Qt Creator

A powerful IDE

What is Qt Creator

Qt Creator is an IDE (Integrated Development Environment).

In other words an application that have an editor you can write code in as well as the UI (User Interface).

You can also build (compile) the code.

Programming languages

- C++: The Qt Framework is written in C++ and it’s the base of a new application.
- QML: QML is a document type just like HTML or XML.
  It defines a structure.
- JavaScript: Some code can be written in JavaScript.

Kits

In order for the IDE to build a project it needs to know how.

The IDE understands how it is done by using kits.

A kit is a bunch of rules and scripts that builds a project.

How it builds depends on what kit you’re using.

More about kits

You can have multiple kits.

As standard Qt Creator comes with a kit to build a project to your native environment (the desktop, what ever operating system you’re using).

But you can add kits for mobile OS’s like Android, iOS, Blackberry, Windows Phone or embedded Linux like Raspbian or Linaro.

There are support for a few other OS’s like embedded Windows PE and Windows RT.

The code C++

When creating a Qt Quick Project a main.cpp looks like this:

```
#include <QApplication>
#include <QQmlApplicationEngine>

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

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

   return app.exec();
}
```

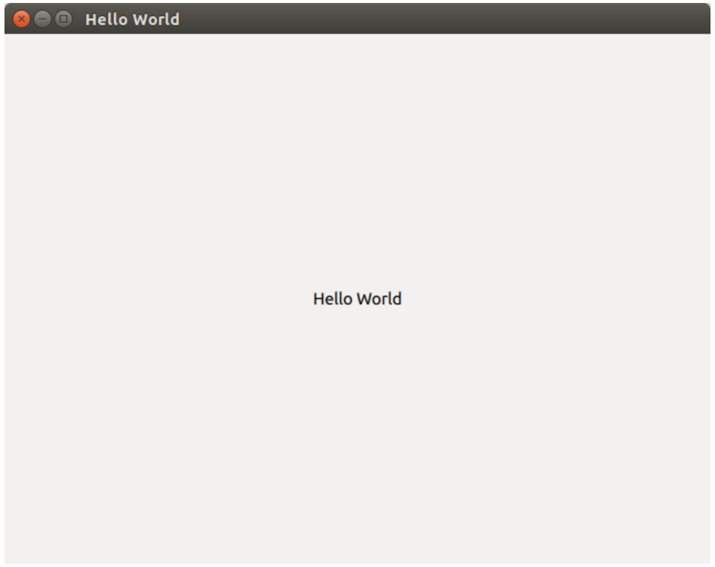
The code QML

A main.qml looks like this: *There’s actually some code between the braces after MenuBar*

```
import QtQuick 2.2
import QtQuick.Controls 1.1

ApplicationWindow {
   visible: true
   width: 640
   height: 480
   title: qsTr("Hello World")

   menuBar: MenuBar { … }

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

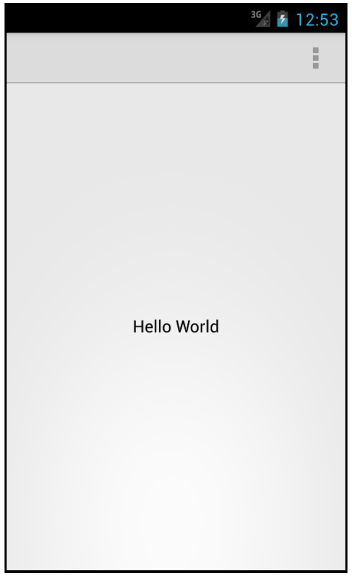
The code result

In Ubuntu it looks like this:



```
Hello World
```

On an Android device it looks like this:



```
Hello World
```

JavaScript?

There's no JavaScript in a new Qt Quick Project from the beginning.

JavaScript can be used for handling QML events:

```
MouseArea {
    onClicked: {
        // This is JavaScript
        console.log("This event logs when someone clicks in an mouse area");
    }
}
```

The basics of QML

A simple declaration of an rectangle can look like this:

```
Rectangle {
  id: rectangle
}
```

Above there is a short QML code that creates an instance of the type "Rectangle" and sets the property "id" to "rectangle".

Like HTML all types have an "id" property and it's optional.

QML property events

Bringing back the “rectangle”, write the following:

```
Rectangle {
  id: rectangle
  width: 200
  height: 300

  onWidthChanged: {
    console.log("The width is now: "+width);
  }
}
```

Whenever the width of “rectangle” changes it will be logged in the console.

QML declaring properties

Declaring a property.

```
Rectangle {
  id: rectangle
  property var myOwnProperty : "This is a default value, now it’s this text string"
}
```

QML property events?

```
Rectangle {
  id: rectangle
  property var myOwnProperty : "This is a default value, now it’s this text string"

  onMyOwnPropertyChanged: {
     console.log("myOwnProperty has changed to: "+myOwnProperty);
  }
}
```

Writing “on” in front of the name, “Changed” at the end and capitalizing the first letter?

QML automatically declares this event and calls it when the value have changed.

QML dynamic properties?

So far the properties have had static values like 300 or “a string with text”.

What do you do if you want a property to change every time another property changes?

Any idea?

QML dynamic properties

```
Rectangle {
   id: rectangle
   property var side : 200

   onSideChanged: {
       width = side; // we could write rectangle.width if we wanted and we would get the same result
       height = side;
   }
}
```

QML dynamic properties!

```
Rectangle {
  id: rectangle
  property var side : 200
  width: side
  height: side
}
```

Wow! easy… but what is going on?

Lets break it down, let us pretend someone changes “side” by writing “Square { side: 300 }” in another qml file. When “side” changes the change propagates to all the related properties.

“width” and “height” will be set to the same as “side”, 300.

QML nested objects

You can nest objects by simply creating an object within another.

```
Rectangle {
  id: outerRectangle
  Rectangle {
    id: innerRectangle
  }
}
```

QML the finale

Only one more thing to say about QML?

But it have been time for C++.

C++ Qt specific code

I expect you to have basic knowledge in C++.

Do Qt have a special C++ version?

No, Qt uses the g++ compiler by default.

But C++ kan have macros. A macro is a script that runs before the code is compiled. This can be used to simplify the code.

C++ QObject

QObject is a very basic class the Qt framework.

If you want to be able to use a new C++ class in the QML environment it must follow a few rules.

- It must inherit QObject.
- It must have the “Q_OBJECT” macro immediately after the opening bracket “{“ of the class definition in the header.
- It must run the super class’s constructor.

Ready to see what a class that inherits QObject might look like?

C++ inheriting QObject

```cpp
#ifndef MYCLASS_H
#define MYCLASS_H

#include <QObject>

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

signals:

public slots:

};

#endif // MYCLASS_H
```

```cpp
#include "myclass.h"

MyClass::MyClass(QObject *parent) :
  QObject(parent)
{
}
```

“MyClass” is a class that is inheriting “QObject”.

“myclass.h” to the left.
“myclass.cpp” to the right.

C++ signals and slots

```
...
class MyClass : public QObject
{
  Q_OBJECT
public:
  explicit MyClass(QObject *parent = 0);

signals:
  void mySignal();

public slots:
  void myPublicSlot();

private slots:
  void myPrivateSlot();

};
...
```

```
...
void MyClass::myPublicSlot()
{

}

void MyClass::myPrivateSlot()
{

}
```

To send a signal the keyword “emit” is used like this:

```
  emit mySignal();
```

C++ connections

```
MyClass *object1 = new MyClass();
MyClass *object2 = new MyClass();
```

The old syntax:

```
connect(object1, SIGNAL(mySignal()), object2, SLOT(myPrivateSlot()));
```

The new syntax:

```
connect(object1, &MyClass::mySignal(), object2, &MyClass::myPrivateSlot());
```

C++ and QML linkage

```
class MyClass : public QObject
{
  Q_OBJECT

  Q_PROPERTY(int myProperty READ myProperty WRITE setMyProperty NOTIFY myPropertyChanged)

public:
  Q_INVOKABLE void myMethod();
  int myProperty();
  void setMyProperty();

signals:
  void myPropertyChanged();
…
```

C++ and QML linkage

To link a C++ class to QML one can use the “qmlRegisterType” method.

In C++:

```
qmlRegisterType<MyClass>("MyApplication.MyClasses", 1, 0, "MyClass");
```

In QML:

```
import MyApplication.MyClasses 1.0

MyClass {
   myProperty: 400
}
```

The end