Qt Quick for Mobile

23.-24.2.2011

Tuukka Ahoniemi
Senior Technical Trainer
Digia Plc

Forum Nokia

NOKIA

digia ?

Qt Alliance Partner

Forum Nokia

- That's where me and ~1499 other computerish professionals work
  - Global, based in Finland
- ~50 % work with mobile technologies
- A large, traditional Symbian house
- Since day 1, heavily involved with Qt on mobile devices
- Qt Training Partner

- **http://www.digia.com**

Course Logistics

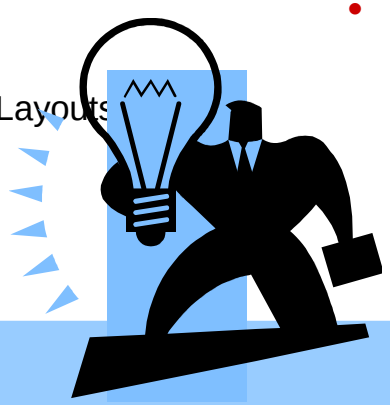
- Welcome
- Trainer
- Facilities
- Course timings
- Refreshments & lunch 

Course Objectives

- To get a good kick-start on developing applications to Nokia platforms with Qt Quick
- To get a really good overview on what is there in the Qt developer offering for one to use

Course Contents 1/2

- **Qt Update & Status**
  - The Big picture
- **Qt Mobile UI Offering**
  - Comparing approaches
  - What is Qt Quick?
  - Development Tools
- **QML Essentials**
  - Basic Syntax
  - Properties
  - Standard QML Elements
  - Property Binding
- **More Layouts**
  - Grid, Row, and Column Layouts



- **User Interaction**
  - Mouse Area
  - KeyNavigation, Key Events
- **States, Transitions and Animations**
  - States
  - State Conditions
  - Transitions
  - Animations
  - Property Behaviour
  - Timers
- **Core QML Features**
  - QML Components
  - Inline Components
  - Modules

Course Contents 2/2

- **Extending QML Components**
  - Extending types with QML
  - Adding new properties, signals, methods
- **Data Models and Views**
  - Model Classes
  - ListView, GridView, PathView
  - Repeater
  - Flickable
- **Using QML in Qt/C++ Applications**
  - Main Classes of QtDeclarative
  - Structured Data
  - Dynamic Structured Data
- **C++ Data Models in QML**
  - QList<QObject*>
  - QAbstractItemModel
  - QStringList

Qt Update & Status

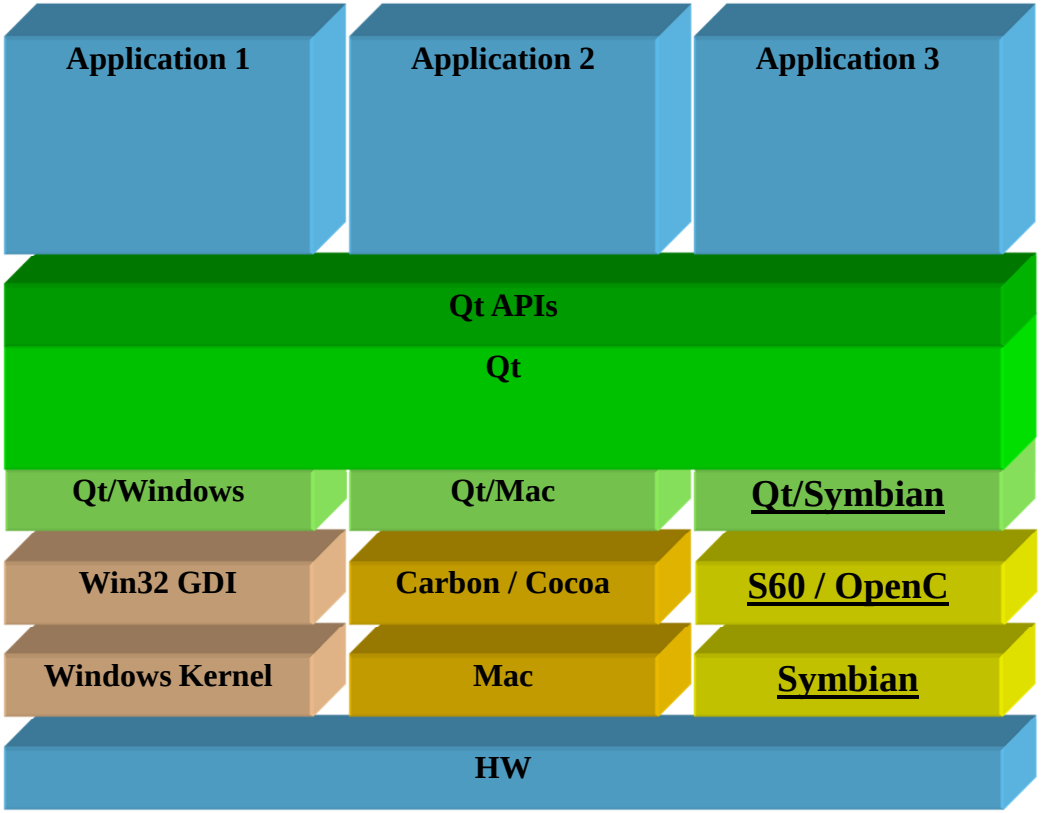
Forum Nokia

NOKIA

What is Qt?

Cross-Platform APIs

- Cross-platform Qt APIs are wrappers around native services



Different Qt Platforms

One and unified Cross-Platform API						
Qt/X11	Qt/MeeGo	Qt/Win	Qt/Mac	Qt/Embedded Linux	Qt/WinCE	Qt/Symbian

What is Symbian^3?

- Actually, just Symbian (or the New Symbian)
- The latest Symbian platform release
  - Multiple home screens
  - Multi-point touch
  - HD multimedia capabilities
  - Improved multitasking
  - Improved graphics architecture
  - *Qt libraries pre-installed!*

- Nokia N8 is the first Symbian^3 device
  - Other phones: C7, C6-01, E7

Qt and the Nokia N8

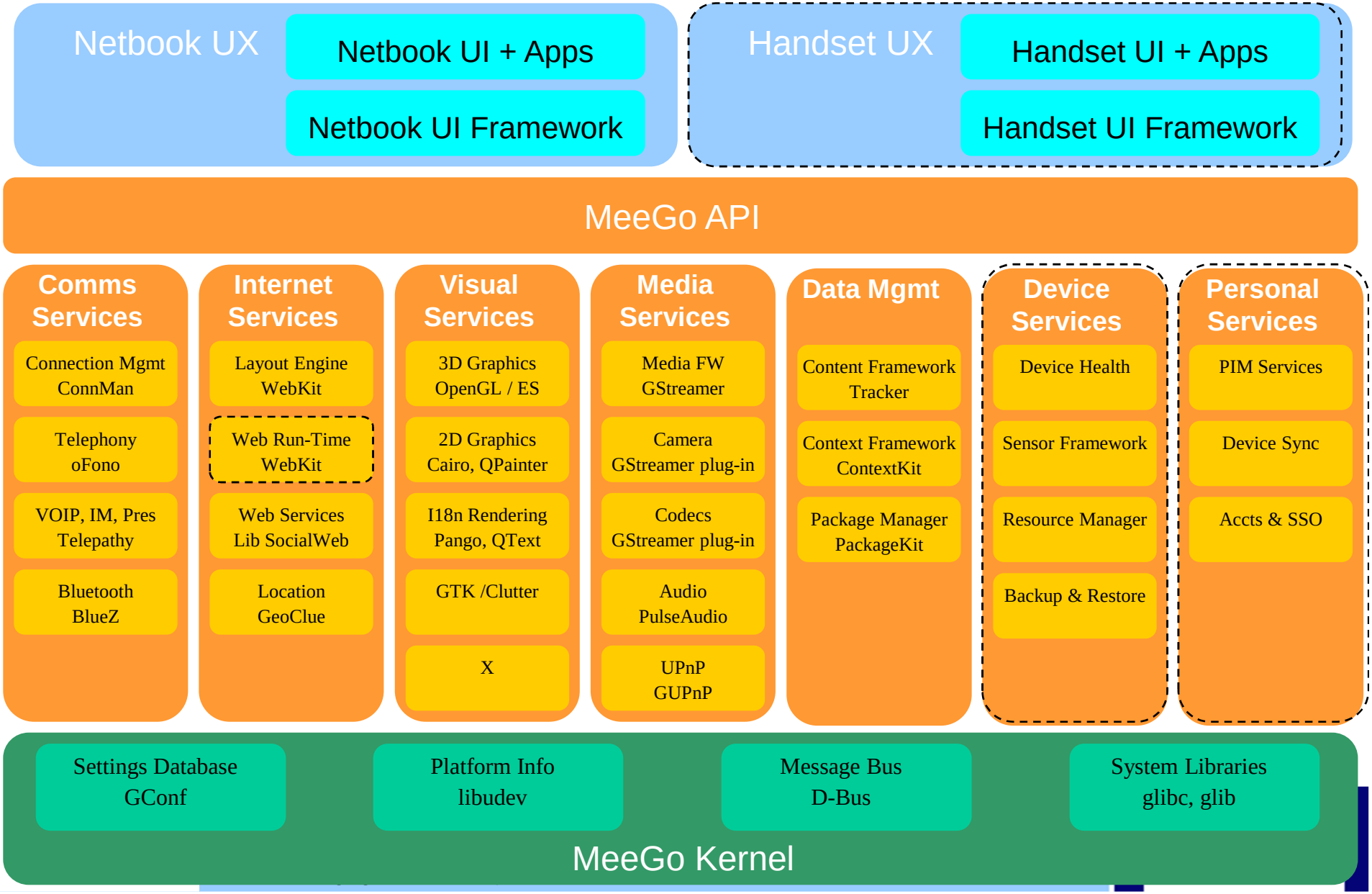
- The first Symbian^3 device
- Qt 4.6.3 (4.6.4) pre-installed
  - Second wave of devices will have Qt 4.7
- Capasitive touch screen
  - Multi-touch and gestures work
- A separate GPU!
  - HW accelerated graphics
  - QtOpenVG rasterization mode

What is MeeGo?

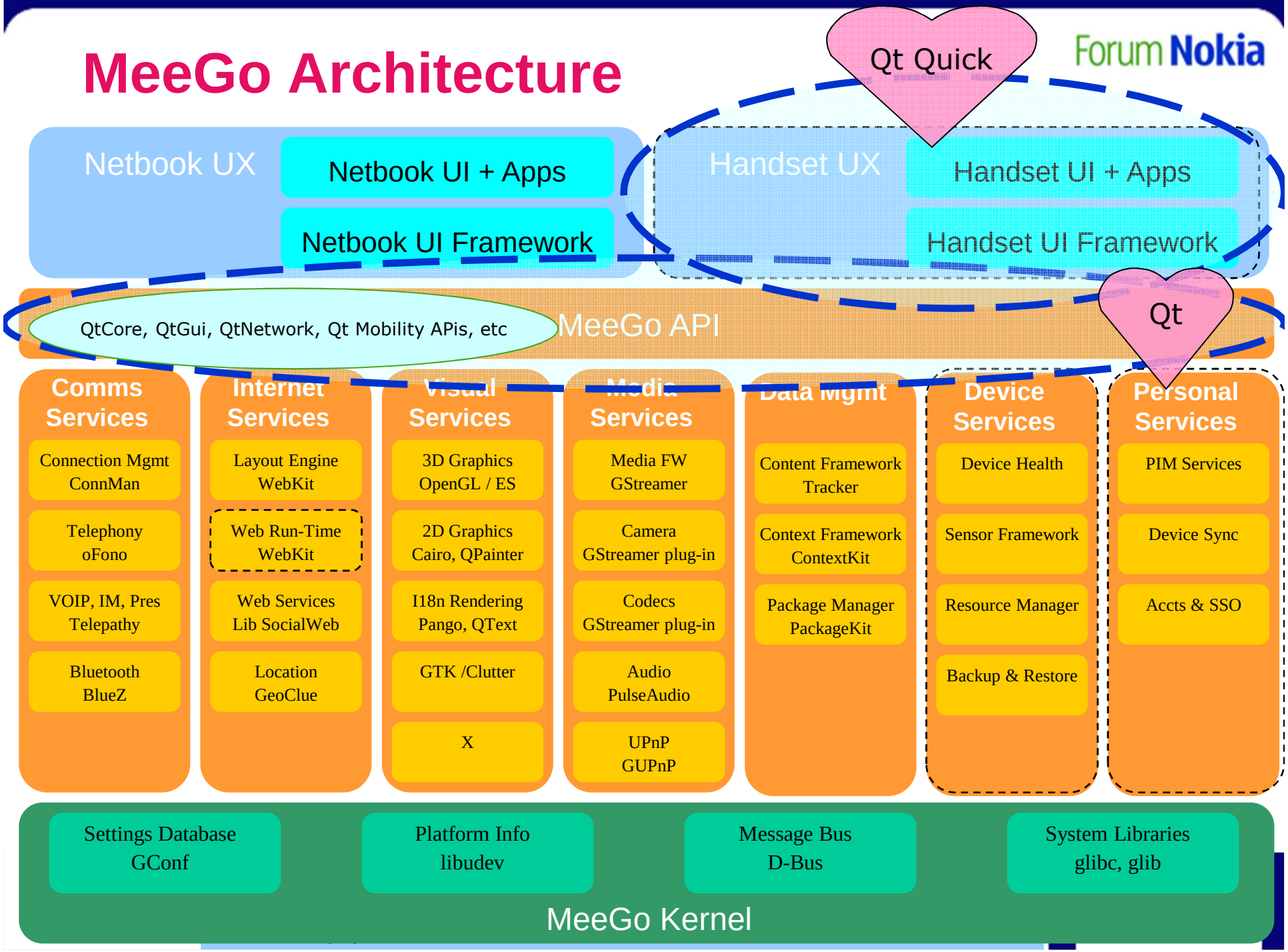
Forum Nokia

MeeGo Architecture

Forum Nokia

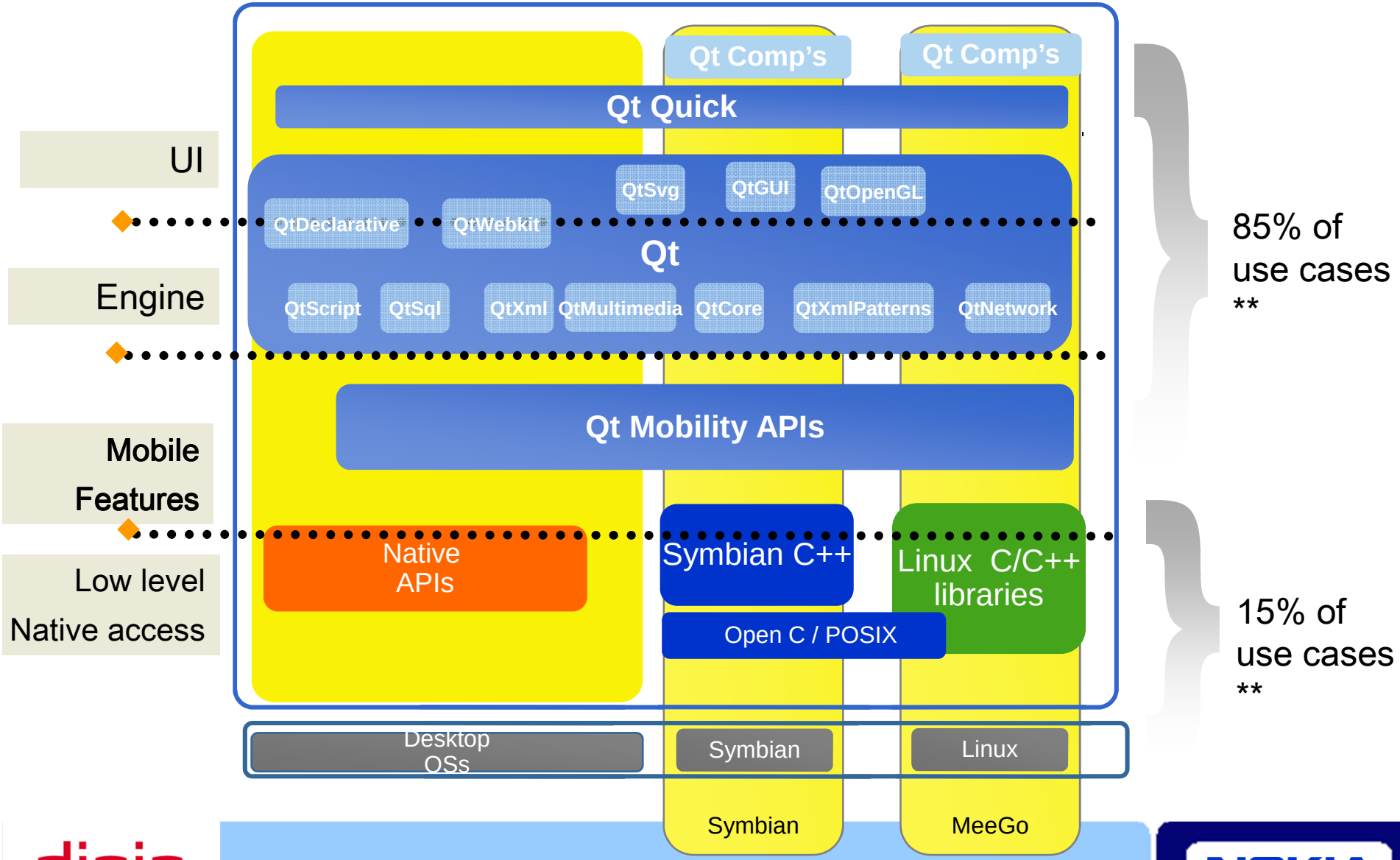


MeeGo Architecture



Qt Developer Offering

Forum Nokia

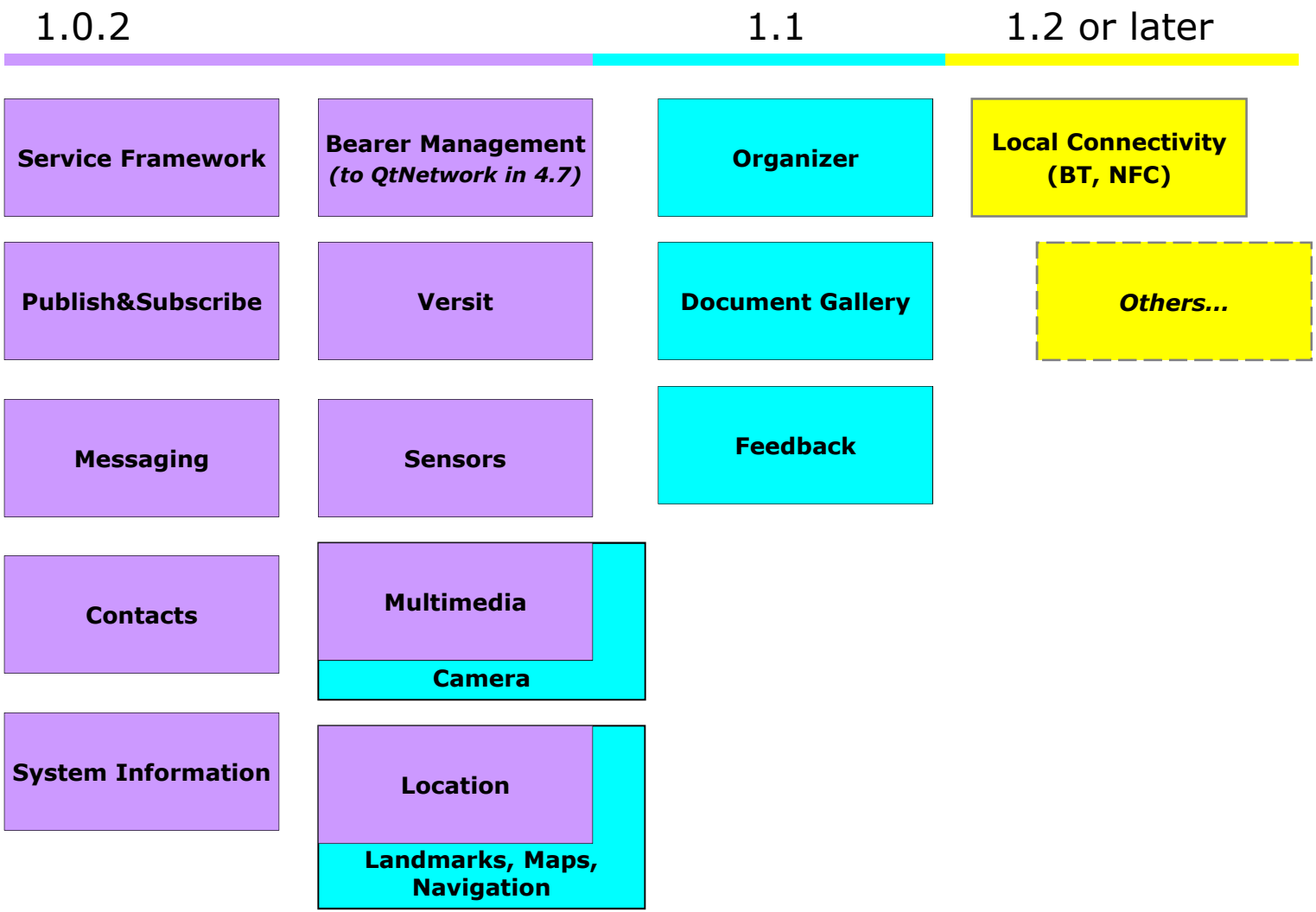


digia

NOKIA

Mobility API Roadmap

Forum Nokia



Mobile Qt UI Offering

Comparing different approaches

Forum Nokia

NOKIA

Mobile Qt UI Offering

Forum Nokia

QWidgets

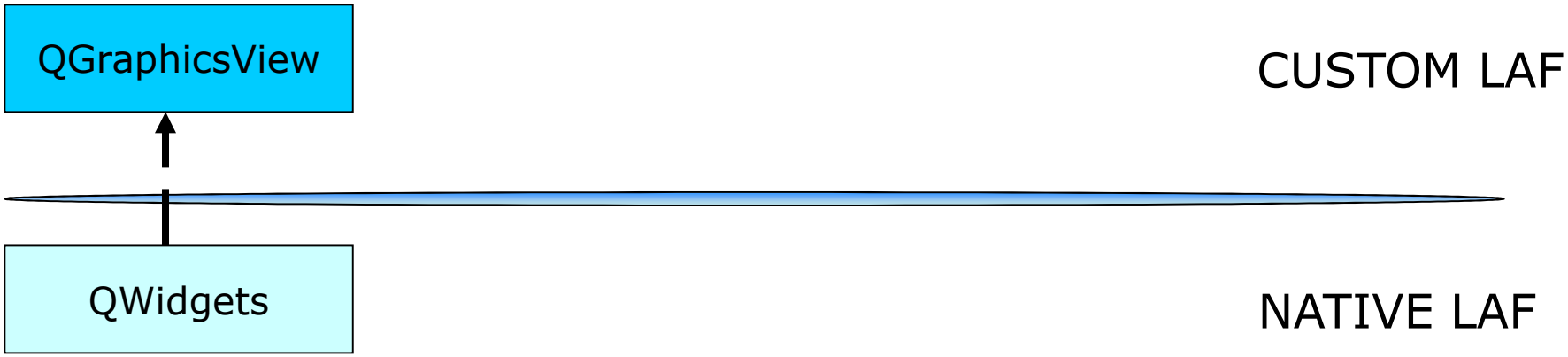
NATIVE LAF

Native Look-and-Feel w/ QWidgets

Forum Nokia

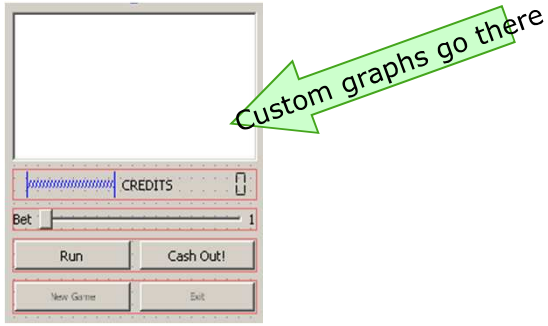
- Code once, look native everywhere
- Pretty straightforward
- Can customize, a bit
  - **Style sheets**
  - Widget properties
  - Own custom widgets
- Complete customization not easy/possible
  - Games, completely different kind of UIs
- Quite desktop-oriented
  - Missing mobile concepts, like views
  - Not really there yet, at least in S^3

Mobile Qt UI Offering

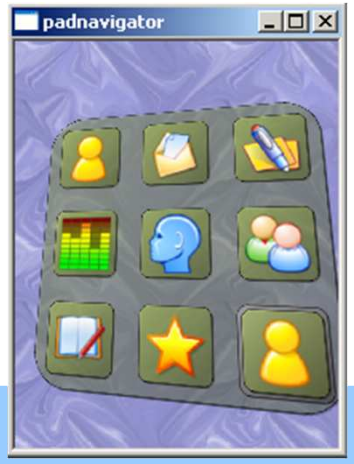


Custom UI w/ QGraphicsView

- QGraphicsView is a QWidget, designed for showing custom 2D graphics:

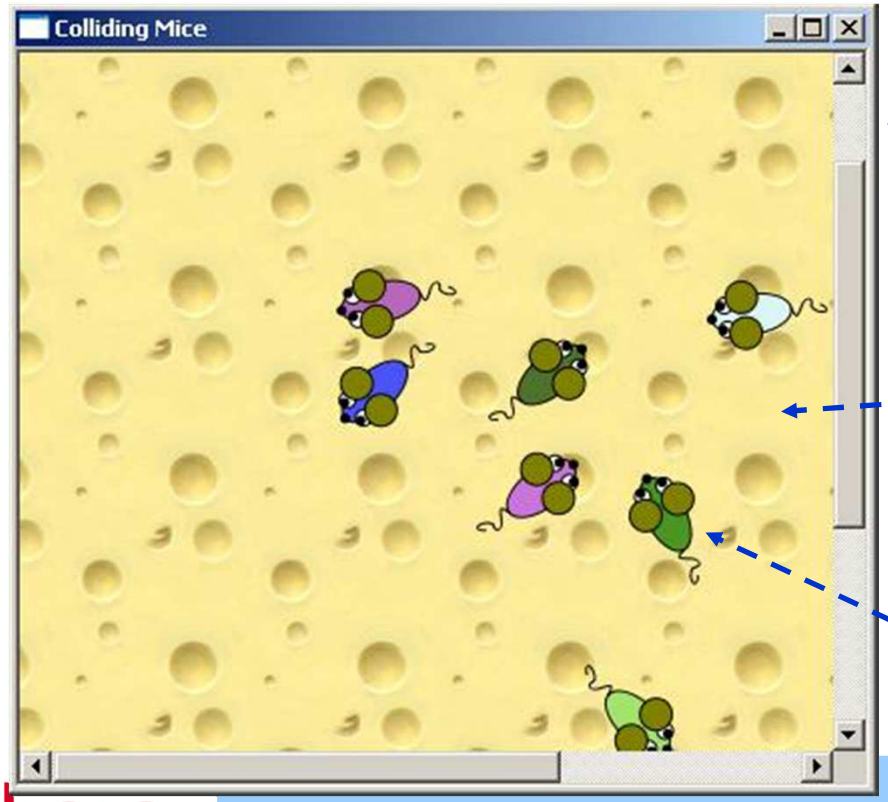


- If the only widget (the window itself) is a QGraphicsView, the whole UI is then custom:



Graphics View Architecture

- Actually, inside a QGraphicsView, lies an architecture of its own
  - Super-duper optimized for doing everything fast and being flexible



QGraphicsView

Shows contents of one

QGraphicsScene

is a 2D container for

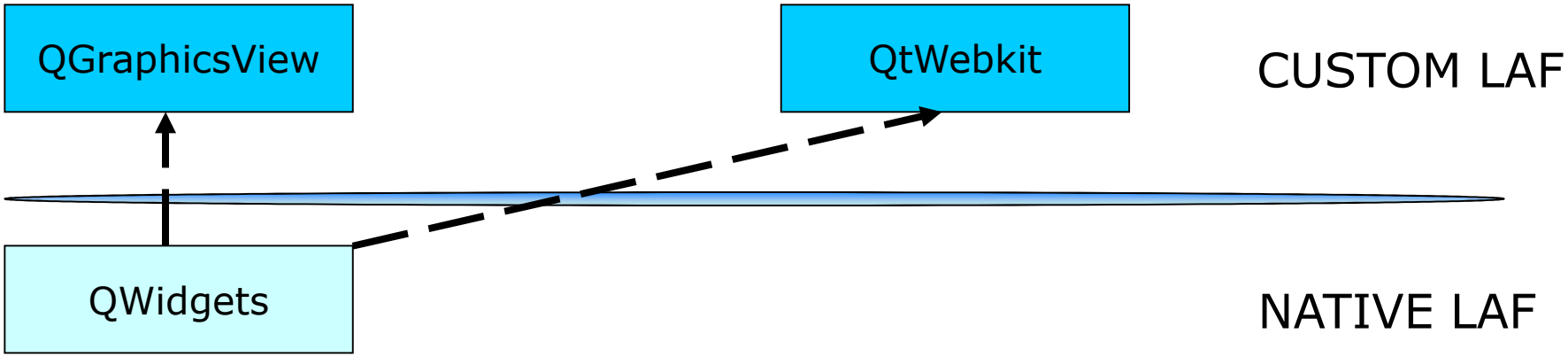
QGraphicsItem

Working with GraphicsView

- You can do anything by doing everything!

- Create a custom graphics item:
  - Derive from QGraphicsItem (or some other QGraphics* base class)
  - Write code for painting operations with QPainter
  - Write code for event handling
  - Write code for animations
  - etc.

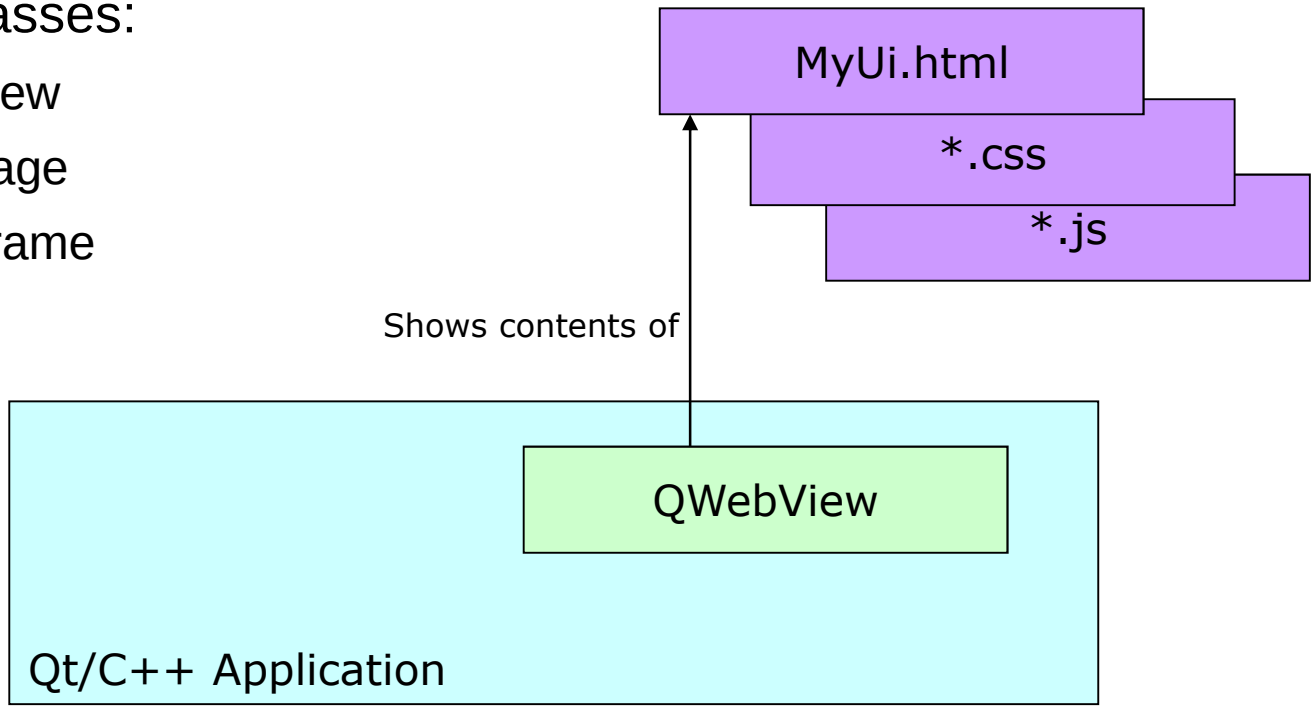
Mobile Qt UI Offering



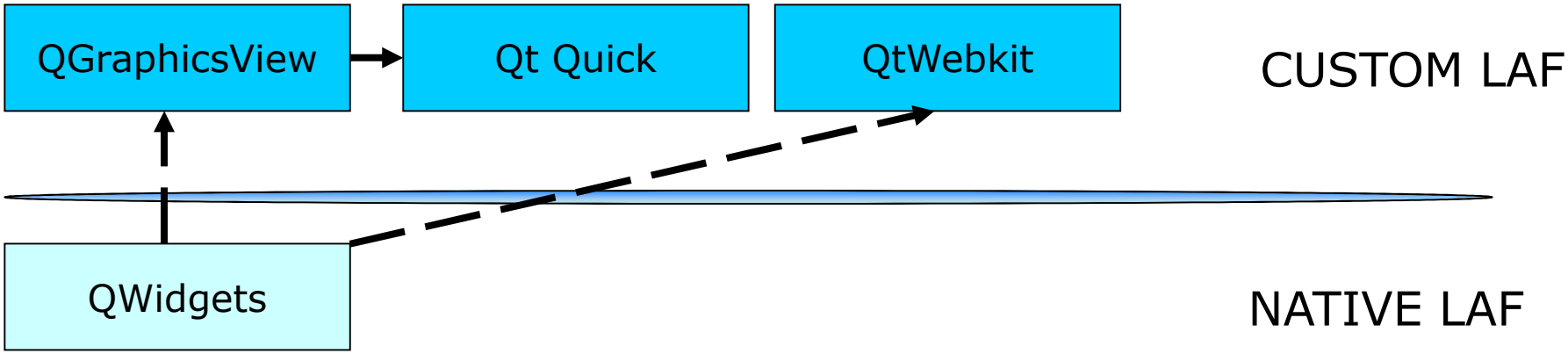
Hybrid Apps with QtWebkit

- Write UI (+logic) with standard web technologies
- Embed in a Qt/C++ Application

- Key C++ classes:
  - QWebView
  - QWebPage
  - QWebFrame

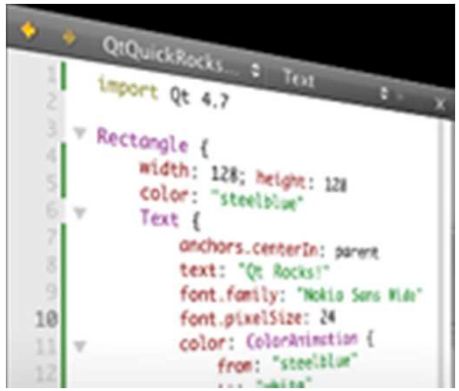


Mobile Qt UI Offering



What is Qt Quick?

Forum Nokia



QML

Qt

Quick

Qt Creator 2.1

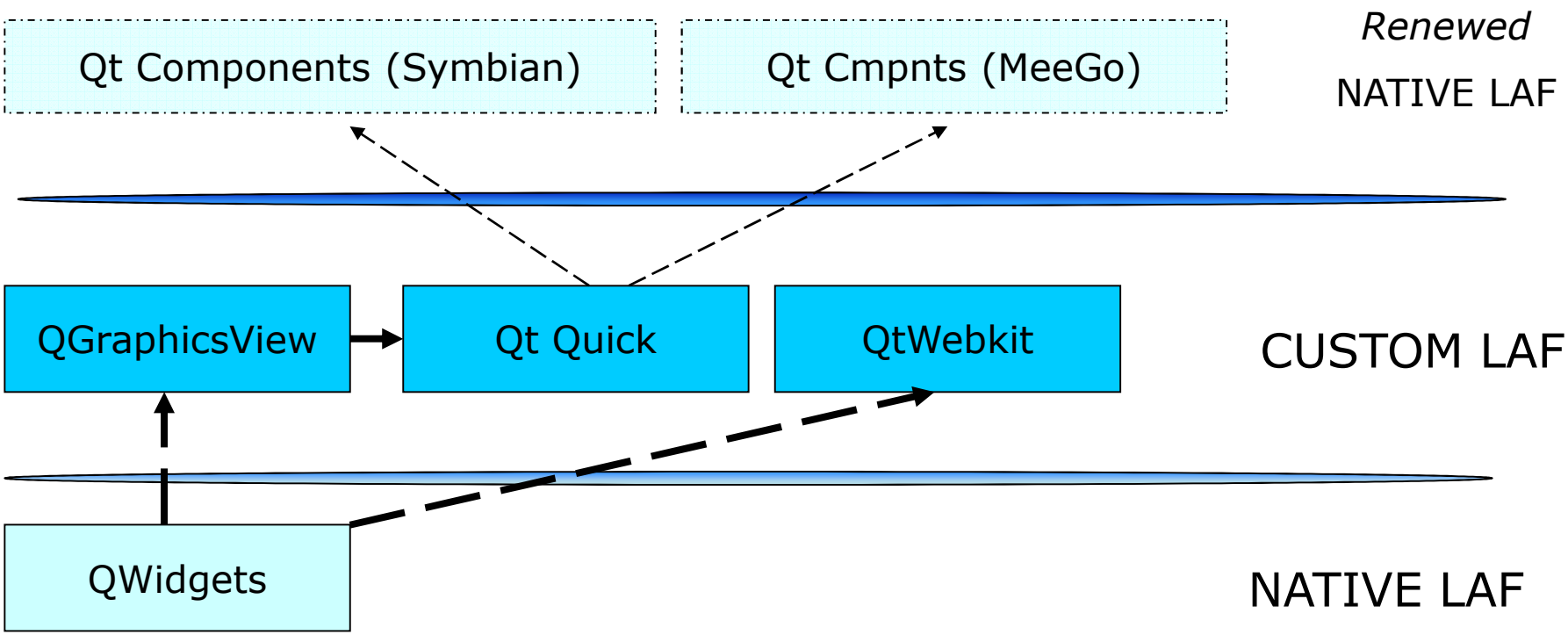
QtDeclarative

Demos


Back
Flowd
Digia Lappeenranta
Flow
Places
Profile


New Game
Quit
Score: 1006

Mobile Qt UI Offering, Later…



Qt SDK 1.1 *(former Nokia Qt SDK)*

Forum Nokia

- Since 4/2010 the Nokia Qt SDK has been available for all Qt-related development
  - Available for Windows, Linux & Mac
- Includes
  - Qt **4.7.1** for Symbian, Maemo and desktop
  - QtMobility APIs 1.1.0
  - Qt Creator IDE 2.1 RC
  - Qt Simulator
  - ODD for all devices
  - Remote compilation service
    - Mainly aimed at Linux and Mac users wishing to compile for Symbian
    - However, can be used to compile for any supported target device from any Nokia Qt SDK desktop environment!

Purpose of Qt SDK 1.1

Forum Nokia

- All Nokia platforms under one tool
  - Symbian, Maemo, MeeGo
  - Desktop (from 1.1 onwards)
  - *Install once, deploy everywhere*
- Compile, deploy, execute and debug applications on all Nokia **devices** by just one click
- Also supports additional SDKs
  - Separate Symbian SDKs with Qt libraries

Qt Quick

QML Essentials

Forum Nokia

NOKIA

QML

- ”Qt Meta-Object Language”, maybe…
- A declarative, script-like language for defining the elements of a graphical UI
  - Actually an extension to ECMAScript (cf. JavaScript)
  - Provides a mechanism to build an object tree of QML elements
  - Enables interaction between QML elements and Qt’s `QObject`-based C++ objects
- QML contains a set of *QML elements* and *items*
  - I.e. graphical and behavioral building blocks
  - These are combined into *QML documents* to build more complex components and QML applications
- Can be used to extend existing applications or to build completely new ones
  - QML itself is also fully extensible with C++!

Introduction

- As mentioned, QML is a declarative language for defining how:
  - An application looks like, and
  - How it behaves
- A QML UI is composed of a tree of elements with certain properties
- Prior knowledge of JavaScript (+ HTML and CSS) is an advantage when learning QML
  - Not strictly required, though

QML at a Glance – HelloWorld.qml

Forum Nokia

```
import QtQuick 1.0

Rectangle {
   width: 200; height: 200
   color: "lightblue"

   Text {
          id: helloText
          anchors.horizontalCenter: parent.horizontalCenter
          font.pixelSize: parent.height / 10
          font.bold: true
          text: "Meet QML!"
   }

   Image {
          id: helloImage
          anchors.centerIn: parent
          source: "icons/qt_logo.png"
   }

   MouseArea {
          anchors.fill: parent
          onClicked: {
             helloImage.visible = false;
             helloText.text = "Bye-bye picture!";
     }
   }
}
```

It's simply all about elements, properties and their values!

Import statement

- Gives you access to the built-in QML elements
  - `Rectangle, Item, Text, Image, …`
- Specifies which version of Qt Quick you are using
  - Notice the syntax change in Qt 4.7.1!

```
// In Qt 4.7.1 onwards:
import QtQuick 1.0

Rectangle {
width: 200; height: 200
// ...
```

```
// In Qt 4.7.0:
import Qt 4.7

Rectangle {
width: 200; height: 200
// ...
```

- Guarantees backwards compatibility
  - Only features of the specified version are loaded

QML Elements & Properties

Forum Nokia

```
import QtQuick 1.0

Rectangle {
    width: 200; height: 200
    color: "lightblue"

    Text {
        id: helloText
        anchors.horizontalCenter: parent.horizontalCenter
        font.pixelSize: parent.height / 10
        font.bold: true
        text: "Meet QML!"
    }

    Image {
        id: helloImage
        anchors.centerIn: parent
        source: "icons/qt_logo.png"
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
            helloImage.visible = false;
            helloText.text = "Bye-bye picture!";
        }
    }
}
```

QML elements form a parent/child hierarchy

Each .qml file has exactly one root element

All visual elements inherit the **Item** element defining certain common properties:

id, anchors
x, y, z
width, height
opacity, visible, rotation, scale
...

Standard QML Elements

- A number of ready-made QML UI elements are provided for convenience
  - `Item`, `Rectangle`, `Image`, `Text`, `MouseArea`, `WebView`, `ListView`, ...
  - Some of them can be used as containers (parent) for other elements (children)
  - Referred to as *QML items* in the documentation
    - All elements meant for constructing the UI inherit the `Item` element
- There are also elements that are used for describing the *behavior* of the application
  - `State`, `PropertyAnimation`, `Transition`, `Timer`, `DateTimeFormatter`, `Connection`, ...
  - Referred to as *QML declarative elements* in the documentation

QML Elements & Properties

Forum Nokia

```
import QtQuick 1.0

Rectangle {
   width: 200; height: 200
   color: "lightblue"

   Text {
     id: helloText
     anchors.horizontalCenter: parent.horizontalCenter
     font.pixelSize: parent.height / 10
     font.bold: true
     text: "Meet QML!"
   }

   Image {
     id: helloImage
     anchors.centerIn: parent
     source: "icons/qt_logo.png"
   }

   MouseArea {
     anchors.fill: parent
     onClicked: {
       helloImage.visible = false;
       helloText.text = "Bye-bye picture!";
     }
   }
}
```

If more than one property on a line, separate with a semi-colon

A special **id** property can be assigned to any element

Used for accessing its properties & methods from elsewhere

A special **parent** property always refers to the element's parent

Can be used instead of the id property to refer to the parent

QML Elements & Properties

Forum Nokia

- QML supports properties of many types
  - Int, bool, real, string, color, list, …
  - There are also so called grouped properties
  - Properties are type-safe

```
Text {
   x: 10.5                              // A real property
   text: "Lorem Ipsum"                  // A string property
   focus: true                          // A bool property

   states: [                                      // A list property
          State { name: "State_1" },
          State { name: "State_2" }
   ]

   font.pixelSize: 14                   // A grouped property
   font.bold: true

   width: "hello"                       // Illegal, real number expected!
}
```

QML Elements & Properties

Forum Nokia

- Property's value can be the result of a JavaScript expression
  - Or the return value of a JavaScript method call

- Properties can be bound to other properties
  - Property value automatically updated when the other one changes

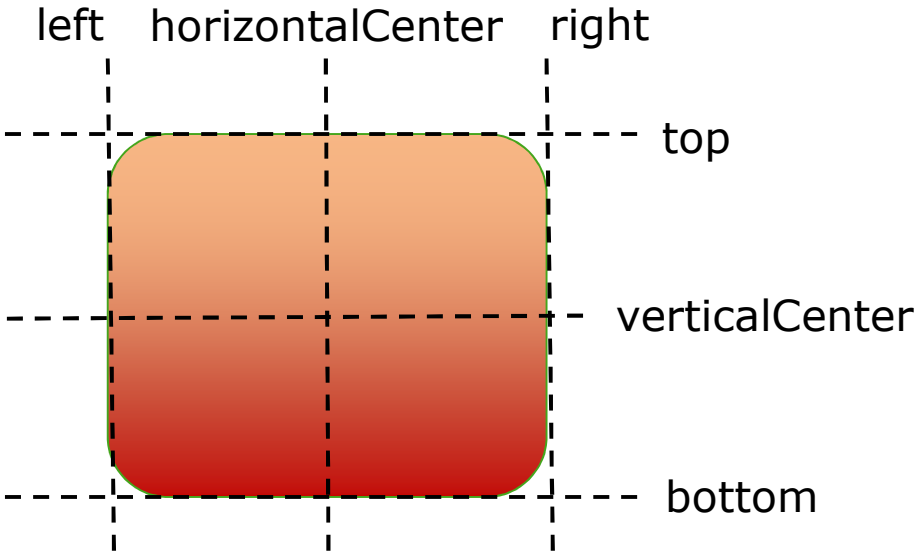
```
Rectangle {
   id: firstRectangle
   color: "blue"
   width: Math.min(30, parent.width)
   height: 2 * width
}

Rectangle {
   id: secondRectangle
   color: firstRectangle.width == 30 ? "red" : "green"
   width: firstRectangle.width
   height: firstRectangle.height / 2 + 20
}
```

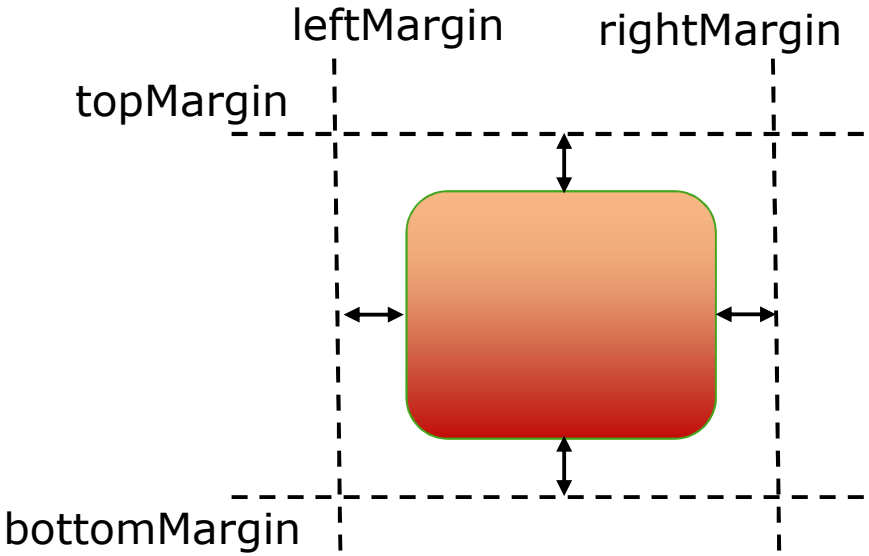
Note!
Using property bindings is highly recommended – this is the truly declarative way!

Anchor Layout 1/4

```
Rectangle {
        anchors.right: parent.right
        …
}
```

- Each QML item can be thought of as having 6 invisible anchor and 4 margin lines:

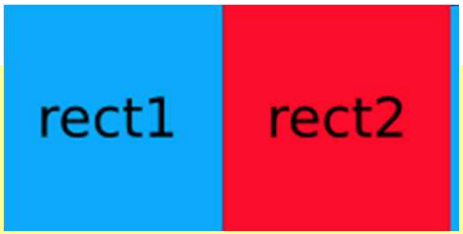


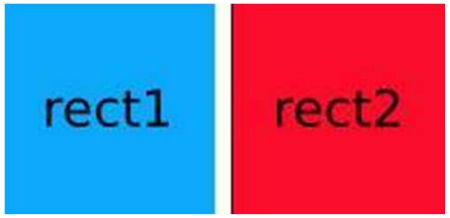


Anchor Layout 2/4

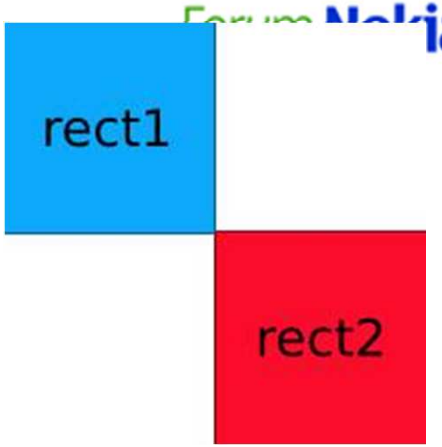
- The anchors are used for specifying relative positions of items
  - As well as offsets and margins

```
Rectangle { id: rect1; ... }
Rectangle { id: rect2; anchors.left: rect1.right; ... }

Rectangle { id: rect1; ... }
Rectangle { id: rect2; anchors.left: rect1.right; anchors.leftMargin: 5; ... }
```





Anchor Layout 3/4



- Multiple anchors can be specified
  - Can also be used to control the size of an item!

```
Rectangle { id: rect1; ... }
Rectangle { id: rect2; anchors.left: rect1.right; anchors.top: rect1.bottom;
    ... }

Rectangle { id: rect1; x: 0; ... }
Rectangle { id: rect2; anchors.left: rect1.right; anchors.right: Rect3.left;
    ... }
Rectangle { id: Rect3; x: 150; ... }
```

Anchor Layout 4/4

- For performance reasons you can only anchor an item to its siblings and direct parent

```
Item {
    id: Group1
    Rectangle { id: rect1; ... }
}

Item {
    id: Group2
    Rectangle { id: rect2; anchors.left: rect1.right; ... } // Invalid anchor!
}
```

QML Exercise 1 - Items

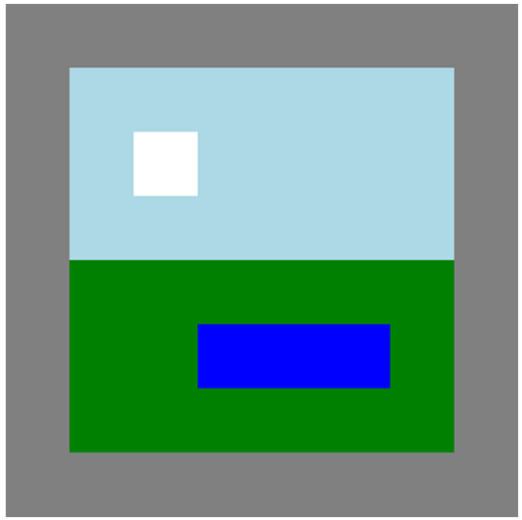
- The image on the right shows two items and two child items inside a 400 x 400 rectangle.

1. Recreate the scene using Rectangle items.
2. Can items overlap?

   Experiment by moving the light blue or green rectangles.

3. Can child items be displayed outside their parents?

   Experiment by giving one of the child items negative coordinates.



Qt Quick

More Layouts

Introduction

- Hard-coding the positions of UI elements is never a good idea
  - Difficult to provide UI scalability
  - Difficult to maintain
- QML provides a number of different kinds of layouts that should be used instead
  - Basic positioners
    - Grid, Row, Column
  - Anchor layout

Grid Layout

- Represented by the QML item `Grid`
  - Arranges child items in a grid formation so that they do not overlap each other
  - Provides for transition effects when items are added (shown), moved or removed (hidden)

```
Grid {
    columns: 3
    spacing: 2
    Rectangle { color: "red"; width: 50; height: 50 }
    Rectangle { color: "green"; width: 20; height: 50 }
    Rectangle { color: "blue"; width: 50; height: 20 }
    Rectangle { color: "cyan"; width: 50; height: 50 }
    Rectangle { color: "magenta"; width: 10; height: 10 }
}
```

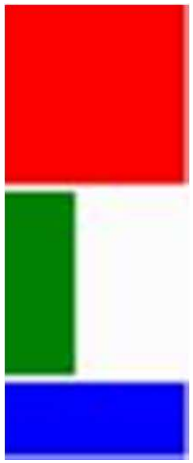
Row Layout

- Represented by the QML item Row
  - Positions child items in a row so that they do not overlap each other
  - Provides for transition effects when items are added (shown), moved or removed (hidden)

```
Row {
    spacing: 2
    Rectangle { color: "red"; width: 50; height: 50 }
    Rectangle { color: "green"; width: 20; height: 50 }
    Rectangle { color: "blue"; width: 50; height: 20 }
}
```

Column Layout

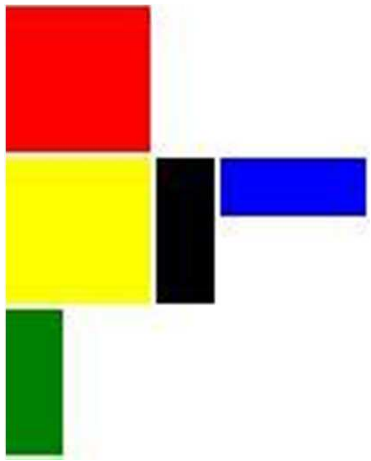
- Represented by the QML item `Column`
  - Positions child items vertically so that they do not overlap each other
  - Provides for transition effects when items are added (shown), moved or removed (hidden)



```
Column {
    spacing: 2
    Rectangle { color: "red"; width: 50; height: 50 }
    Rectangle { color: "green"; width: 20; height: 50 }
    Rectangle { color: "blue"; width: 50; height: 20 }
}
```

Combining Layouts

- The basic positioners `Grid`, `Row` and `Column` can be combined, if needed
- For example, a `Row` inside a `Column`:



```
Column {
    spacing: 2
    Rectangle { color: "red"; width: 50; height: 50 }
    Row {
          spacing: 2
           Rectangle { color: "yellow"; width: 50; height: 50 }
           Rectangle { color: "black"; width: 20; height: 50 }
           Rectangle { color: "blue"; width:50; height: 20 }
    }
    Rectangle { color: "green"; width: 20; height: 50 }
}
```

Qt Quick

User Interaction

User Interaction

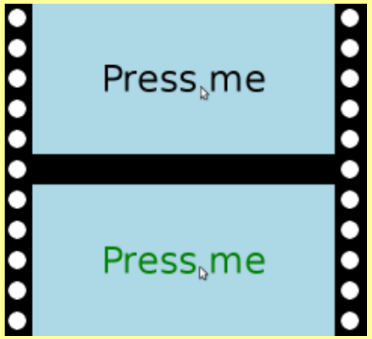
- In QML, the existing Items are just *drawing primitives*
- User interaction must be manually added
- MouseArea element for mouse interaction
- Keys attached property to raw key event handling
- KeyNavigation attached property for changing focus with keys

Mouse Areas

- Mouse areas define parts of the screen where cursor input occurs
- Placed and resized like ordinary items
  - Using anchors if necessary
- Two ways to monitor mouse input:
  - Handle signals
  - Dynamic property bindings

Clickable Mouse Area

```
import Qt 4.7

Rectangle {
    width: 400; height: 200; color: "lightblue"
    Text {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        text: "Press me"; font.pixelSize: 48
        MouseArea {
            anchors.fill: parent
            onPressed: {
                parent.color = "green“
            }
            onReleased: parent.color = "black"
        }
    }
}
```



Mouse Area Signals

```
Text {
    ...
    MouseArea {
        anchors.fill: parent
        onPressed: parent.color = "green"
        onReleased: parent.color = "black"
    }
}
```

- Define responses to signals with `onPressed` and `onReleased`
  - By default, only left clicks are handled
  - Set the `acceptedButtons` property to change this
- These change the color of the parent Text element
- Can do something similar with properties...

Different Mouse Events

- Signals are provided for handling mouse events
  - `onClicked, onDoubleClicked, onPressAndHold, onReleased, …`
  - A `MouseEvent` called *mouse* is delivered with the signal

```
Rectangle {
  width: 100; height: 100; color: "green"
  MouseArea {
        anchors.fill: parent
        // See Qt::MouseButtons for a list of available buttons
        acceptedButtons: Qt.LeftButton | Qt.RightButton
        onClicked: {
               if (mouse.button == Qt.RightButton)
                      parent.color = 'blue';
               else
                      parent.color = 'red';
        }
  }
}
```

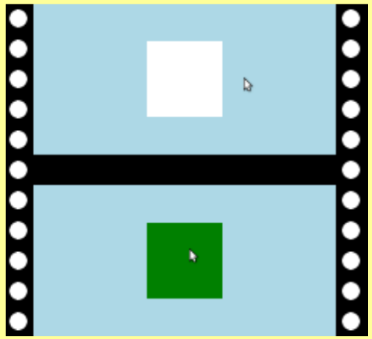
Dragging Elements

- MouseArea provides also a convenient way of making an item draggable with the `drag` property

```
Rectangle {
    id: opacitytest; width: 600; height: 200; color: "white"
    Image {
          id: pic; source: "qtlogo-64.png"
          anchors.verticalCenter: parent.verticalCenter
          opacity: (600.0-pic.x) / 600;
          MouseArea {
                  anchors.fill: parent
                  drag.target: pic
                  drag.axis: "XAxis"
                  drag.minimumX: 0
                  drag.maximumX: opacitytest.width-pic.width
          }
    }
}
```

Mouse Hover

```
import Qt 4.7
Rectangle {
    width: 400; height: 200; color: "lightblue"
    Rectangle {
        x: 150; y: 50; width: 100; height: 100
        color: mouse_area.containsMouse ? "green" : "white"
        MouseArea {
            id: mouse_area
            anchors.fill: parent
            hoverEnabled: true
        }
    }
}
```



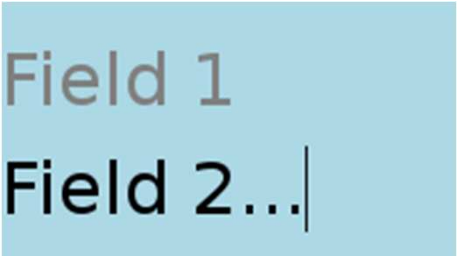
Keyboard Input

- Use cases for keyboard input:

1. Accepting text input
   - TextInput (single-line) and TextEdit (multi-line)
2. Navigation between elements
   - Changing the focused element
   - Directional (arrow keys), tab and backtab
3. Raw keyboard input
   - Reacting to arbitraty key presses, a game for instance

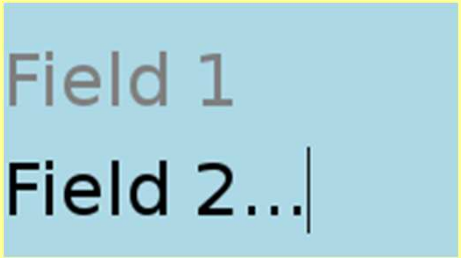
Assigning Focus

- UIs with just one TextInput
  - Focus assigned automatically
- More than one TextInput
  - Need to change focus by clicking
- What happens if a TextInput has no text?
  - No way to click on it
  - Unless it has a width or uses anchors
- Set the `focus` property to assign focus



Using TextInputs

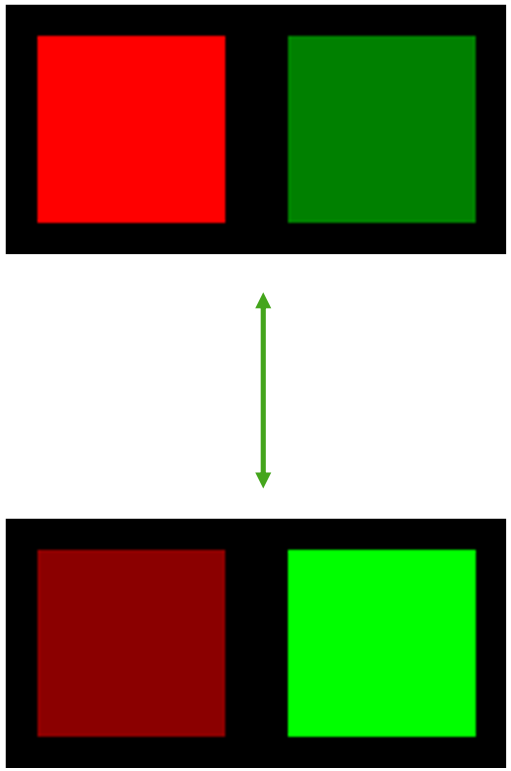
```
import Qt 4.7
Rectangle {
    width: 200; height: 112; color: "lightblue"
    TextInput {
        anchors.left: parent.left; y: 16
        anchors.right: parent.right
        text: "Field 1"; font.pixelSize: 32
        color: focus ? "black" : "gray"
        focus: true
    }
    TextInput {
        anchors.left: parent.left; y: 64
        anchors.right: parent.right
        text: "Field 2"; font.pixelSize: 32
        color: focus ? "black" : "gray"
    }
}
```



Key Navigation

```
Rectangle {
    width: 400; height: 200; color: "black"
    Rectangle {
        id: leftRect
        x: 25; y: 25; width: 150; height: 150
        color: focus ? "red" : "darkred"
        KeyNavigation.right: rightRect
        focus: true }
    Rectangle {
        id: rightRect
        x: 225; y: 25; width: 150; height: 150
        color: focus ? "#00ff00" : "green"
        KeyNavigation.left: leftRect }
}
```

Left rectangle has the initial focus



- Using cursor keys with non-text items
- Non-text items can have focus, too

Raw Keyboard Input 1/2

Forum Nokia

- All visual elements automatically support key event handling via the `Keys` attached property
- There are multiple signals associated with this property
  - The ”generic” ones: `onPressed, onReleased`
  - The ”specialized” ones: `onReturnPressed, onSelectPressed, onVolumeUpPressed, …`
  - These contain a `KeyEvent` parameter called *event*
- When handling the *generic* signals
  - You should explicitly state if the event was handled

    ```
    event.accepted = true;
    ```

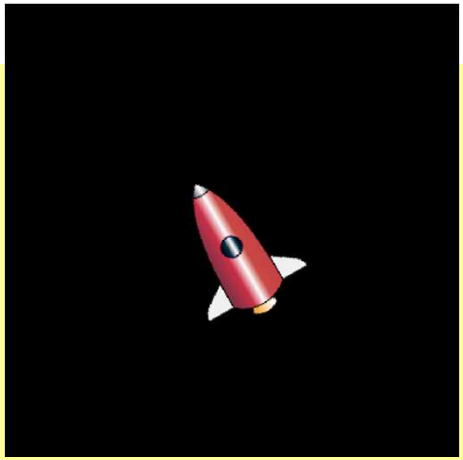
  - Otherwise the event is propagated to other objects
- *Specialized* handlers accept the event by default

Raw Keyboard Input 2/2

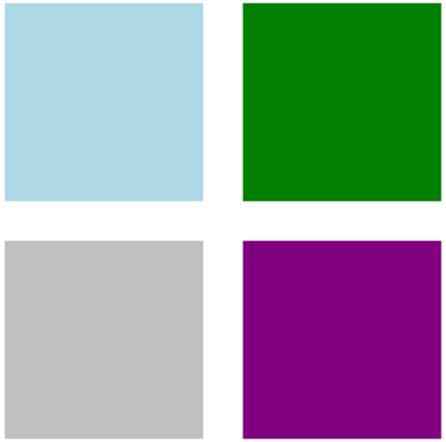
```
Item {   // Handle a key event with a generic handler
  focus: true
  Keys.onPressed: {
        if (event.key == Qt.Key_Left) {   // See Qt::Key for codes
              console.log("move left");
              event.accepted = true;   // Must accept explicitly
        }
  }
}

Item { // Handle a key event with a specialized handler
  focus: true
  Keys.onLeftPressed:                // Accepts the event by default
        console.log("move left")
}
```

Another Example

```
import Qt 4.7
Rectangle {
    width: 400; height: 400; color: "black"
    Image {
            id: rocket
            x: 150; y: 150
            source: "../images/rocket.svg"
            transformOrigin: Item.Center
    }
    Keys.onLeftPressed:
            rocket.rotation = (rocket.rotation - 10) % 360
    Keys.onRightPressed:
            rocket.rotation = (rocket.rotation + 10) % 360
    focus: true
}
```



QML Exercise 2



- Create a user interface using layouts similar to the one shown above with these features:
  - Items that change color when they have the focus
  - Clicking an item gives it the focus
  - The current focus can be moved using the cursor keys

Qt Quick

States, Transitions and Animations

Purpose

- Can define user interface behavior using states and transitions:
  - Provides a way to formally specify a user interface
  - Useful way to organize application logic
  - Helps to determine if all functionality is covered
  - Can extend transitions with animations and visual effects

States

- States manage named items
- Represented by the `State` element
- Each item can define a set of states
  - With the `states` property
  - Current state is set with the `state` property
- Properties are set when a state is entered
- Can also
  - Modify anchors
  - Change the parents of items
  - Run scripts

States Example 1/3

```
import Qt 4.7
Rectangle {
    width: 150; height: 250
    Rectangle {
        id: stop_light
        x: 25; y: 15; width: 100; height: 100
    }
    Rectangle {
        id: go_light
        x: 25; y: 135; width: 100; height: 100
    }
    ...
```

- Prepare each item with an id
- Set up properties not modified by states

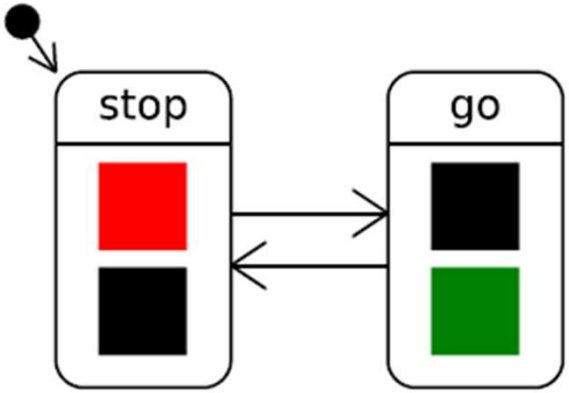
States Example 2/3

```
states: [
    State {
        name: "stop"
        PropertyChanges { target: stop_light; color: "red" }
        PropertyChanges { target: go_light; color: "black" }
    },
    State {
        name: "go"
        PropertyChanges { target: stop_light; color: "black" }
        PropertyChanges { target: go_light; color: "green" }
    }
]
...
```

- Define states with names: "stop" and "go"
- Set up properties for each state with PropertyChanges
- These define differences from the default values for each item

States Example 3/3

```
    …
    state: "stop“ // Define initial state

    MouseArea {
        anchors.fill: parent
        onClicked: parent.state == "stop" ?
        parent.state = "go" : parent.state = "stop"
    }
}
```



- Use a MouseArea to switch between states
  - Reacts to a click on the user interface
    - Toggles the parent’s state property
    - Between "stop" and "go" states

Changing Properties

- States change properties with the PropertyChanges element:

```
State {
    name: "stop"
    PropertyChanges { target: stop_light; color: "red" }
    PropertyChanges { target: go_light; color: "black" }
}
```

- Acts on the named target element
- Applies the other property definitions to the target element
  - One PropertyChanges element can redefine multiple properties
- Property definitions are evaluated when the state is entered

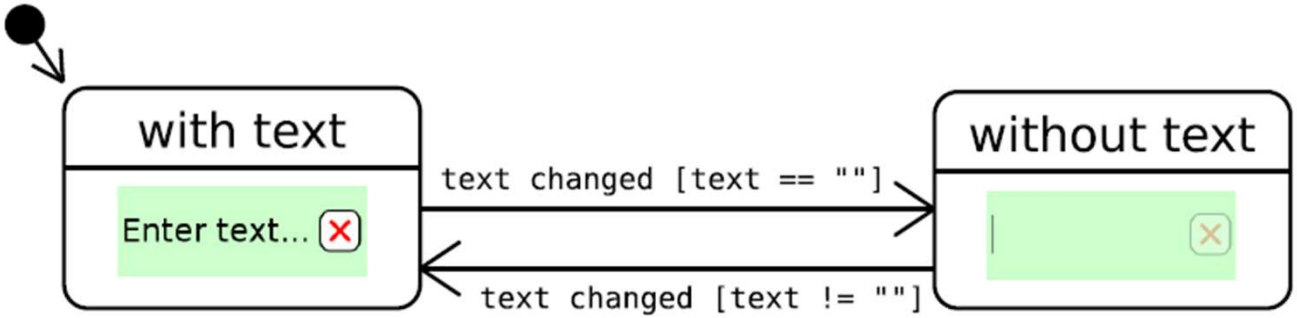
Default Properties

- Each object can specify one default property
  - In this case the property name tag can be omitted when the property is assigned a value
  - Consider the `changes` property, which is the default property of the `State` object

```
State {
   changes: [
         PropertyChanges {},
         PropertyChanges {}
   ]
}

// ... can be simplified to:
State {
   PropertyChanges {}
   PropertyChanges {}
}
```

State Conditions

- Another way to use states:
- Let the State decide when to be active
  - Using conditions to determine if a state is active
- Define the `when` property
  - Using an expression that evaluates to true or false
- Only one state in a states list should be active
  - Ensure `when` is true for only one state



State Conditions Example

```
Rectangle {
    width: 250; height: 50; color: "#ccffcc"
    TextInput {
        id: text_field
        text: "Enter text..."
        ...
    }
    Image {
        id: clear_button
        source: "../images/clear.svg"
        ...
        MouseArea {
            anchors.fill: parent
            onClicked: text_field.text = ""
        }
    }
```

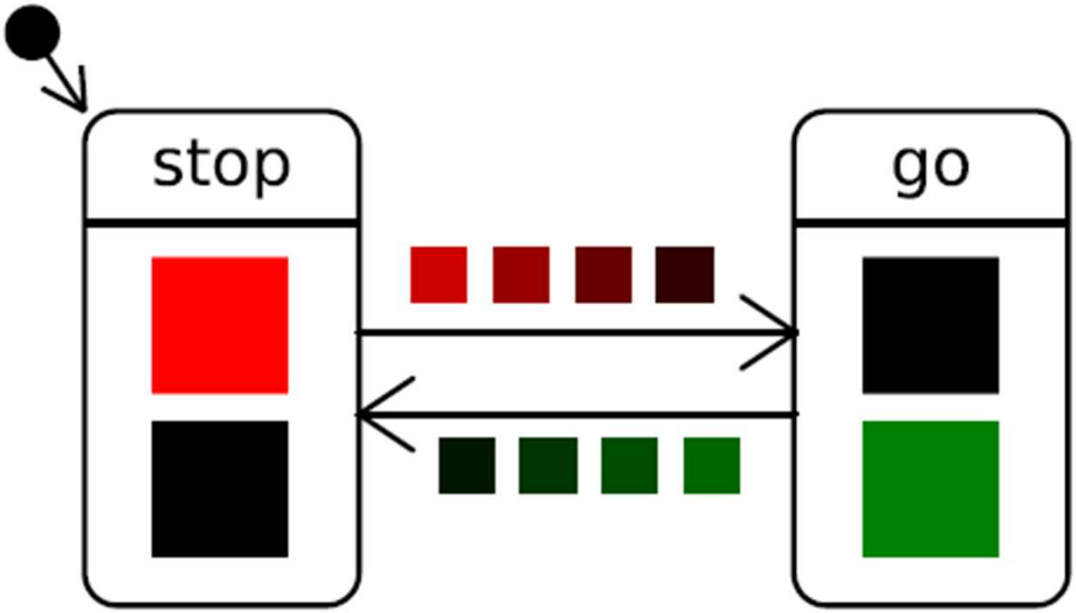
```
    states: [
        State {
            name: "with text"
            when: text_field.text != ""
            PropertyChanges { target: clear_button;
                              opacity: 1.0 }
            },
        State {
            name: "without text"
            when: text_field.text == ""
            PropertyChanges { target: clear_button;
                              opacity: 0.25 }
            PropertyChanges { target: text_field;
                              focus: true }
        }
    ]
}
```

Transitions

- The `Transition` element provides a way of adding animations to state changes
  - If fact, a transition can only be triggered by a state change
  - As usual, transition animations can be run in sequence and/or in parallel
- By specifying the `to` and `from` properties you can explicitly specify the states the transition is associated with
  - By default these have the value `"*"`, i.e. any state
- A transition can be set to be `reversible` (default `false`)
  - When conditions triggering the transition are reversed, the transition is automatically run backwards
  - For example: state change 1 -> 2 and then 2 -> 1

Transitions Example

- Let’s add transitions to a previous example...



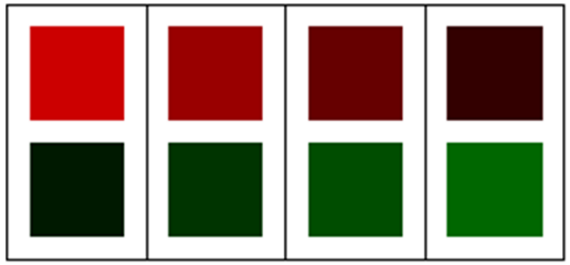
Transitions Example

```
transitions: [
        Transition {
            from: "stop"; to: "go"
            PropertyAnimation {
               target: stop_light
               properties: "color"; duration: 1000
            }
        },
        Transition {
            from: "go"; to: "stop"
            PropertyAnimation {
               target: go_light
               properties: "color"; duration: 1000
            }
        }
]
```

- The transitions property defines a list of transitions
- Transitions between "stop" and "go" states

Wildcard Transitions

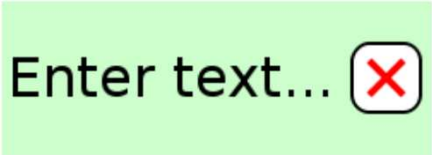
```
transitions: [
        Transition {
            from: "*"; to: "*"
            PropertyAnimation {
               target: stop_light
               properties: "color"; duration: 1000
            }
            PropertyAnimation {
               target: go_light
               properties: "color"; duration: 1000
            }
} ]
```



- Use "*" to represent any state (actually, default value is “*”)
- Now the same transition is used whenever the state changes
- Both lights fade at the same time

Reversible Transitions

```
transitions: [
        Transition {
            from: "with text"; to: "without text"
            reversible: true
            PropertyAnimation {
               target: clear_button
               properties: "opacity"; duration: 1000
            }
        }
]
```



- Useful when two transitions operate on the same properties
- Transition applies from "with text" to "without text"
  - And back again from "without text" to "with text"
- No need to define two separate transitions

Animations 1/2

Forum Nokia

- It is possible to animate the properties of objects
  - Types: `real, int, color, rect, point, size`
- Three different forms of animations are available
  - Basic property animation, transitions, property behaviors
  - See the next slide
- Animations can be grouped, i.e. run in parallel or in sequence
  - `SequentialAnimation, ParallelAnimation, PauseAnimation`
- A set of pre-defined easing curves is available
  - `OutQuad, InElastic, OutBounce, …`
  - For more information, see `PropertyAnimation` documentation

Animations 2/2

Forum Nokia

- For property animations, use either
  - `PropertyAnimation`, `NumberAnimation`, or `ColorAnimation`
  - All of these inherit the base element `Animation`
- For property behaviors, use the element `Behavior`
- For transitions, use the `Transition` element
  - Covered already

```
PropertyAnimation {                 // A basic property animation
    target: theObject               // The object (id) to be animated
    property: "size“                // The property to be animated
    to: "20x20“; duration: 200      // End value & duration
}
```

Animation Example 1/2

```
Rectangle {    // Example of a drop-and-bounce effect on an image
  id: rect
  width: 120; height: 200;
  Image {
        id: img
        source: "qt-logo.png"
        x: 60-img.width/2
        y: 0

        SequentialAnimation on y {
            running: true; loops: Animation.Infinite
            NumberAnimation {
                to: 200-img.height; easing.type: "OutBounce"; duration: 2000
            }
            PauseAnimation { duration: 1000 }
            NumberAnimation {
                to: 0; easing.type: "OutQuad"; duration: 1000
           }
        }
  }
}
```

Animation Example 2/2

```
PropertyAnimation {          // Animation as a separate element,
  id: animation       // referred to by its id
  target: image
  property: "scale"
  from: 1; to: .5
}

Image {
  id: image
  source: "image.png"
  MouseArea {         // The animation is started upon mouse press
        anchors.fill: parent
        onClicked: animation.start()
  }
}
```

Property Behavior

- Specifies a default animation to run whenever the property's value changes
  - Regardless of what caused the change!
- The example below animates the `x` position and `width` of `redRect` whenever they change

```
Rectangle {
    id: redRect
    color: "red"
    width: 100; height: 100
    x: Behavior {
          NumberAnimation { duration: 300; easing.type: "InOutQuad" }
    }
    Behavior on width {
          NumberAnimation { duration: 1000 }
    }
}
```

Using States and Transitions

Forum Nokia

- Avoid defining complex state machines
  - Not just one state machine to manage the entire UI
  - Usually defined individually for each component
  - Link together components with internal states
- Setting state with script code
  - Easy to do, but might be difficult to manage
  - Cannot use reversible transitions
- Setting state with state conditions
  - More declarative style
  - Can be difficult to specify conditions

Summary - States

- State items manage properties of other items:
- Items define states using the `states` property
  - Must define a unique `name` for each state
- Useful to assign `id` properties to items
  - Use `PropertyChanges` to modify items
- The `state` property contains the current state
  - Set this using JavaScript code, or
  - Define a `when` condition for each state

Summary - Transitions

- Transition items describe how items change between states
- Items define transitions using the `transitions` property
- Transitions refer to the states they are between
  - Using the `from` and `to` properties
  - Using a wildcard value, `"*"`, to mean any state
- Transitions can be reversible
  - Used when the `from` and `to` properties are reversed

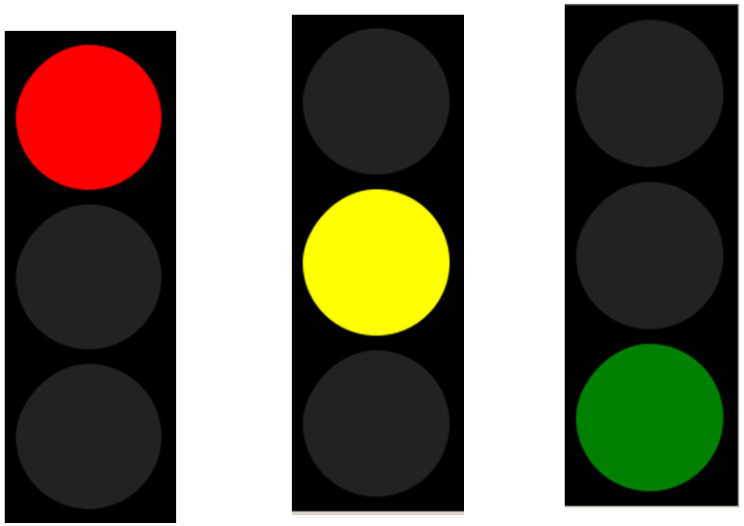
Timer

- Timers are handled using the `Timer` item
  - Provides only one signal: `onTriggered`
  - Can be either a single-shot or a repetitive timer

```
Timer {
    interval: 500;
    running: true;
    repeat: true
    onTriggered: time.text = Date().toString()
}

Text {
    id: time
}
```

QML Exercise 3 – Traffic Lights

- Using the following items, construct a simulation of a traffic lights:
  - States
  - Transitions
  - Timer



  - Feel free to make your application as smart as possible (different times for different states, yellow behaves differently based on the direction, etc.)

Qt Quick

QML Components

QML Document 1/2

- Simply a block of QML code containing QML elements
  - A `.qml` file, or constructed from text data
  - Always encoded in UTF-8
  - Always begins with at least one `import` statement
    - Nothing is imported by default
    - Does not ”include” code, rather just tells the interpreter where to find the definitions of elements at run-time
- Defines a single, top-level *QML component*
- Self-contained
  - No preprocessor or similar is run to modify the code before execution
  - Interpreted at run-time!

QML Document 2/2

- Our HelloWorld is a QML document stored e.g. in the **`HelloWorld.qml`** file

```
import Qt 4.7  // Import existing QML types to be used in this
               // application, such as Rectangle and Text
Rectangle {
   id: page
   width: 500; height: 200
   color: "lightgray"
   Text {
        id: helloText
        text: "Hello world!"
        font.pointSize: 24; font.bold: true
        y: 30; anchors.horizontalCenter: page.horizontalCenter
   }
}
```

QML Component

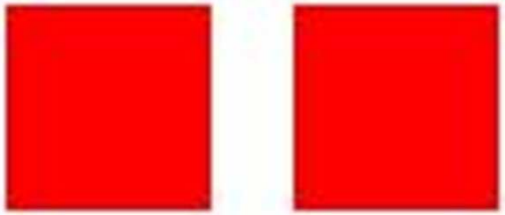
- As mentioned, a QML document defines a single, top-level QML component
  - A template (cf. a class in C++/Java) out of which objects are created – i.e. the component is *instantiated* at run-time
  - For example, a ”button” component instantiated multiple times with different button text values
- Components are among the basic building blocks in QML
  - Easy to create your own re-usable components
  - Component file name starts with a capital letter (”MyButton.qml”)
- A component can contain *inline components*
  - Declared with the keyword `Component`
  - Share the characteristics and import list of the parent
  - Useful e.g. when re-using a component within a single QML file (component logically belongs only to that file)

Top-Level QML Component

```
// Definition in MyButton.qml
// (Notice the capital ”M” in the file name above)
import Qt 4.7
Rectangle {
  property alias text: textElement.text
  width: 100; height: 30
  source: "images/toolbutton.sci"
  Text {
        id: textElement
        anchors.centerIn: parent
        font.pointSize: 20
        style: Text.Raised; color: "white"
  }
}
// Usage e.g. in main.qml
// (Just an ”entry point” file, thus lower-case ”m”)
import Qt 4.7
Rectangle {
     MyButton {
       anchors.horizontalCenter: parent.horizontalCenter
        text: "Orange"
     }
…}
```

Inline QML Component

```
// In MyComponent.qml
import Qt 4.7

Item {
  Component {  // The inline component
        id: redSquare
        Rectangle {
               color: "red"
               width: 50
               height: 50
        }
  }
  Loader { sourceComponent: redSquare }
  Loader { sourceComponent: redSquare; x: 70 }
}
```



Loader Element

- You'll need Loader element for inline components
- Loader can also be used for implementing *lazy loading* or replacing parts of UI dynamically
- `source` property of loader can be an external qml file
  - Loaded run-time
  - Changing source causes new file to be loaded
- Loader is a graphical item
  - Needs place and size, after loading, will resize
- Unloading possible and recommendable
  - Frees resources!

QML Modules 1/2

Forum Nokia

- Multiple QML components can be grouped into *QML modules*
  - The easiest way is to just create a subdirectory containing all the components for the module
  - These modules can then be imported in QML documents:

    `import "path_to_mymodule"`
  - The path to the module is relative to the file importing it
- You can also use *named imports*
  - To allow identically named modules, or just for code readability

```
import Qt 4.7 as TheQtLibrary   // Into a namespace called TheQtLibrary
TheQtLibrary.Rectangle { ... }

// Multiple imports into the same namespace are also allowed:
import Qt 4.7 as Nokia
import Ovi 1.0 as Nokia
```

QML Modules 2/2

Forum Nokia

- QML component files can also be installed somewhere outside your project
  - In this case an unquoted URI is used:

    ```
    import com.nokia.SomeStuff 1.0
    ```

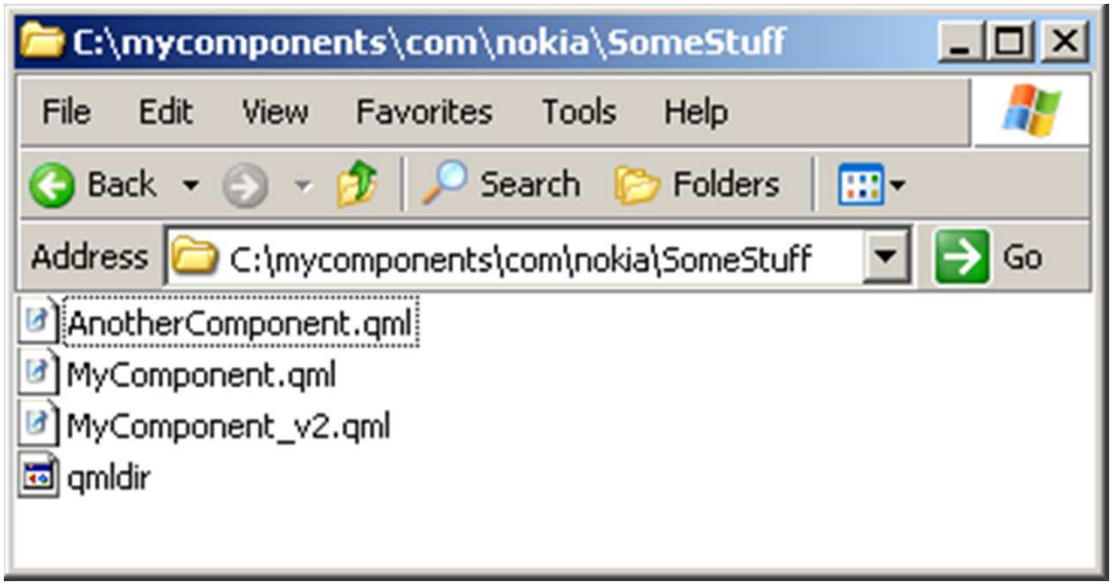
  - This would access files in folder **`com/nokia/SomeStuff/`** located somewhere in your system
    - E.g. under **`c:/mycomponents/`**
  - The path leading to this URI can be set either
    - In C++ by using `QmlEngine::addImportPath()`, or
    - By specifying the –L command line option to qml.exe
- Another possibility:

  ```
  import "http://myserver.com/.../..." 1.0
  ```

- Either way, a file called **`qmldir`** must be present in the directory describing the contents:

  ```
  # <Comment>
  <TypeName> <InitialVersion> <File>
  ```

QML Modules – Example




```
// qmldir contents:
# This is a comment
MyComponent 2.0 MyComponent_v2.qml
MyComponent 1.0 MyComponent.qml
AnotherComponent 1.0 AnotherComponent.qml
```

Network Transparency 1/2

- Simply means that all references from a QML document to other content are handled as URLs
  - Works for both local and remote content as well as relative and absolute URLs

```
// Test1.qml containing a reference to an absolute URL
Image { source: "http://www.example.com/images/logo.png" }

// Test2.qml with a relative URL
Image { source: "images/logo.png" }
```

Network Transparency 2/2

Forum Nokia

- Relative URLs are resolved automatically into absolute ones
  - Absolute URLs always stay as they are
- Example 1: **`Test2.qml`** itself is loaded from **`http://www.example.com/mystuff/Test2.qml`**
  - URL to image is automatically resolved into **`http://www.example.com/mystuff/images/logo.png`**
- Example 2: **`Test2.qml`** itself is loaded from **`C:/temp/mystuff/Test2.qml`**
  - URL to image is automatically resolved into **`C:/temp/mystuff/images/logo.png`**

Progressive Loading

- QML objects that reference a network resource typically provide information on the loading status
  - Needed because networking is inherently asynchronous
- For example, the `Image` element has special properties related to this:
  - `status (Null, Ready, Loading, Error)`
  - `progress (0.0 – 1.0)`
  - `width` and `height` also change as the image is loaded
- Applications can bind to these properties to e.g. show a progress bar when applicable
- For local image files the `status` is `Ready` to start with
  - In future versions this might change
  - If you wish to remain network transparent, do not rely on this!

Qt Quick

Extending QML Components

Extending Types with QML

Forum Nokia

- Many of the QML core types/elements are actually implemented in C++
- However, extending these types purely with QML is also possible
  - We will talk about extending QML types with C++ later
- With QML, a developer can
  - Add new properties,
  - Add new signals,
  - Add new methods, and
  - Define totally new QML Components
    - We covered this already

Adding New Properties 1/4

- Each new property has to have a *type*
  - QML has a set of predefined types to use
  - Each QML type maps to a C++ type

```
// Syntax of adding a new property to an element
[default] property <type> <name>[: defaultValue]

// Example:
Rectangle {
    property color innerColor: "black"
    color: "red"; width: 100; height: 100
    Rectangle {
        anchors.centerIn: parent
        width: parent.width - 10
        height: parent.height - 10
        color: innerColor
    }
}
```

QML Type	C++ Type
int	int
bool	bool
double	double
real	double
string	QString
url	QUrl
color	QColor
date	QDate
var	QVariant
variant	QVariant

Adding New Properties 2/4

Forum Nokia

- The new property can also be an alias of an existing property (a.k.a. *the aliased property*)
  - A new property (and the storage space for it) is not actually allocated
  - The type is determined by the aliased property

```
// Syntax of creating a property alias
[default] property alias <name>: <alias reference>

// The previous example using a property alias:
Rectangle {
    property alias innerColor: innerRect.color
    color: "red"; width: 100; height: 100
    Rectangle {
          id: innerRect; anchors.centerIn: parent
          width: parent.width – 10; height: parent.height - 10
          color: "black"
    }
}
```

Adding New Properties 3/4

- Property aliases are most useful when defining new *components*
- However, there are a few limitations with aliases
  - Can only be activated once the component specifying them is completed
    - I.e. you cannnot use the alias in the component itself!
    - An alias cannot refer to another alias in the same component

```
// Does NOT work:
property alias innerColor: innerRect.color
innerColor: "black"

// ...and neither does this:
id: root
property alias innerColor: innerRect.color
property alias innerColor2: root.innerColor
```

Adding New Properties 4/4

Forum Nokia

- Despite the limitations, the alias mechanism does provide quite a lot of flexibility as well
  - You can redefine the behaviour of existing property names, and
  - Still within the component use the property ”as usual”
- In the example below:
  - The outer rectangle is always red and the user can only modify the color of the inner rectangle, by
  - Using the familiar property called `color` instead of `innerColor`!

```
Rectangle {
    property alias color: innerRect.color
    color: "red"; width: 100; height: 100
    Rectangle { id: innerRect; ...; color: "black" }
}
```

Adding New Signals (1/2)

Forum Nokia

- We saw earlier various signals used in existing QML elements
  - `MouseArea.onClicked, Timer.onTriggered, …`
- Custom signals can be defined as well
  - Can be used within QML
  - Also appear as regular Qt signals in the C++ side!
  - Signals can have arguments (of the QML types shown earlier)

```
Item {
    signal hovered() // A signal without arguments
    signal clicked   // The same as above, empty argument list can be omitted
    signal performAction(string action, var actionArgument)
}
```

Adding New Signals (2/2)

Forum Nokia

```
// MyItem.qml
Rectangle {
    color: "red"; width: 100; height: 100
    signal superClicked
    MouseArea: {
          anchors.fill: parent
          onClicked(): { superClicked() }
     }
```

Defining a signal: *"Elements of this type can emit signal superClicked"*—part of the type interface

Emitting a signal, declaring the situation where this signal is emitted: *"When this rectangle is clicked, it will emit signal superClicked()"*—just internal implementational details…

```
// main.qml
Rectangle {
    …
    MyItem {
          onSuperClicked: {
                  color: "yellow"
          }
    }
    …
}
```

Using the type and creating a signal handler ("slot") for it: *"What do we do when THIS object emits superClicked?"*

Adding New Methods

- New methods can be added to existing types
  - Normally implemented in JavaScript
  - Usable from QML directly and from C++ as slot functions
  - Can have un-typed parameters
    - Because JavaScript itself is un-typed
    - In C++ the parameter type is `QVariant`

```
// Define a method
Item {
   id: myItem
   function say(text) {
        console.log("You said " + text);
   }
}

// Use the method
myItem.say(”HelloWorld!”);
```

QML Exercise 4 - Components

Forum Nokia

- Create your own UI component:
  - Push button
  - Line edit
  - Slider
  - Something else?
- What elements? What signals for others to use this?
- Try using these somewhere!

Qt Quick for Mobile

Data Models and Views

Forum Nokia

NOKIA

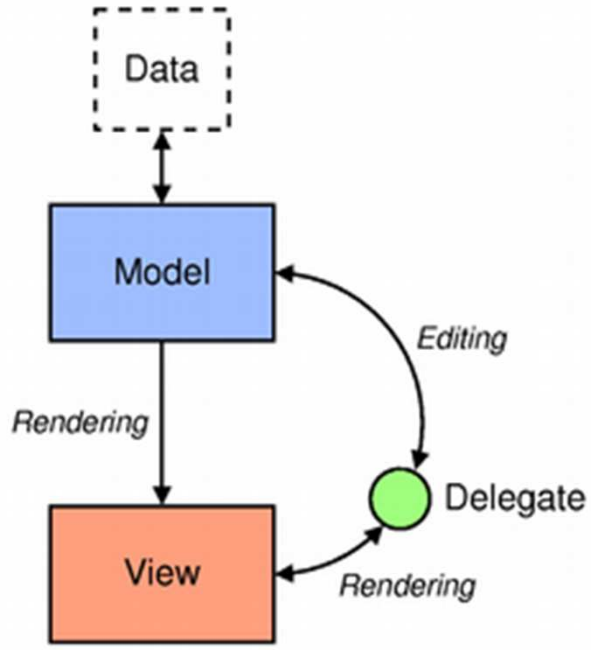
Data Models and Views

Forum Nokia

- QML uses a similar Model-View pattern as Qt
- *Model* classes provide data
  - Models can be either in QML (simple cases) or C++ (more complex cases)
  - **QML:** `ListModel, XmlListModel, VisualItemModel`
  - **C++:** `QAbstractItemModel, QStringList, QList<QObject*>`
- *View* classes are used for displaying the data in a model
  - `ListView, GridView, PathView, Repeater` (all QML)
  - All automatically support ”scrolling by flicking”
- *Delegates* are used for creating instances of items in the model for the view
- *Highlight* components are used highlighting list items in the view

For Reference: Model-View in Qt

- Model provides interface to data for other components
  - QAbstractItemModel
- View obtains model indices
  - Indices are references to the data
- Delegate usually handles custom rendering for the view
  - Delegate communicates directly with model when user edits



What We Need and What Are They Again?

- Model
  - Your data
- Delegate
  - A Component that describes a prototype item of each piece of data in model
- View
  - A graphical element that automatically shows the contents of a model using the delegate

Example – A Simple List 1/3

```
// Define the data in MyModel.qml - data is static in this simple case
import Qt 4.7

ListModel {
  id: contactModel
        ListElement {
               name: "Bill Smith"
               number: "555 3264"
  }
  ListElement {
        name: "John Brown"
        number: "555 8426"
  }
  ListElement {
        name: "Sam Wise"
        number: "555 0473“
  }
}
```

Example – A Simple List 2/3

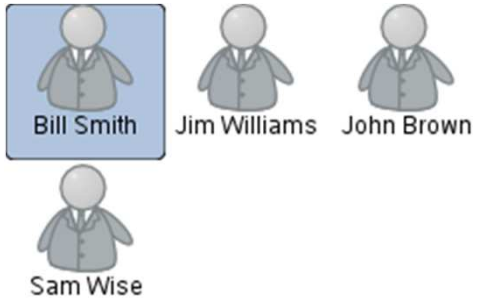
```
// Create a view to use the model e.g. in myList.qml
import Qt 4.7

Rectangle {
  width: 180; height: 200; color: "green"

  // Define a delegate component.  A delegate will be
  // instantiated for each visible item in the list.
  Component {
        id: delegate
        Item {
              id: wrapper
              width: 180; height: 40
              Column {
                x: 5; y: 5
                Text { text: '<b>Name:</b> ' + name }
                Text { text: '<b>Number:</b> ' + number }
              }
        }
  }  // Rectangle continues on the next slide...
```

Example – A Simple List 3/3

```
// ...Rectangle continued...
  // Define a highlight component. Just one of these will be
  // instantiated by each ListView and placed behind the current item.
  Component {
        id: highlight
        Rectangle {
              color: "lightsteelblue"
              radius: 5
        }
  }

  // The actual list
  ListView {
        width: parent.width; height: parent.height
        model: MyModel{}              // Refers to MyModel.qml
        delegate: delegate    // Refers to the delegate component
        highlight: highlight  // Refers to the highlight component
        focus: true
  }
}  // End of Rectangle element started on previous slide
```

GridView

- `GridView`
  - Shows items in a grid formation
  - Usage is otherwise identical to `ListView`




```
GridView {
    width: parent.width; height: parent.height
    model: MyModel
    delegate: delegate
    highlight: highlight
    cellWidth: 80; cellHeight: 80
    focus: true
}
```

PathView 1/3

- `PathView`
  - Shows items in a formation specified by a separate `Path` object
  - Several pre-defined elements exist, of which the `Path` is constructed
    - `PathLine, PathQuad, PathCubic`
  - Distribution of items along different parts of the path are controlled with the element `PathPercent`
  - Appearence of items can be controlled with `PathAttribute`

PathView 2/3

```
PathView {      // With equal distribution of dots
  anchors.fill: parent; model: MyModel{ }; delegate: delegate
  path: Path {
        startX: 20; startY: 0
        PathQuad { x: 50; y: 80; controlX: 0; controlY: 80 }
        PathLine { x: 150; y: 80 }
        PathQuad { x: 180; y: 0; controlX: 200; controlY: 80 }
  }
}

PathView {      // With 50% of the dots in the bottom part
  anchors.fill: parent; model: MyModel{ }; delegate: delegate
  path: Path {
        startX: 20; startY: 0
        PathQuad { x: 50; y: 80; controlX: 0; controlY: 80 }
        PathPercent { value: 0.25 }
        PathLine { x: 150; y: 80 }
        PathPercent { value: 0.75 }
        PathQuad { x: 180; y: 0; controlX: 200; controlY: 80 }
        PathPercent { value: 1 }
  }
}
```

PathView 3/3

```
Component {
  id: delegate
  Item {
        id: wrapper; width: 80; height: 80
        scale: PathView.scale
        opacity: PathView.opacity
        Column {
            Image { ... }
            Text { ... }
        }
  }
}

PathView {
  anchors.fill: parent; model: MyModel{ }; delegate: delegate
  path: Path {
        startX: 120; startY: 100
        PathAttribute { name: "scale"; value: 1.0 }
        PathAttribute { name: "opacity"; value: 1.0 }
        PathQuad { x: 120; y: 25; controlX: 260; controlY: 75 }
        PathAttribute { name: "scale"; value: 0.3 }
        PathAttribute { name: "opacity"; value: 0.5 }
        PathQuad { x: 120; y: 100; controlX: -20; controlY: 75 }
  }
}
```

Repeater 1/2

- An item for creating a large number of similar items
- Uses a model just like the view elements shown earlier
  - The model can be an object list, a string list, **a number**, or a Qt/C++ model
  - The current model index is exposed as an `index` property

```
Column {
    Repeater {
        model: 10 // The model is just a number here!
        Text { text: "I'm item " + index }
    }
}
```

I'm item 0
I'm item 1
I'm item 2
I'm item 3
I'm item 4
I'm item 5
I'm item 6
I'm item 7
I'm item 8
I'm item 9

Repeater 2/2

- Items created by the `Repeater` are inserted (in order) as children of the `Repeater`'s parent
  - Enables using the `Repeater` inside layouts
  - For example, a `Repeater` inside a `Row` layout:

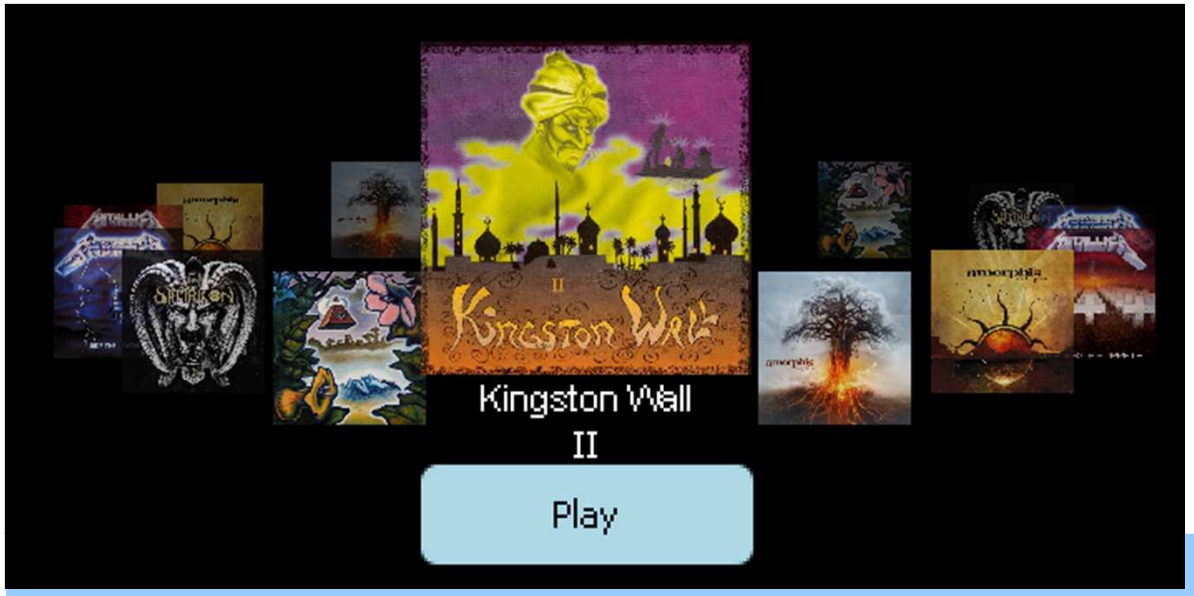
```
Row {
    Rectangle { width: 10; height: 20; color: "red" }
    Repeater {
          model: 10
          Rectangle { width: 20; height: 20; radius: 10; color: "green" }
    }
    Rectangle { width: 10; height: 20; color: "blue" }
}
```

Flickable

- An item that places its children on a surface that can be dragged and ”flicked”
  - No need to create a `MouseArea` or manually handle mouse events in any other way
- The flickable surface is easily configurable via its properties
  - `flickDirection, flickDeceleration, horizontalVelocity, verticalVelocity, overShoot, …`
- Certain QML elements are flickable by default
  - The `ListView` element, for example

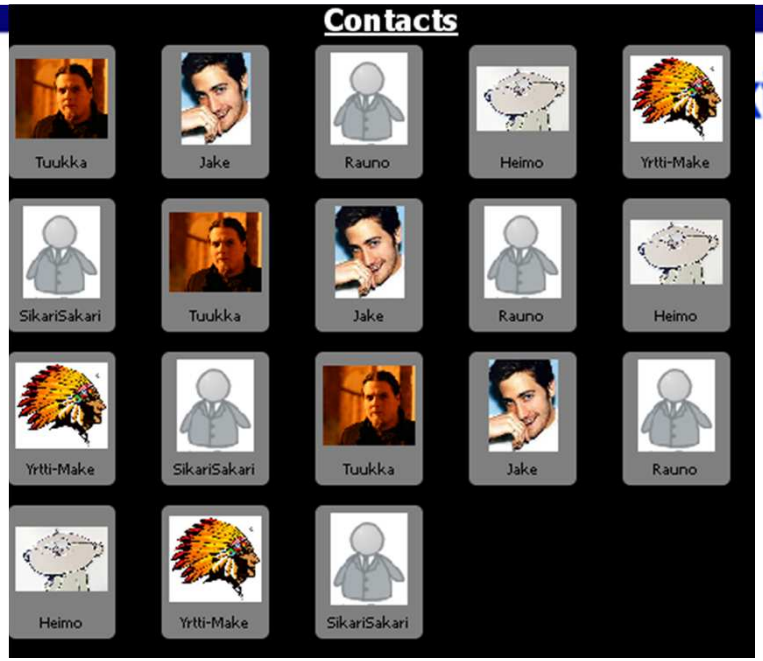
```
Flickable {
   width: 200; height: 200
   contentWidth: image.width; contentHeight: image.height
   Image { id: image; source: "bigimage.png" }
}
```

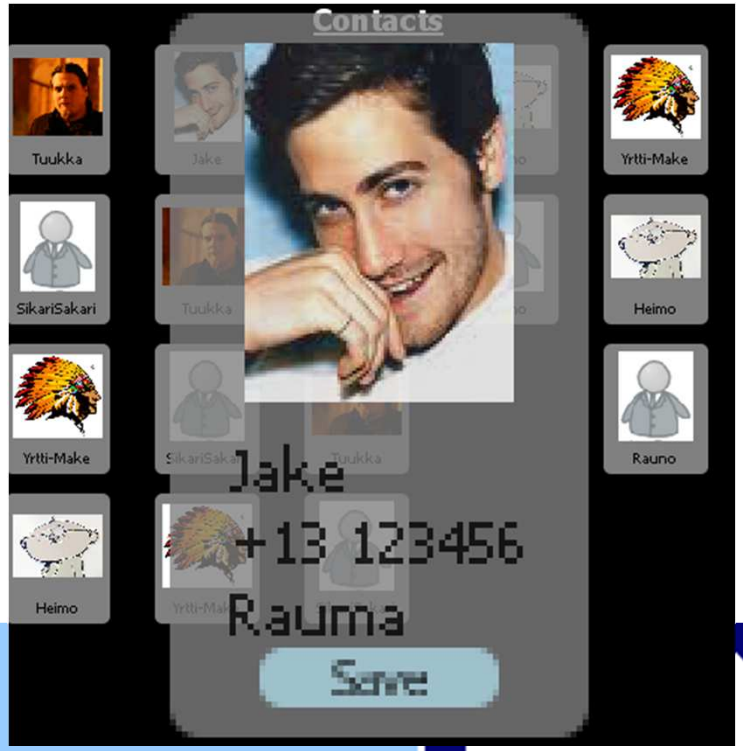
Example, CD Cover View



Mega-Exercise

- Contacts
- List of Contacts (in a GridView, e.g.)
- Separate Model (as a separate Component)
- Click contact -> Open/Close details
- Start with small (text), extend bit-by-bit





Random Tips & Tricks

- Loader { }
  - Visual element, to be replaced with contents of `source`
  - Dynamic loading + unloading for resource optimization
- View Switching now implicitly supported
  - loader
  - Views side-by-side, modify/animate x,y-values
- Stubs for testing (just Rectangle with a color and a size)
  - Incremental development

Qt Quick

Using QML in Qt/C++ Applications

Introduction

- You'll need to C++ side for creating a real application of your .qml file(s)
- There are four main classes in the `QtDeclarative` module for using QML from C++
  - `QDeclarativeView`
  - `QDeclarativeEngine`
  - `QDeclarativeComponent`
  - `QDeclarativeContext`
- Many QML elements also have a corresponding C++ class that gets instantiated when the element is used
  - `Item <-> QDeclarativeItem`
- In order to take `QtDeclarative` into use, add the following to your application .pro file:
  - `QT += declarative`

The Very Minimum

```
#include <QtGui/QApplication>
#include <QtCore/QUrl>
#include <QtDeclarative/QDeclarativeView>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QDeclarativeView canvas(QUrl("main.qml"));
    canvas.show();
    return app.exec();
}
```

QDeclarativeView

- A separate, simple view class is also provided
  - `QDeclarativeView` (derives from `QGraphicsView`)
- Can access the engine and root context through the view
  - `engine()`
  - `rootContext()`

QDeclarativeEngine

- Every application wishing to access QML from Qt/C++ needs at least one instance of `QDeclarativeEngine`
  - Provides an environment for instantiating QML components from C++
  - Allows configuration of global settings applying to all QML component instances
    - E.g. the `QNetworkAccessManager` instance and path for persistent storage
  - Multiple instances of this class are only needed, if the settings need to differ between QML component instances

QDeclarativeComponent

- A simple class used for loading QML documents
  - Each `QDeclarativeComponent` instance represents a single QML document
- The content can be given as a document URL or raw text
  - The URL can point to local file system or any network URL supported by `QNetworkAccessManager`
- Contains status information about the document
  - Null, Ready, Loading, Error

Example – Instantiating a QML Component

```
// Create the engine (root context created automatically as well)
QDeclarativeEngine engine;

// Create a QML component associated with the engine
// (Alternatively you could create an empty component and then set
// its contents with setData().)
QDeclarativeComponent component(&engine, QUrl("main.qml"));

// Instantiate the component (as no context is given to create(),
// the root context is used by default)
QDeclarativeItem *item =
  qobject_cast<QDeclarativeItem *>(component.create());

// Add item to a view, etc ...
```

QDeclarativeContext 1/3

- Each QML component is instantiated in a `QDeclarativeContext`
  - The engine automatically creates a default root context
- Additional sub-contexts can be created as needed
  - Sub-contexts are arranged hierarchically
  - The root context is the parent of all sub-contexts
  - The hierarchy is managed by the `QDeclarativeEngine`
- Data meant to be available to all QML component instances should be put in the engine's root context
- Data meant for a subset of component instances should be put in a sub-context

QDeclarativeContext 2/3

Forum Nokia

- Using a context you can expose C++ data and objects to QML

```
// main.qml
import Qt 4.7
Rectangle {
    color: myBackgroundColor
    Text {
          anchors.centerIn: parent
          text: "Hello Light Steel Blue World!"
    }
}

// main.cpp
QDeclarativeEngine engine;
// engine.rootContext() returns a QDeclarativeContext*
(engine.rootContext())->setContextProperty(“myBackgroundColor",
                                           QColor(Qt::lightsteelblue));
QDeclarativeComponent component(&engine, "main.qml");
QObject *window = component.create();  // Create using the root context
```

QDeclarativeContext 3/3

- This mechanism would also be used when providing a C++ model for e.g. a QML `ListView`

```
QDeclarativeEngine engine;

// Expose modelData (e.g. of type QAbstractItemModel) by the name
// myModel to QML
(engine.rootContext())->setContextProperty("myModel", modelData);

// Create a QML component
QDeclarativeComponent component(&engine,
   "import Qt 4.7 \n ListView { model: myModel }");

// Instantiate component
component.create();
```

Structured Data

- In case you have larger data sets to expose, consider exposing *a context object* `QObject` instead
  - All `QProperties` defined in the default object become available to the QML component instance by name
  - The data exposed this way can be made *writable* from QML as well!
  - Slightly faster than manually exposing multiple property values using `setContextProperty()`
- Multiple default objects can be added to the same QML component instance
  - Default objects added first take presedence over those added later
  - However, properties added with `setContextProperty()` take presedence over any default objects

Structured Data – A Simple Example

```
// MyDataSet.h
class MyDataSet : ... {
  ...
  // The NOTIFY signal informs about changes in the property's value
  Q_PROPERTY(QAbstractItemModel* myModel READ model NOTIFY modelChanged)
  Q_PROPERTY(QString text READ text NOTIFY textChanged)
  ...
};

// SomeOtherPieceOfCode.cpp exposes the QObject using e.g. a sub-context
QDeclarativeEngine engine;
QDeclarativeContext context(engine.rootContext());
context.setContextObject(new MyDataSet(...));
QDeclarativeComponent component(&engine, "ListView { model=myModel }");
component.create(&context);
```

Calling C++ Functions From QML

Forum Nokia

- Any public slot function of a `QObject` can be called from QML
- In case you do not want your function to be a slot, you can declare it as `Q_INVOKABLE`
  - `Q_INVOKABLE void myMethod();`
- These functions can have arguments and return types
- Currently the following types are supported:
  - `bool`
  - `unsigned int, int, float, double, real`
  - `QString, QUrl, QColor`
  - `QDate, QTime, QDateTime`
  - `QPoint, QPointF, QSize, QSizeF, QRect, QRectF`
  - `QVariant`

Example 1/2

Forum Nokia

```
// In C++:
class LEDBlinker : public QObject {
  Q_OBJECT
  // ...
public slots:
  bool isRunning();
  void start();
  void stop();
};

int main(int argc, char **argv) {
  // ...
  QDeclarativeContext *context =
        engine->rootContext();
  context->setContextProperty("ledBlinker",
        new LEDBlinker);
  // ...
}
```

```
// In QML:
import Qt 4.7

Rectangle {
  MouseArea {
    anchors.fill: parent
      onClicked: {
        if (ledBlinker.isRunning())
          ledBlinker.stop()
        else
          ledBlinker.start();
      }
  }
}
```

Example 2/2

- Notice that the same result could be achieved by declaring a ”running” property
  - Leads to much nicer code
  - Implementation of `isRunning()` and `setRunning()` omitted here for simplicity

```
// In C++:
class LEDBlinker : public QObject {
    Q_OBJECT
    Q_PROPERTY(bool running READ isRunning WRITE setRunning)
    // ...
};

// In QML:
Rectangle {
    MouseArea {
          anchors.fill: parent
          onClicked: ledBlinker.running = !ledBlinker.running
    }
}
```

Calling QML Functions From C++

- Obviously the reverse works as well – you can use functions declared in QML from your C++ code
  - Any function you declare in QML appears as a slot function in C++ – simply connect a signal to it !
  - As mentioned before, also any signals declared in QML can be connected to slots in C++

QML Components in Resource File 1/2

- Probably the most convenient way of including QML components in your Qt project is to put them into a resource file
  - Also JavaScript files can be included, of course
- Easier access to the files
  - No need to know the exact path to the file
  - Simply pass a URL pointing to the resource file
- *Resource files get compiled into the application binary*
  - These files are thus automatically distributed along with the binary, just like any other resource (e.g. images)

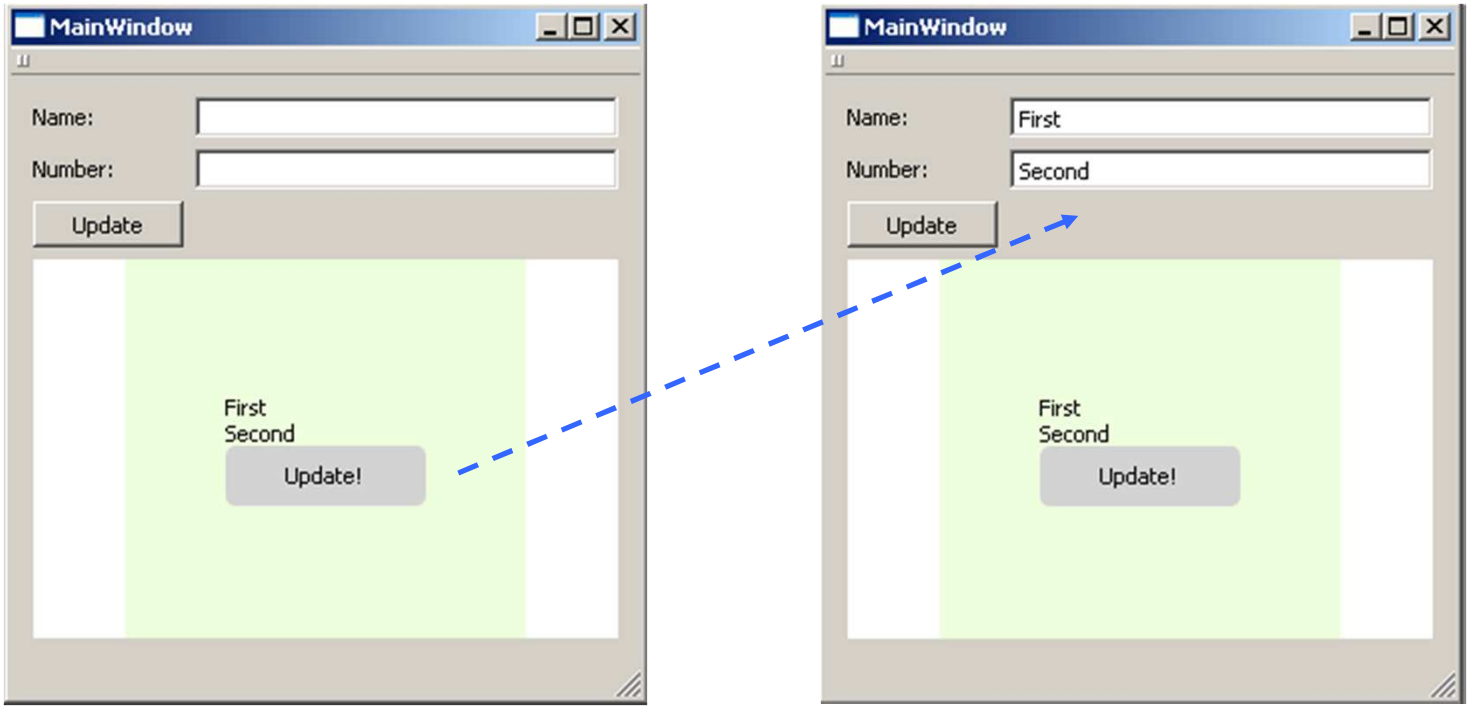
QML Components in Resource File 2/2

```
// MyApp.qrc
<!DOCTYPE RCC>
<RCC version="1.0">
  <qresource> <file>qml/main.qml</file> </qresource>
</RCC>

// MyObject.cpp
MyObject::MyObject() {
  component = new QDeclarativeComponent(engine,
        QUrl("qrc:/qml/main.qml"));
  if (!component->isError()) {
        QObject *myObject = component->create();
  }
}

// main.qml
import Qt 4.7
Image { source: "images/background.png" }
```

Qt Quick

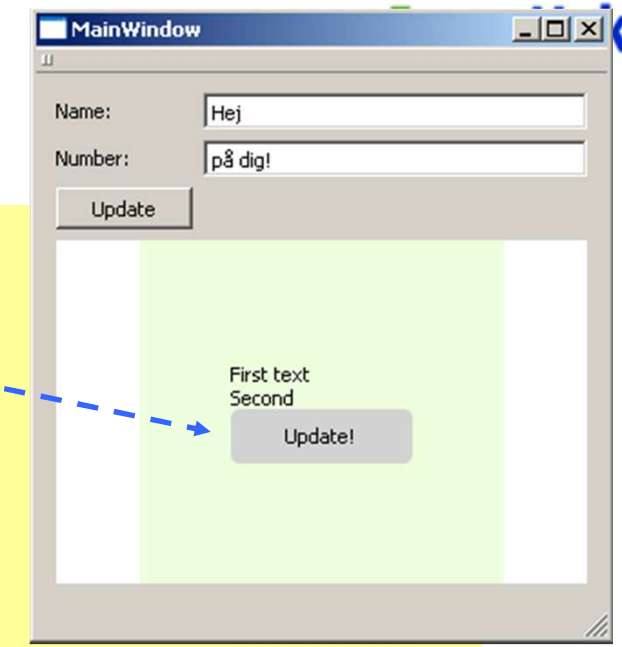
Concrete Example on QML/C++ Connections

From QML to C++


MainWindow
Name:
Number:
Update
First
Second
Update!
MainWindow
Name:
First
Number:
Second
Update
First
Second
Update!

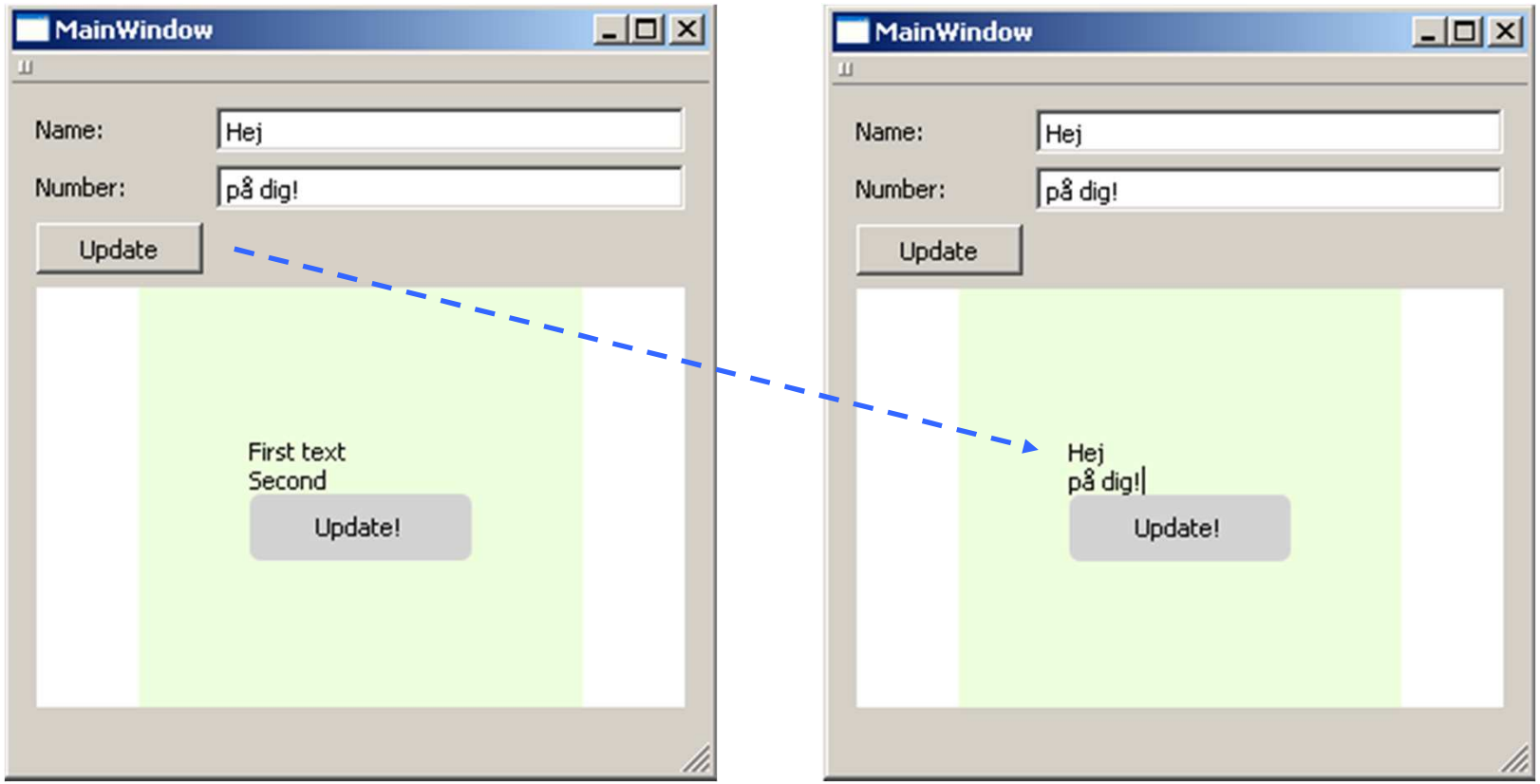
QML Side

```
Rectangle {
        id: button
        color: "lightgray"
        width: 100
        height: 30
        radius: 5
        Text {
            anchors.centerIn: parent
            text: "Update!"
        }
        MouseArea {
            anchors.fill: parent
            onClicked: mainWindow.updateValues(one.text,
                        two.text)
        }
    }
```



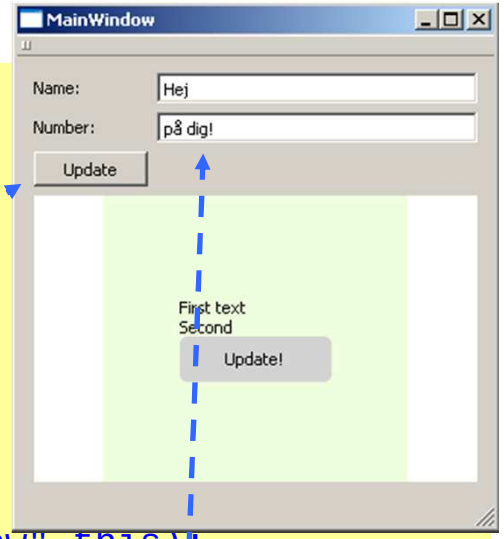
Calls slot function updateValues with these arguments

From C++ to QML ?


MainWindow
Name:
Hej
Number:
på dig!
Update
First text
Second
Update!
MainWindow
Name:
Hej
Number:
på dig!
Update
Hej
på dig!
Update!

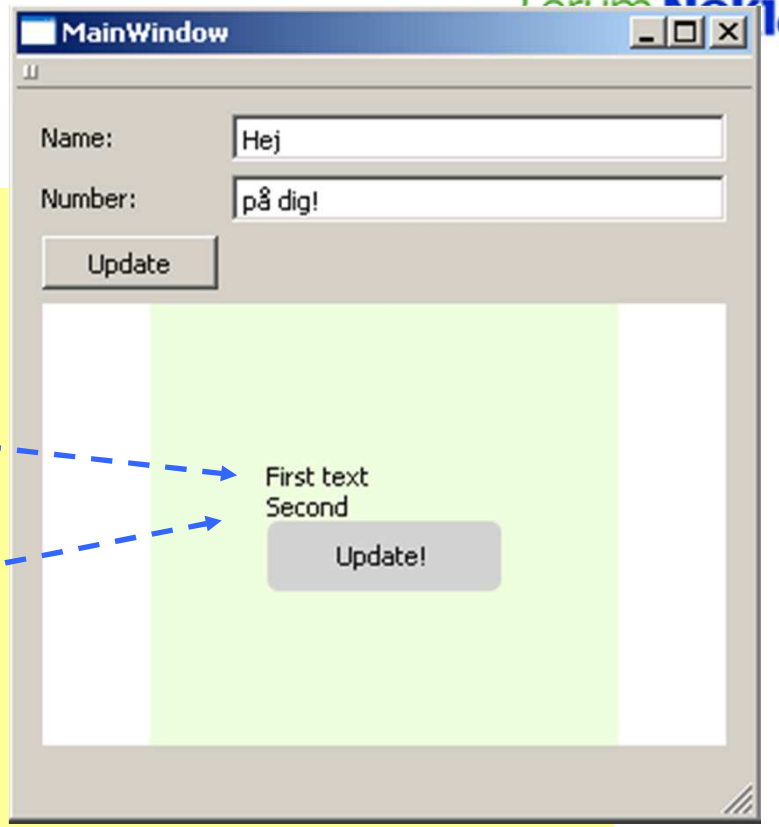
Qt Side, Emitting Signal when Clicked

```
class MainWindow : public QMainWindow
{
signals:
   void updateRequested(QString v1, QString v2);
…
}


MainWindow::MainWindow(QWidget *parent) : …
{
    m_view->rootContext()->setContextProperty("mainWindow",this);
    connect( ui->pushButton, SIGNAL(clicked()),
             this, SLOT(updateClicked()));
…


}


void MainWindow::updateClicked() {
    emit updateRequested(ui->lineEdit->text(),ui->lineEdit_2->text());
}
```



QML Side

```
Column {
        id: columnLayout
        anchors.centerIn: parent
        TextInput {
            id: one
            text: "First text"

        }
        TextInput {
            id: two
            text: "Second"
        }
         …


 Connections {
        target: mainWindow
        onUpdateRequested: {
            one.text = v1
            two.text = v2
        }
    }
```



"When mainWindow emits updateRequested()"

v1 and v2 are names of signal arguments

Qt Quick

C++ Data Models with QML

Introduction

- Normally, the data model comes from C++ side, not a static QML data model
  - Exposed model visible in the delegate through ”`model`” property
    - A default property
- Options
  - QList<QObject*>
  - QAbstractDataModel
  - QStringList

QList<QObject*>, C++ Side

```
class DataObject : public QObject
 {
     Q_OBJECT

     Q_PROPERTY(QString name READ name WRITE setName)
     Q_PROPERTY(QString color READ color WRITE setColor)
 ...
 };

 QList<QObject*> dataList;
 dataList.append(new DataObject("Item 1", "red"));
 dataList.append(new DataObject("Item 2", "green"));
 dataList.append(new DataObject("Item 3", "blue"));
 dataList.append(new DataObject("Item 4", "yellow"));

 QDeclarativeContext *ctxt = view.rootContext();
 ctxt->setContextProperty("myModel", QVariant::fromValue(dataList));
```

QList<QObject*>, QML Side

- The `QObject*` is available as the `modelData` property in the delegate
  - Use properties for the object

```
ListView {
   width: 100
   height: 100
   anchors.fill: parent
   model: myModel
   delegate: Component {
       Rectangle {
            height: 25
            width: 100
            color: model.modelData.color
            Text { text: name }
       }
   }
}
```

QAbstractItemModel

- Base class for all Qt/C++ data model classes
  - Currently only list data is supported by QML
- Expose model class normally
- Data provided via named *data roles,* by default
  - `display`
  - `decoration`
- QAbstractItemModel::setRoleNames()

```
Component {
        id: delegate
        Rectangle {
            width: 100
            height: 40
            color: "lightblue"
            Text {
                color: "darkgray"
                text: model.display
            }
}
```

QStringList

```
// main.cpp
QStringList dataList;
dataList.append("Fred");
dataList.append("Ginger");
dataList.append("Skipper");

QDeclarativeContext *ctxt = view.rootContext();
ctxt->setContextProperty("myModel", QVariant::fromValue(dataList));
```

```
// main.qml
ListView {
   width: 100
   height: 100
   anchors.fill: parent
   model: myModel
   delegate: Component {
       Rectangle {
            height: 25
            Text { text: modelData }
       }
   }
}
```

Thank You!

tuukka.ahoniemi@digia.com

www.digia.com