApplication>
#include <QQmlContext>
#include <QQuickView>

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

    // Creating the view and manually setting the QML file it should display
    QQuickView view;
    view.setSource(QStringLiteral("main.qml"));

    // Retrieving the QML context. This context allows us to expose data to the QML components
    QQmlContext* rootContext = view.rootContext();

    // Creating 2 new properties: the width and height of the view
    rootContext->setContextProperty("WINDOW_WIDTH", 640);
    rootContext->setContextProperty("WINDOW_HEIGHT", 360);

    // Let's display the view
    view.show();

    return app.exec();
}
```

main.qml

```
import QtQuick 2.0

Rectangle {
    // We can now access the properties we defined from C++ from the whole QML file
    width: WINDOW_WIDTH
    height: WINDOW_HEIGHT

    Text {
        text: qsTr("Hello World")
        anchors.centerIn: parent
    }
}
```

Creating a QtQuick Window from C++

As of Qt 5.1 and later you can use QQmlApplicationEngine instead of QQuickView to load and render a QML script.

With QQmlApplicationEngine you do need to use a QML Window type as your root element.

You can obtain the root context from the engine where you can then add global properties to the context which can be access by the engine when processing QML scripts.

main.cpp

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

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

    QQmlApplicationEngine engine;

    QQmlContext* rootContext = engine.rootContext();
    rootContext->setContextProperty("WINDOW_WIDTH", 640);
    rootContext->setContextProperty("WINDOW_HEIGHT", 360);

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

    return app.exec();
}
```

main.qml

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window { // Must be this type to be loaded by QQmlApplicationEngine.
    visible: true
    width: WINDOW_WIDTH   //Accessing global context declared in C++
    height: WINDOW_HEIGHT //Accessing global context declared in C++
    title: qsTr("Hello World")
    Component.onCompleted: {
        // We can access global context from within JavaScript too.
        console.debug( "Width: " + WINDOW_WIDTH )
        console.debug( "Height: " + WINDOW_HEIGHT )
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
            Qt.quit();
        }
    }

    Text {
        text: qsTr("Hello World")
        anchors.centerIn: parent
    }
}
```

Creating a simple model for TreeView

Since Qt 5.5 we have a new wonderful TreeView, a control we've all been waiting for. A TreeView implements a tree representation of items from a model. In general it looks like other QML views - ListView or TableView. But data structure of TreeView is more complex.

A data in ListView or TableView is represented by one-dimensional array of nodes. In TreeView each node can contain its own array of nodes. Therefore, unlike the others views in TreeView to get specified node we must know parent node, not only row or column of element.

Another major difference is that TreeView doesn't support ListModel. To provide a data we must subclass QAbstractItemModel. In Qt there are ready to use model classes like QFileSystemModel which provides access to local file system, or QSqlTableModel which provides access to a data base.

In following example we will create such model derived from QAbstractItemModel. But to make the example more realistic I suggest to make the model like ListModel but specified for trees so we can add nodes from QML. It's necessary to clarify that model itself doesn't contain any data but only provide access to it. So providing and organization of data is entirely our responsibility.

Since model data is organized in a tree the simplest node structure is seen as follows, in pseudo code:

```
Node {
    var data;
    Node parent;
    list<Node> children;
}
```

In C++ the node declaration should be as following:

```
class MyTreeNode : public QObject
{
    Q_OBJECT
public:
    Q_PROPERTY(QQmlListProperty<MyTreeNode> nodes READ nodes)
    Q_CLASSINFO("DefaultProperty", "nodes")
    MyTreeNode(QObject *parent = Q_NULLPTR);

    void setParentNode(MyTreeNode *parent);
    Q_INVOKABLE MyTreeNode *parentNode() const;
    bool insertNode(MyTreeNode *node, int pos = (-1));
    QQmlListProperty<MyTreeNode> nodes();

    MyTreeNode *childNode(int index) const;
    void clear();

    Q_INVOKABLE int pos() const;
    Q_INVOKABLE int count() const;

private:
    QList<MyTreeNode *> m_nodes;
    MyTreeNode *m_parentNode;
```

```
};
```

We derive our class from QObject to be able to create a node in `QML`. All the children nodes will be added to `nodes` property so next 2 part of code are the same:

```
TreeNode {
    nodes:[
        TreeNode {}
        TreeNode {}
    ]
}

TreeNode {
    TreeNode {}
    TreeNode {}
}
```

See this acticle to know more about default property.

Node class implementation:

```
MyTreeNode::MyTreeNode(QObject *parent) :
    QObject(parent),
    m_parentNode(nullptr) {}

void MyTreeNode::setParentNode(MyTreeNode *parent)
{
    m_parentNode = parent;
}

MyTreeNode *MyTreeNode::parentNode() const
{
    return m_parentNode;
}

QQmlListProperty<MyTreeNode> MyTreeNode::nodes()
{
    QQmlListProperty<MyTreeNode> list(this,
                                      0,
                                      &append_element,
                                      &count_element,
                                      &at_element,
                                      &clear_element);
    return list;
}

MyTreeNode *MyTreeNode::childNode(int index) const
{
    if(index < 0 || index >= m_nodes.length())
        return nullptr;
    return m_nodes.at(index);
}

void MyTreeNode::clear()
{
    qDeleteAll(m_nodes);
    m_nodes.clear();
}
```

```
bool MyTreeNode::insertNode(MyTreeNode *node, int pos)
{
    if(pos > m_nodes.count())
        return false;
    if(pos < 0)
        pos = m_nodes.count();
    m_nodes.insert(pos, node);
    return true;
}

int MyTreeNode::pos() const
{
    MyTreeNode *parent = parentNode();
    if(parent)
        return parent->m_nodes.indexOf(const_cast<MyTreeNode *>(this));
    return 0;
}

int MyTreeNode::count() const
{
    return m_nodes.size();
}

MyTreeNode *MyTreeModel::getNodeByIndex(const QModelIndex &index)
{
    if(!index.isValid())
        return nullptr;
    return static_cast<MyTreeNode *>(index.internalPointer());
}

QModelIndex MyTreeModel::getIndexByNode(MyTreeNode *node)
{
    QVector<int> positions;
    QModelIndex result;
    if(node) {
        do
        {
            int pos = node->pos();
            positions.append(pos);
            node = node->parentNode();
        } while(node != nullptr);


        for (int i = positions.size() - 2; i >= 0 ; i--)
        {
            result = index(positions[i], 0, result);
        }
    }
    return result;
}

bool MyTreeModel::insertNode(MyTreeNode *childNode, const QModelIndex &parent, int pos)
{
    MyTreeNode *parentElement = getNode(parent);
    if(pos >= parentElement->count())
        return false;
    if(pos < 0)
        pos = parentElement->count();

    childNode->setParentNode(parentElement);
```

```
    beginInsertRows(parent, pos, pos);
    bool retValue = parentElement->insertNode(childNode, pos);
    endInsertRows();
    return retValue;
}

MyTreeNode *MyTreeModel::getNode(const QModelIndex &index) const
{
    if(index.isValid())
        return static_cast<MyTreeNode *>(index.internalPointer());
    return m_rootNode;
}
```

To expose list-like property to QML through QQmlListProperty we need next 4 function:

```
void append_element(QQmlListProperty<MyTreeNode> *property, MyTreeNode *value)
{
    MyTreeNode *parent = (qobject_cast<MyTreeNode *>(property->object));
    value->setParentNode(parent);
    parent->insertNode(value, -1);
}

int count_element(QQmlListProperty<MyTreeNode> *property)
{
    MyTreeNode *parent = (qobject_cast<MyTreeNode *>(property->object));
    return parent->count();
}

MyTreeNode *at_element(QQmlListProperty<MyTreeNode> *property, int index)
{
    MyTreeNode *parent = (qobject_cast<MyTreeNode *>(property->object));
    if(index < 0 || index >= parent->count())
        return nullptr;
    return parent->childNode(index);
}

void clear_element(QQmlListProperty<MyTreeNode> *property)
{
    MyTreeNode *parent = (qobject_cast<MyTreeNode *>(property->object));
    parent->clear();
}
```

Now let's declare the model class:

```
class MyTreeModel : public QAbstractItemModel
{
    Q_OBJECT
public:
    Q_PROPERTY(QQmlListProperty<MyTreeNode> nodes READ nodes)
    Q_PROPERTY(QVariantList roles READ roles WRITE setRoles NOTIFY rolesChanged)
    Q_CLASSINFO("DefaultProperty", "nodes")

    MyTreeModel(QObject *parent = Q_NULLPTR);
    ~MyTreeModel();

    QHash<int, QByteArray> roleNames() const Q_DECL_OVERRIDE;
    QVariant data(const QModelIndex &index, int role) const Q_DECL_OVERRIDE;
    Qt::ItemFlags flags(const QModelIndex &index) const Q_DECL_OVERRIDE;
    QModelIndex index(int row, int column, const QModelIndex &parent = QModelIndex()) const
```

```
Q_DECL_OVERRIDE;
    QModelIndex parent(const QModelIndex &index) const Q_DECL_OVERRIDE;
    int rowCount(const QModelIndex &parent = QModelIndex()) const Q_DECL_OVERRIDE;
    int columnCount(const QModelIndex &parent = QModelIndex()) const Q_DECL_OVERRIDE;
    QQmlListProperty<MyTreeNode> nodes();

    QVariantList roles() const;
    void setRoles(const QVariantList &roles);

    Q_INVOKABLE MyTreeNode * getNodeByIndex(const QModelIndex &index);
    Q_INVOKABLE QModelIndex getIndexByNode(MyTreeNode *node);
    Q_INVOKABLE bool insertNode(MyTreeNode *childNode, const QModelIndex &parent =
QModelIndex(), int pos = (-1));

protected:
    MyTreeNode *getNode(const QModelIndex &index) const;

private:
    MyTreeNode *m_rootNode;
    QHash<int, QByteArray> m_roles;

signals:
    void rolesChanged();
};
```

Since we derived out model class from abstract QAbstractItemModel we must redefine next function: data(), flags(), index(), parent(), columnCount() and rowCount(). In order our model could work with `QML` we define roleNames(). Also, as well as in node class we define default property to be able to add nodes to the model in `QML`. `roles` property will hold a list of role names.

The implementation:

```
MyTreeModel::MyTreeModel(QObject *parent) :
    QAbstractItemModel(parent)
{
    m_rootNode = new MyTreeNode(nullptr);
}
MyTreeModel::~MyTreeModel()
{
    delete m_rootNode;
}

QHash<int, QByteArray> MyTreeModel::roleNames() const
{
    return m_roles;
}

QVariant MyTreeModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid())
        return QVariant();

    MyTreeNode *item = static_cast<MyTreeNode*>(index.internalPointer());
    QByteArray roleName = m_roles[role];
    QVariant name = item->property(roleName.data());
    return name;
}

Qt::ItemFlags MyTreeModel::flags(const QModelIndex &index) const
```

```
{
    if (!index.isValid())
        return 0;

    return QAbstractItemModel::flags(index);
}

QModelIndex MyTreeModel::index(int row, int column, const QModelIndex &parent) const
{
    if (!hasIndex(row, column, parent))
        return QModelIndex();

    MyTreeNode *parentItem = getNode(parent);
    MyTreeNode *childItem = parentItem->childNode(row);
    if (childItem)
        return createIndex(row, column, childItem);
    else
        return QModelIndex();
}

QModelIndex MyTreeModel::parent(const QModelIndex &index) const
{
    if (!index.isValid())
        return QModelIndex();

    MyTreeNode *childItem = static_cast<MyTreeNode*>(index.internalPointer());
    MyTreeNode *parentItem = static_cast<MyTreeNode *>(childItem->parentNode());

    if (parentItem == m_rootNode)
        return QModelIndex();

    return createIndex(parentItem->pos(), 0, parentItem);
}

int MyTreeModel::rowCount(const QModelIndex &parent) const
{
    if (parent.column() > 0)
        return 0;
    MyTreeNode *parentItem = getNode(parent);
    return parentItem->count();
}

int MyTreeModel::columnCount(const QModelIndex &parent) const
{
    Q_UNUSED(parent);
    return 1;
}

QQmlListProperty<MyTreeNode> MyTreeModel::nodes()
{
    return m_rootNode->nodes();
}

QVariantList MyTreeModel::roles() const
{
    QVariantList list;
    QHashIterator<int, QByteArray> i(m_roles);
    while (i.hasNext()) {
        i.next();
        list.append(i.value());
    }
```

```
    return list;
}

void MyTreeModel::setRoles(const QVariantList &roles)
{
    static int nextRole = Qt::UserRole + 1;
    foreach(auto role, roles) {
        m_roles.insert(nextRole, role.toByteArray());
        nextRole ++;
    }
}

MyTreeNode *MyTreeModel::getNodeByIndex(const QModelIndex &index)
{
    if(!index.isValid())
        return nullptr;
    return static_cast<MyTreeNode *>(index.internalPointer());
}

QModelIndex MyTreeModel::getIndexByNode(MyTreeNode *node)
{
    QVector<int> positions;
    QModelIndex result;
    if(node) {
        do
        {
            int pos = node->pos();
            positions.append(pos);
            node = node->parentNode();
        } while(node != nullptr);


        for (int i = positions.size() - 2; i >= 0 ; i--)
        {
            result = index(positions[i], 0, result);
        }
    }
    return result;
}


bool MyTreeModel::insertNode(MyTreeNode *childNode, const QModelIndex &parent, int pos)
{
    MyTreeNode *parentElement = getNode(parent);
    if(pos >= parentElement->count())
        return false;
    if(pos < 0)
        pos = parentElement->count();

    childNode->setParentNode(parentElement);
    beginInsertRows(parent, pos, pos);
    bool retValue = parentElement->insertNode(childNode, pos);
    endInsertRows();
    return retValue;
}

MyTreeNode *MyTreeModel::getNode(const QModelIndex &index) const
{
    if(index.isValid())
        return static_cast<MyTreeNode *>(index.internalPointer());
```

```
    return m_rootNode;
}
```

In general, this code it's not much different from the standard implementation, for example Simple tree example

Instead of defining roles in C++ we provide a way to do that from QML. TreeView events and methods basically work with QModelIndex. I personally don't see much sense to pass that to qml as the only thing you can do with it is to pass it back to the model.

Anyway, our class provides a way to convert index to node and vice versa. To be able to use our classes in QML we need to register it:

```
qmlRegisterType<MyTreeModel>("qt.test", 1, 0, "TreeModel");
qmlRegisterType<MyTreeNode>("qt.test", 1, 0, "TreeElement");
```

And finelly, en example of how we can use our model with TreeView in QML:

```
import QtQuick 2.7
import QtQuick.Window 2.2
import QtQuick.Dialogs 1.2
import qt.test 1.0


Window {
    visible: true
    width: 800
    height: 800
    title: qsTr("Tree example")

    Component {
        id: fakePlace
        TreeElement {
            property string name: getFakePlaceName()
            property string population: getFakePopulation()
            property string type: "Fake place"
            function getFakePlaceName() {
                var rez = "";
                for(var i = 0;i < Math.round(3 + Math.random() * 7);i ++) {
                    rez += String.fromCharCode(97 + Math.round(Math.random() * 25));
                }
                return rez.charAt(0).toUpperCase() + rez.slice(1);
            }
            function getFakePopulation() {
                var num = Math.round(Math.random() * 100000000);
                num = num.toString().split("").reverse().join("");
                num = num.replace(/(\d{3})/g, '$1,');
                num = num.split("").reverse().join("");
                return num[0] === ',' ? num.slice(1) : num;
            }
        }
    }

    TreeModel {
        id: treemodel
        roles: ["name","population"]
```

```
        TreeElement {
            property string name: "Asia"
            property string population: "4,164,252,000"
            property string type: "Continent"
            TreeElement {
                property string name: "China";
                property string population: "1,343,239,923"
                property string type: "Country"
                TreeElement { property string name: "Shanghai"; property string population:
"20,217,700"; property string type: "City" }
                TreeElement { property string name: "Beijing"; property string population:
"16,446,900"; property string type: "City" }
                TreeElement { property string name: "Chongqing"; property string population:
"11,871,200"; property string type: "City" }
            }
            TreeElement {
                property string name: "India";
                property string population: "1,210,193,422"
                property string type: "Country"
                TreeElement { property string name: "Mumbai"; property string population:
"12,478,447"; property string type: "City" }
                TreeElement { property string name: "Delhi"; property string population:
"11,007,835"; property string type: "City" }
                TreeElement { property string name: "Bengaluru"; property string population:
"8,425,970"; property string type: "City" }
            }
            TreeElement {
                property string name: "Indonesia";
                property string population: "248,645,008"
                property string type: "Country"
                TreeElement {property string name: "Jakarta"; property string population:
"9,588,198"; property string type: "City" }
                TreeElement {property string name: "Surabaya"; property string population:
"2,765,487"; property string type: "City" }
                TreeElement {property string name: "Bandung"; property string population:
"2,394,873"; property string type: "City" }
            }
        }
        TreeElement { property string name: "Africa"; property string population:
"1,022,234,000"; property string type: "Continent" }
        TreeElement { property string name: "North America"; property string population:
"542,056,000"; property string type: "Continent" }
        TreeElement { property string name: "South America"; property string population:
"392,555,000"; property string type: "Continent" }
        TreeElement { property string name: "Antarctica"; property string population: "4,490";
property string type: "Continent" }
        TreeElement { property string name: "Europe"; property string population:
"738,199,000"; property string type: "Continent" }
        TreeElement { property string name: "Australia"; property string population:
"29,127,000"; property string type: "Continent" }
    }

    TreeView {
        anchors.fill: parent
        model: treemodel
        TableViewColumn {
            title: "Name"
            role: "name"
            width: 200
        }
        TableViewColumn {
```

```
            title: "Population"
            role: "population"
            width: 200
        }

        onDoubleClicked: {
            var element = fakePlace.createObject(treemodel);
            treemodel.insertNode(element, index, -1);
        }
        onPressAndHold: {
            var element = treemodel.getNodeByIndex(index);
            messageDialog.text = element.type + ": " + element.name + "\nPopulation: " +
element.population;
            messageDialog.open();
        }
    }
    MessageDialog {
          id: messageDialog
          title: "Info"
      }
}
```

Double click for adding a node, press and hold for node info.

Read Integration with C++ online: https://riptutorial.com/qml/topic/2254/integration-with-cplusplus

Chapter 5: Property binding

Remarks

An object's property can be assigned a static value which stays constant until it is explicitly assigned a new value. However, to make the fullest use of QML and its built-in support for dynamic object behaviors, most QML objects use property bindings.

Property bindings are a core feature of QML that lets developers specify relationships between different object properties. When a property's dependencies change in value, the property is automatically updated according to the specified relationship.

Examples

Basics about property bindings

Consider this simple example:

```
import QtQuick 2.7
import QtQuick.Controls 2.0

ApplicationWindow {
    visible: true
    width: 400
    height: 640

    Rectangle{
        id: rect
        anchors.centerIn: parent
        height: 100
        width: parent.width
        color: "blue"
    }
}
```

In the above example, the width of `Rectangle` is bound to that of it's parent. If you change the width of the running application window, the width of rectangle also changes.

A more complicated example

In the simple example, we simply set the width of the rectangle to that of it's parent. Let's consider a more complicated example:

```
ApplicationWindow {
    visible: true
    width: 400
    height: 640

    Rectangle{
        id: rect
```

```
        anchors.centerIn: parent
        height: 100
        width: parent.width/2 + parent.width/3
        color: "blue"
    }
}
```

In the example, we perform arithmetic operation on the value being binded. If you resize the running application window to maximum width, the gap between the rectangle and the application window will be wider and vice-versa.

Create Bindings with Dynamically Created QML Files

When using instances of QML files by directly declaring them, every `property` creates a binding. This is explained in the above examples.

This is how you dynamically create components:

```
var component = Qt.createComponent("Popup.qml");
var popup = component.createObject(parent, {"width": mainWindow.width, "height":
mainWindow.height});
```

When the size of the `mainWindow` changes, the size of the created `PopUp` is not affected. To create a binding you set the size of the `popup` like this:

```
var component = Qt.createComponent("Popup.qml");
var options = {
    "width": Qt.binding(function() { return mainWindow.width }),
    "height": Qt.binding(function() { return mainWindow.height }),
};
var popup = component.createObject(parent, options);
```

Now the size of the `PopUp` will depend on `mainWindow`.

Read Property binding online: https://riptutorial.com/qml/topic/1967/property-binding

Credits

S. No	Chapters	Contributors
1	Getting started with qml	Akash Agarwal, Beriol, Community, CroCo, dangsonbk, jpnurmi, Mailerdaimon, Massimo Callegari, Mitch, Violet Giraffe
2	Animation	Akash Agarwal, Eluvatar, Violet Giraffe
3	Creating custom elements in C++	folibis
4	Integration with C++	Beriol, Brad van der Laan, folibis, Violet Giraffe
5	Property binding	Akash Agarwal, Eluvatar, Furkanzmc, Violet Giraffe