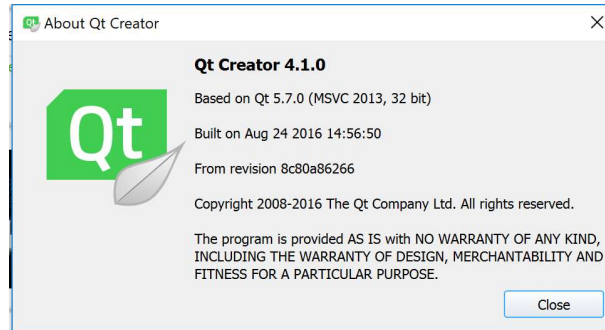


Tutorial

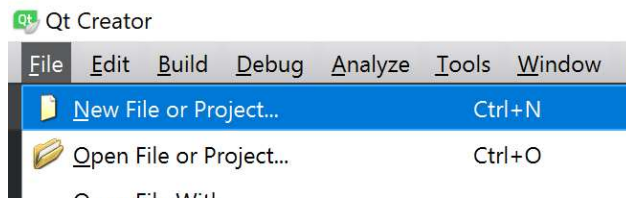
Introduction to QML

Peter Jackson

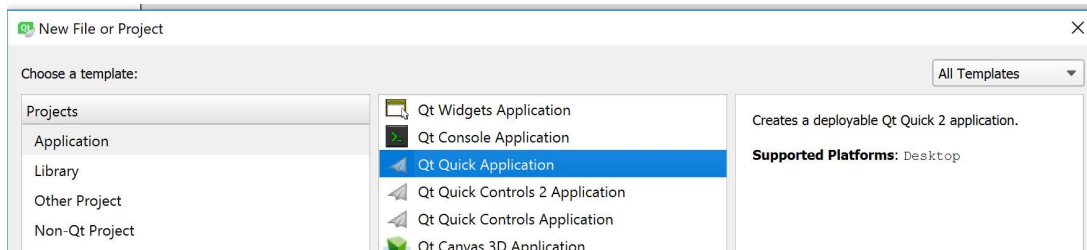
This tutorial assumes you have successfully installed the Qt platform. The screenshots are taken from version 5.7.0:



Let's start by creating a new project:



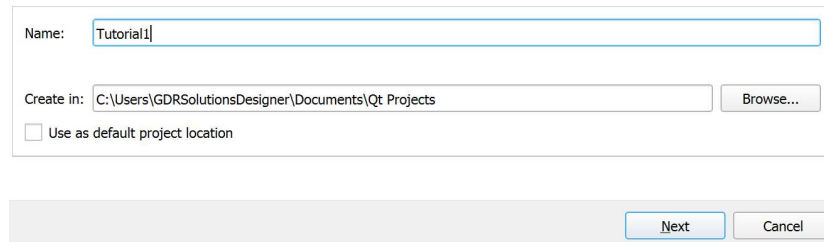
Choose a Qt Quick Application to build:



Give your project a name and a folder location:

Project Location

Creates a deployable Qt Quick 2 application.



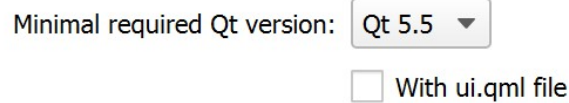
Name:

Create in:

☐ Use as default project location

Click “Next”. I recommend you uncheck the box “With ui.qml file” because we will not use the screen design editor feature.

Define Project Details



Minimal required Qt version:


☐ With ui.qml file


I chose the 64 bit kit option:

Kit Selection

Qt Creator can use the following kits for project **Tutorial1**:

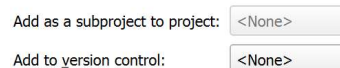
☒ Select all kits

☐  Desktop Qt 5.6.2 MSVC2015 64bit

☒  Desktop Qt 5.7.0 MSVC2015_64bit

Next, I left the project management options blank:

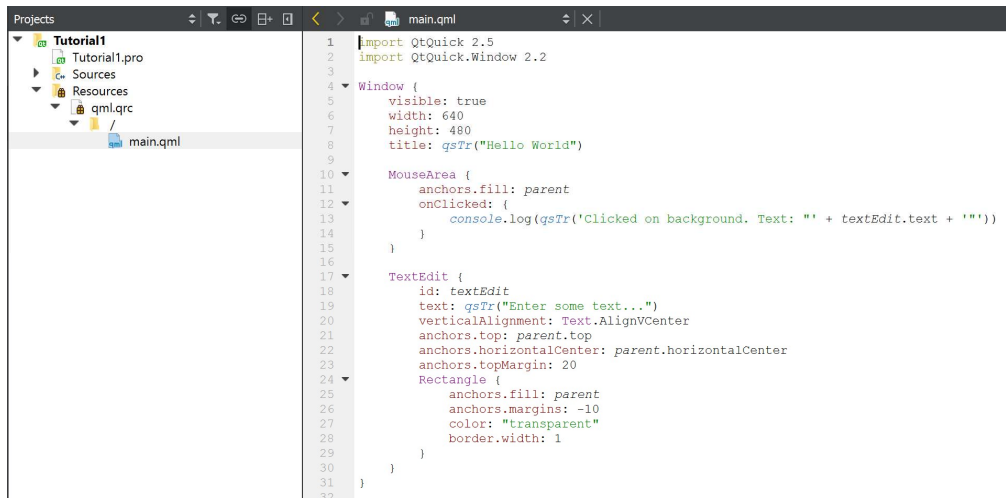
Project Management



Add as a subproject to project:

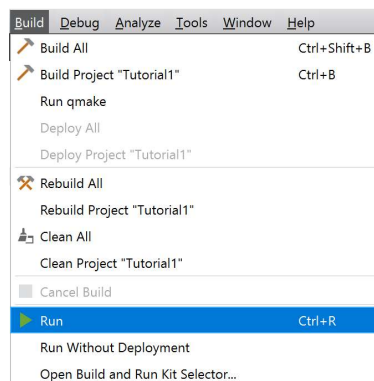
Add to version control:

Click “Finish.” The project should appear in the project section of the window, and the file “main.qml” should be open in the editor:



In general, we can build our applications using just qml files so you can ignore all the other files in the project (such as Tutorial1.pro). You can see that the main.qml file is quite simple.

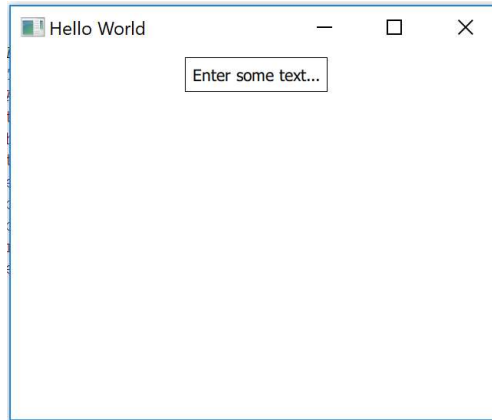
This is a working application. To run it, select Build->Run:



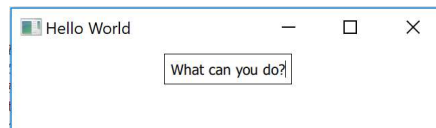
Alternatively, click the run button in the lower left of the screen:



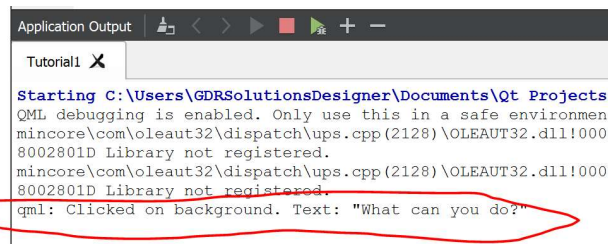
This should launch the application in a new window:



Note that the window has a banner name (“Hello World”) and basic functionality (minimize, maximize, and close). It also contains a textbox in which you can type text. Enter some text (eg. “What can you do?”)



and then click somewhere in the window outside of the textbox. In the Qt window, Application Output section, you should see a message like this:



That is all that this application can do. Close the “Hello World” application.

Let’s understand how the main.qml file defined this application. It lists some imports:

```
import QtQuick 2.5
import QtQuick.Window 2.2
```

These are libraries of objects we can use in our project. For example, you can see that the initial project already uses four objects “Window,” “MouseArea,” “TextEdit,” and “Rectangle.” There are many useful objects in these libraries and we can import other libraries with more objects.

QML is a markup language. That is, you describe your application as a collection of objects and objects nested within other objects. You can see the basic structure of this application is:

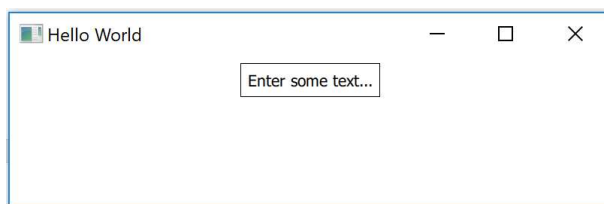
```
Window{
    MouseArea{}
    TextEdit{}
}
```

The object “Window” contains two objects: “MouseArea” and “TextEdit.” The curly braces define the contents of each object.

Objects have properties (eg. height, width, color), events or signals (eg. “onClicked”), and methods (eg. console.log() prints a message on the Application Output window). If you want to change the appearance of each object you simply need to change its properties. For example to make the window a different shape, change the height and width properties:

```
Window {  
    visible: true  
    width: 800  
    height: 200  
}
```

This results in a short fat window:



To find more properties to change for an object, simply Google it to find the documentation:



This looks promising:

Window QML Type | Qt Quick 5.9 - Qt Documentation

doc.qt.io/qt-5/qml-qtquick-window-window.html Proxy Highlight

Omitting this import will allow you to have a QML environment without access to window system features. A Window can be declared inside an Item or inside ...

Sure enough, it takes me to a list of properties, events (“signals”), and methods for the Window object:

Properties

- > **active** : bool
- > **activeFocusItem** : Item
- > **color** : color
- > **contentItem** : Item
- > **contentOrientation** : Qt::ScreenOrientation
- > **data** : list<Object>
- > **flags** : Qt::WindowFlags
- > **height** : int

I am not sure what all these properties are for, but color looks useful:

```
color : color
```

The background color for the window.

Setting this property is more efficient than using a separate Rectangle.

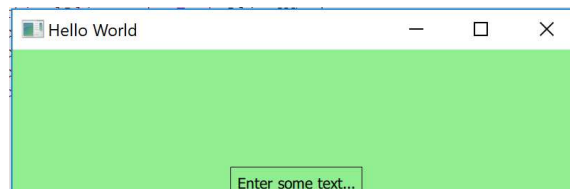
So let's change the background color of our Window:

```
Window {  
    visible: true  
    width: 800  
    height: 200  
    color: "lightgreen"  
    title: qsTr("Hello World")  
}
```

Run the application to test:



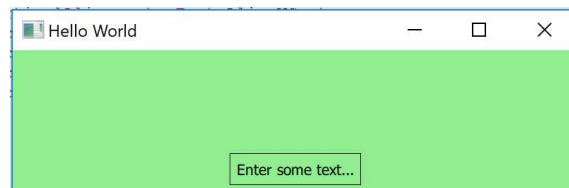
Other properties that are useful include the positioning properties. The `TextEdit` object is a child of the `Window` object (the `Window` object is the parent of the `TextEdit` object). The anchor properties of the `TextEdit` object position it within the parent. For example, change the property `anchors.top: parent.top` to `anchors.bottom=parent.bottom`. This should be the result:



That is not attractive. So, change the bottom margin:

```
TextEdit {  
    id: textEdit  
    text: qsTr("Enter some text...")  
    verticalAlignment: Text.AlignVCenter  
    anchors.bottom: parent.bottom  
    anchors.horizontalCenter: parent.horizontalCenter  
    anchors.bottomMargin: 20  
}
```

This looks better:

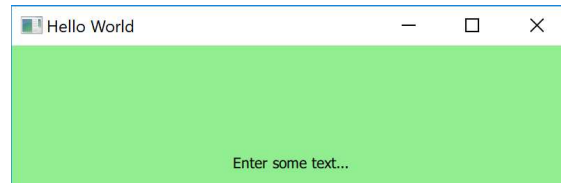


To learn more about anchors and positioning objects, search for “qml anchors” or “qml positioning.”

What is the purpose of that Rectangle object inside the TextEdit object?

```
Rectangle {
    anchors.fill: parent
    anchors.margins: -10
    color: "transparent"
    border.width: 1
}
```

Try cutting it to clipboard, running the application without it, and then pasting it back in.



Ok, it was just providing the visual border to the TextEdit object.

So, a lot of user interface design is simply placing objects on the screen and controlling their appearance. With a markup language such as QML that is handled with objects and their properties. The next topic is behavior.

Behavior

To be interesting, the application must be interactive: it must detect events and respond to them. Let's focus on the MouseArea, whose purpose is to detect mouse events.

```
MouseArea {
    anchors.fill: parent
    onClicked: {
        console.log(qsTr('Clicked on background. Text: ' + textEdit.text + ''))
    }
}
```

The size of the MouseArea fills the Window (“anchors.fill: parent”). When a mouse click is detected in the MouseArea it will trigger the actions associated with the onClicked property. Currently, that is simply to log a message on the console. The function “qsTr()” is useful if you are creating an application with multi-language support. We are not doing that so let's delete that function:

```
MouseArea {
    anchors.fill: parent
    onClicked: {
        console.log('Clicked on background. Text: ' + textEdit.text + '')
    }
}
```

Run the application to make sure it still works.

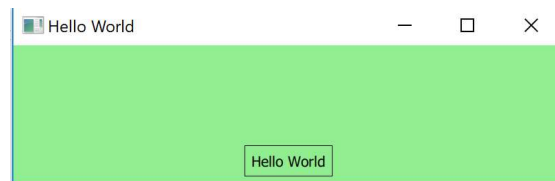
The property “onClicked” expects to be given a function to execute. The code in braces after “onClicked:” is not QML: it is Javascript (console.log is a javascript function). So we are switching from QML, the markup language, to javascript whenever a property of QML expects a function. Javascript is a nice scripting language that is also useful in web development. It has a different syntax than QML. In

javascript you provide a series of statements ending in semicolons (“;”) to be performed in sequence. Let’s add another javascript statement to the sequence:

```
MouseArea {
    anchors.fill: parent
    onClicked: {
        console.log('Clicked on background. Text: ' + textEdit.text + '');
        textEdit.text = "Hello World";
    }
}
```

Note that I added a semicolon to properly terminate the first statement before adding the second statement.

Run the application and click in the MouseArea:



The text in the TextEdit area should change to “Hello World”.

Observe how we are able to interact with different objects by referring to their id. The id of the TextEdit object is “textEdit”. We can both get and set the text property of the TextEdit object by referring to “textEdit.text”

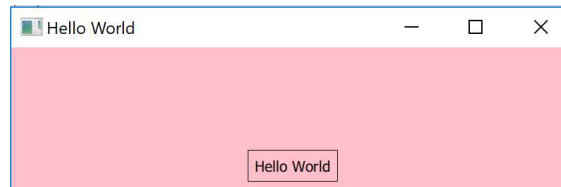
Suppose we wanted to change the background color of the window to pink whenever the user clicks on it? We just need to add a javascript statement that changes the color to “pink”. But how do we refer to the Window? We must give it an id:

```
Window {
    id: myWindow
    visible: true
    width: 800
    height: 200
    color: "lightgreen"
    title: qsTr("Hello World")
}
```

Now, with the id we can change its color:

```
MouseArea {
    anchors.fill: parent
    onClicked: {
        console.log('Clicked on background. Text: ' + textEdit.text + '');
        textEdit.text = "Hello World";
        myWindow.color = "pink";
    }
}
```

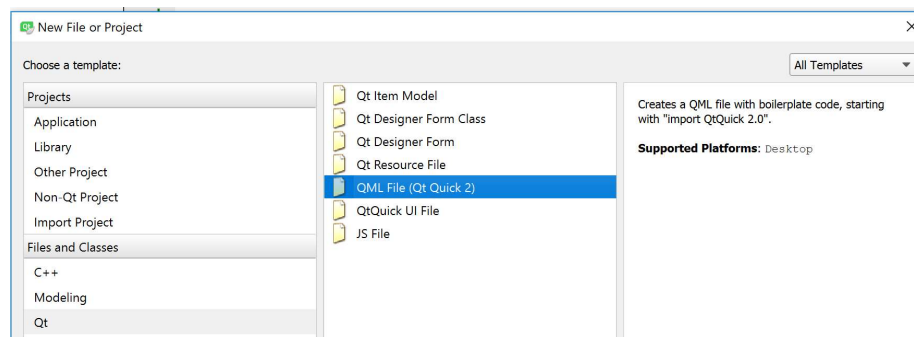
Run and click to test:



Components

We have seen that objects have properties and methods and we can create interactive applications. Another valuable idea is that we can create objects out of objects and then replicate them in our application. Let's demonstrate that concept. We will build a component, an object that stands alone, and then use it in our application.

To begin, create a separate qml file as follows:



Let's call our new component "TBox":

Location

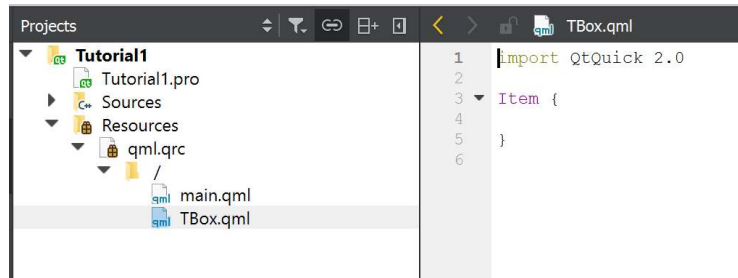
| | |
|-------|--|
| Name: | <input type="text" value="TBox"/> |
| Path: | <input type="text" value="C:\Users\GDRSolutionsDesigner\Documents\Qt Projects\Tutorial1"/> |

Accept the default Project Management names:

Project Management

| | |
|-------------------------|--|
| Add to project: | <input type="text" value="qml.qrc Prefix: /"/> |
| Add to version control: | <input type="text" value="<None>"/> |

This gives us a mostly empty file, "TBox.qml," to begin with:



Put a colored rectangle into Item{} object:

```
import QtQuick 2.0

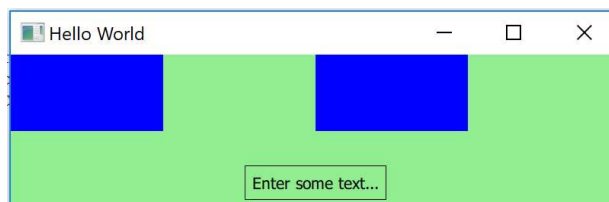
Item {
    Rectangle{
        height: 100
        width: 200
        color: "blue"
    }
}
```

Save the file ("Ctrl-S") and switch to the main.qml file. Add two new elements to the window based on the TBox component.

```
Window {
    id: myWindow
    visible: true
    width: 800
    height: 200
    color: "lightgreen"
    title: qsTr("Hello World")

    TBox{
        anchors.left: parent.left
        anchors.top: parent.top
    }

    TBox{
        anchors.left: parent.horizontalCenter
        anchors.top: parent.top
    }
}
```



This demonstrates that we can define a component once and then re-use it multiple times. But suppose we want to make adjustments to it each time we use it? For example, suppose we want to have control over what color it is?

We can define properties for our components by adding a property statement. The format of the statement is

Property <type> <name>

Or

Property <type> <name>:<default value>.

For example, we can add a property named “tboxcolor” of type “color” to our component as follows:

```
TBox.qml*
import QtQuick 2.0

Item {
    property color tboxcolor
    Rectangle {
        height: 100
        width: 200
        color: tboxcolor
    }
}
```

Note that we bind that property to the color property of the Rectangle. Binding means that whenever the value tboxcolor changes the color of the Rectangle will change.

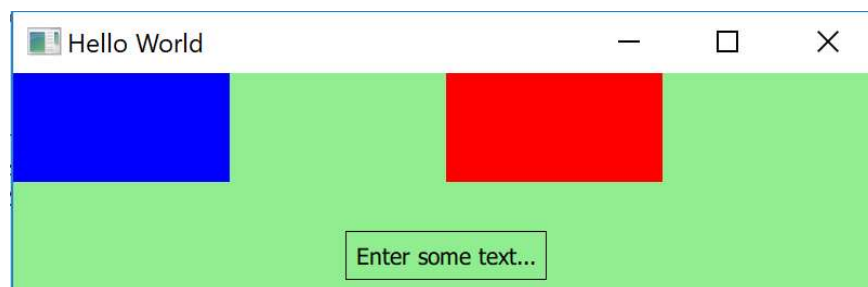
Next we modify our application to set the “tboxcolor” property of each TBox:

```
Window {
    id: myWindow
    visible: true
    width: 800
    height: 200
    color: "lightgreen"
    title: qsTr("Hello World")

    TBox {
        anchors.left: parent.left
        anchors.top: parent.top
        tboxcolor: "blue"
    }

    TBox {
        anchors.left: parent.horizontalCenter
        anchors.top: parent.top
        tboxcolor: "red"
    }
}
```

So now we have color control over the Tbox objects:



Hopefully these basic concepts will help you understand how QML applications are built.