Qt Quick – GUI programming with QML

Timo Strömmer, Jan 4, 2011

Previous day quick quizz

- What are the ways to lay out items?
- What is the purpose of *id* property
- Which colors are visible in following:

```
Rectangle {
    width: 200
    height: 200

    Rectangle { color: "red"; width: 50; height: 50; x: 25; y: 25; z:1 }
    Rectangle { color: "yellow"; width: 50; height: 50; x: 25; y: 25 }

    Rectangle { color: "black"; width: 50; height: 50; x: 50; y: 50 }
    Rectangle { color: "green"; width: 50; height: 50; x: 50; y: 50 }
}
```

Contents – Day 2

- Dynamic object management
    - Inline components
    - Dynamic loading
- Building fluid user interfaces
    - Animations
    - States and transitions
- User interaction
    - Mouse and key
    - Interactive containers

Contents – Day 2

- Adding data to GUI
  - Data models
  - Views
  - Delegates

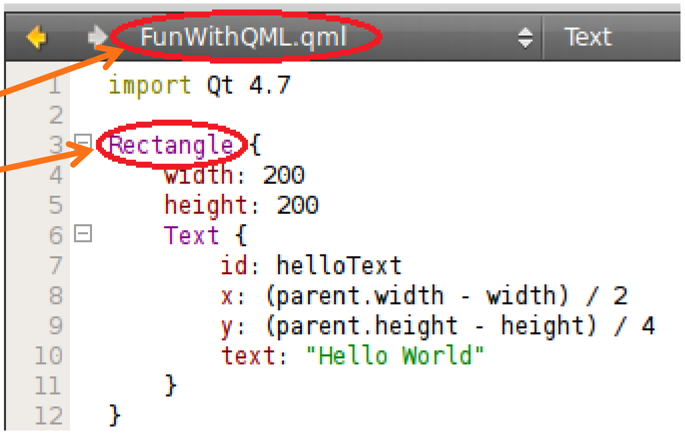
Component and script files, dynamic object loading

STRUCTURING QML PROGRAMS

QML components

- Refresher from yesterday

FunWithQML extends Rectancle

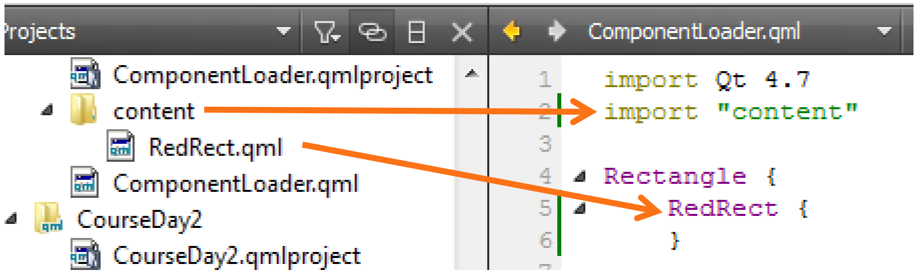


```
import Qt 4.7

Rectangle {
    width: 200
    height: 200
    Text {
        id: helloText
        x: (parent.width - width) / 2
        y: (parent.height - height) / 4
        text: "Hello World"
    }
}
```

Component files

- The *import* statement can be used to reference QML files in other directories
  - Single file import
  - Directory import



```
import Qt 4.7
import "content"

Rectangle {
    RedRect {
    }
```

- Imported directory can be *scoped*

```
import Qt 4.7
import "content" as Content

Rectangle {
     Content.RedRect {
     }
```

Script files

- The *import* statement also works with JavaScript
  - Can import *files*, not directories
  - Must have the *as* qualifier

```
import "js/startup.js" as Startup

Rectangle {
    Component.onCompleted: Startup.loadItems(rootRect);
```

Property scopes

- Properties of components are visible to child components
  - But, considered bad practice

Main.qml

```
Rectangle {
    width: 200
    height: 200
    property string inheritedText: "x"
    RedRect { }
}
```

RedRect.qml

```
Rectangle {
    width: 25
    height: 25
    x: 25; y: 25
    color: "red"
    Text {
        anchors.fill: parent
        verticalAlignment: Text.AlignVCenter
        horizontalAlignment: Text.AlignHCenter
        text: inheritedText
    }
}
```

Property scopes

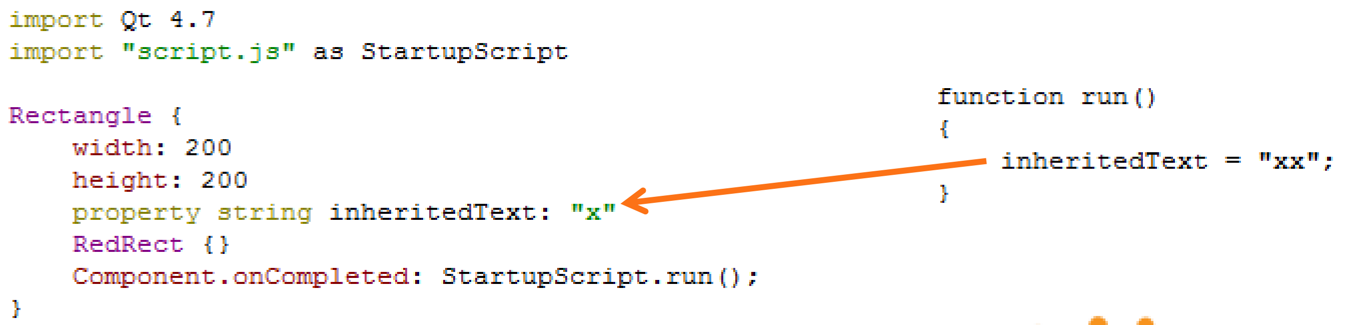
- Instead, each component should provide an API of it’s own

```
Rectangle {
    width: 200
    height: 200
    property string inheritedText: "x"
    RedRect {
        text: inheritedText
    }
}
```

```
Rectangle {
    property alias text: text.text
    width: 25
    height: 25
    x: 25; y: 25
    color: "red"
    Text {
        id: text
        anchors.fill: parent
        verticalAlignment: Text.AlignVCenter
        horizontalAlignment: Text.AlignHCenter
        text: ""
    }
}
```

Script scopes

- Same scoping rules apply to scripts in external JavaScript files
  - i.e. same as replacing the function call with the script
  - Again, not good practice as it makes the program quite confusing



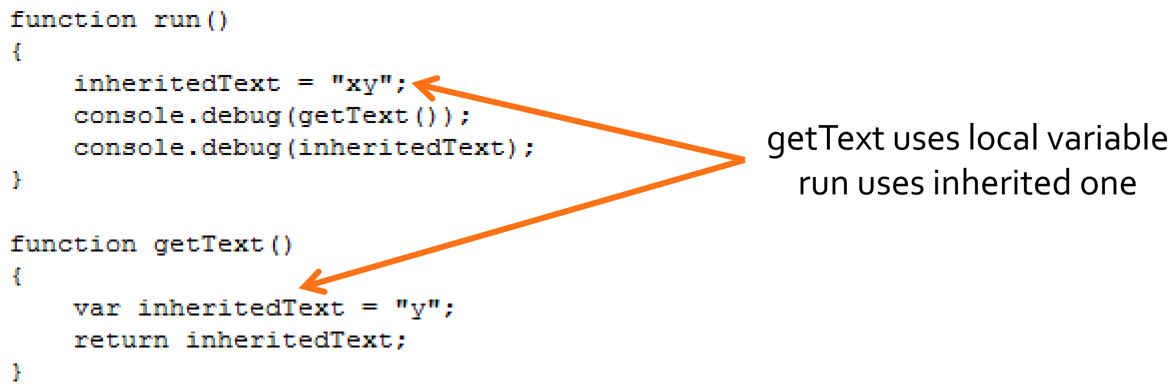
```
import Qt 4.7
import "script.js" as StartupScript

Rectangle {
    width: 200
    height: 200
    property string inheritedText: "x"
    RedRect {}
    Component.onCompleted: StartupScript.run();
}
```

```
function run()
{
    inheritedText = "xx";
}
```

JavaScript scoping

- If script function declares variables with same name, the script variable is used



```
function run()
{
    inheritedText = "xy";
    console.debug(getText());
    console.debug(inheritedText);
}

function getText()
{
    var inheritedText = "y";
    return inheritedText;
}
```

getText uses local variable
run uses inherited one

Inline components

- Components can be declared *inline*
  - *Component* element

```
Component {
    id: helloComponent
    Text { text: "Loaded from: " + helloComponent.url }
}
```

  - Useful for small or private components
    - For example data model delegates
  - *Loader* can be used to create instances
    - *Loader* inherits *Item*
    - Can be used to load components from web
- Example in *ComponentLoader* directory

Dynamic loading

- In addition to *Loader*, components can be loaded dynamically via script code
  - *Qt.createComponent* loads a *Component*
    - File or URL as parameter
  - *component.createObject* creates an instance of the loaded component
    - Parent object as parameter
  - *Qt.createQmlObject* can be used to create QML objects from arbitrary string data
- Example in *ScriptComponents* directory

Overview of QML animations

BUILDING FLUID GUI

Animations overview

- *Animation* changes a property gradually over a time period
    - Brings the ”fluidness” into the UI
- Different types for different scenarios
- Supports grouping and nesting

Animation basics

- All animations inherit from *Animation* base component
  - Basic properties (just like *Item* for GUI)
- Properties for declarative use:
  - *running*, *paused*, *loops*, *alwaysRunToEnd*
- Can also be used imperatively:
  - *start*, *stop*, *pause*, *resume*, *restart*, *complete*

Animation types

- Property value sources
- Behavioral
- Standalone
- Signal handlers
- State transitions

Animation types

- *Property value source* animation is run as soon as the target object is created
  - Animation provides the property value
  - *Animation on Property* syntax
    - Starts at *from* or current value
    - Ends at *to*
    - Lasts *duration* milliseconds

```
Rectangle {
    width: 200
    height: 200
    Text {
        x: 66
        text: "Hello World"
        PropertyAnimation on y {
            from: 0
            to: 93
            duration: 2000
        }
    }
}
```

Animation types

- *Behavioral* animation
  - Default animation that is run when property changes
  - *Behavior on Property* syntax

```
Rectangle {
    id: redRect
    color: "red"
    width: 25
    height: 25
    Behavior on y { NumberAnimation { duration: 1000 } }
}
```

  - No *from* or *to* needed, since old and new values come from the property change

Animation types

- *Standalone* animations are created as any other QML object
  - Attached to *target* object
    - Affects a *property* or *properties*
    - *from* optional, *to* mandatory
  - Need to be explicitly started

```
NumberAnimation {
    id: standalone
    target: redRect
    property: "x"
}

MouseArea {
    anchors.fill: parent
    onClicked: {
        standalone.to = mouseX;
        standalone.running = true;
    }
}
```

Animation types

- *Signal handler* animation is quite similar to standalone animation
  - *Start* is triggered by the signal
  - Otherwise same rules
    - Needs to be bound to *target* and *property*
    - *from* optional, *to* mandatory

```
MouseArea {
    anchors.fill: parent
    onClicked: PropertyAnimation {
        target: someText
        property: "x"
        to: redRect.x
    }
}
```

- More about *state transitions* in later slides

Animation types

- Example code in *AnimationExamples* directory
  - Uses *NumberAnimation* for various scenarios

Animation objects

- The actual animation is built from animation objects
  - *PropertyAnimation* and it's derivatives
    - *NumberAnimation, SmoothedAnimation, ColorAnimation, RotationAnimation, SpringAnimation*
  - Grouping and nesting
    - *SequentialAnimation, ParallelAnimation, PauseAnimation*
  - GUI layout changes
    - *AnchorAnimation, ParentAnimation*

Animation grouping

- Animations can be grouped to build more complex scenarios
  - *SequentialAnimation* is a list of animations that is run one at a time
  - *ParallelAnimation* is a list of animations that is run simultaneously
  - *PauseAnimation* is used to insert delays into sequential animations

Animation grouping

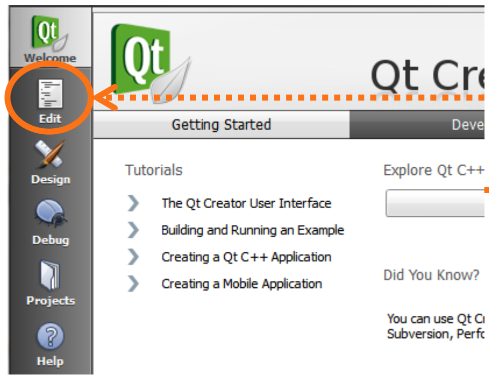
- Sequential and parallel animations can be nested
  - For example, a parallel animation may contain multiple sequential animations
- Example in *AnimationGrouping* directory
  - Also uses *ColorAnimation*

GUI states and animated state transitions

BUILDING FLUID GUI

GUI states

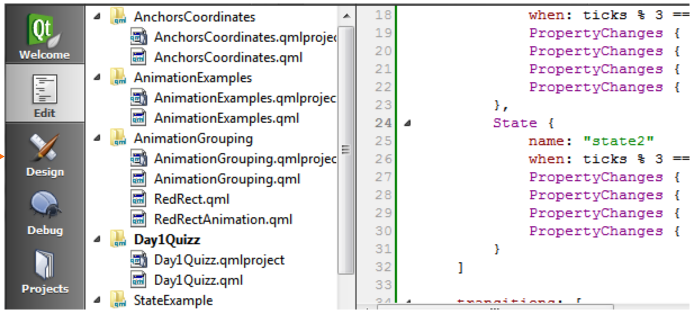
- A *state* represents a *snapshot* of a GUI



- The Qt Creator User Interface
- Building and Running an Example
- Creating a Qt C++ Application
- Creating a Mobile Application

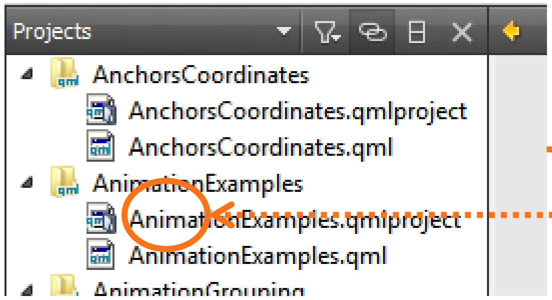
You can use Qt C
Subversion, Perfo

Click on "Edit"

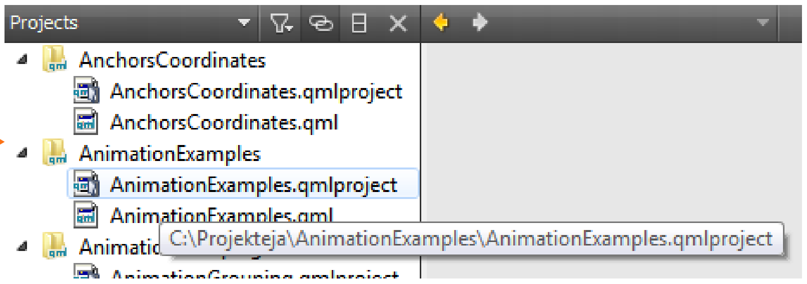


- AnchorsCoordinates
  - AnchorsCoordinates.qmlprojec
  - AnchorsCoordinates.qml
- AnimationExamples
  - AnimationExamples.qmlprojec
  - AnimationExamples.qml
- AnimationGrouping
  - AnimationGrouping.qmlprojec
  - AnimationGrouping.qml
  - RedRect.qml
  - RedRectAnimation.qml
- **Day1Quizz**
  - Day1Quizz.qmlproject
  - Day1Quizz.qml
- StateExample

```
            when: ticks % 3 ==
            PropertyChanges {
            PropertyChanges {
            PropertyChanges {
            PropertyChanges {
        },
        State {
            name: "state2"
            when: ticks % 3 ==
            PropertyChanges {
            PropertyChanges {
            PropertyChanges {
            PropertyChanges {
        }
    ]
```



- AnchorsCoordinates
  - AnchorsCoordinates.qmlproject
  - AnchorsCoordinates.qml
- AnimationExamples
  - AnimationExamples.qmlproject
  - AnimationExamples.qml
- AnimationGrouping



- AnchorsCoordinates
  - AnchorsCoordinates.qmlproject
  - AnchorsCoordinates.qml
- AnimationExamples
  - AnimationExamples.qmlproject
  - AnimationExamples.qml
- Animatic

C:\Projekteja\AnimationExamples\AnimationExamples.qmlproject

Mouse on file name

GUI states

- States are usually applicable at many levels, regardless of problem complexity
  - i.e. whole program vs. small trinket
- *Transitions* between states
  - Response to user interaction or other events
  - Many transitions may be run parallel
  - May be animated with QML

GUI states in QML

- State framework built into QML:
  - Every GUI *Item* has a *state* property, *default state* and a list of *states*
    - States are identified by *name*, default has no name
  - Each *State* object inherits properties from default state and declares the differences
    - *PropertyChanges* element
  - A state may inherit properties from another state instead of the default
    - *extend* property

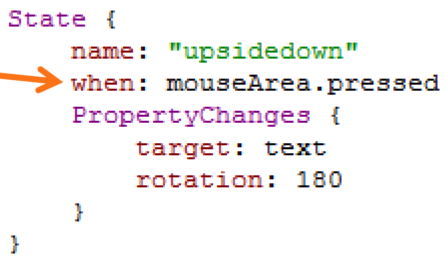
```
Rectangle {
        id: rect
        width: 200
        height: 200
}
```

```
states: State {
        name: "shorter"
        PropertyChanges {
                target: rect;
                height: 100
        }
}
```

```
State {
        name: "smaller"
        extend: "shorter";
        PropertyChanges {
                target: rect
                width: 100
        }
}
```

GUI states

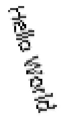
- Only one state is active at a time
  - So, only properties from *default* and changes from *active* state are in use
  - State can be activated via script or with the help of *when* binding
- Example in *SimpleState* directory



```
State {
    name: "upsidedown"
    when: mouseArea.pressed
    PropertyChanges {
        target: text
        rotation: 180
    }
}
```

State transitions

- The transitions between states are declared separately from the states
  - List of *transitions* under the *Item*
  - Quite similar to *ParallelAnimation*
    - Although, doesn't inherit Animation
- Example in *SimpleStateTransition* directory

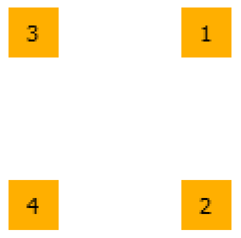


State transitions

- All transitions are applied by default
  - Can be scoped with *from* and *to*
    - Both are bound to *state name*
  - Transition overrides *Behavior on <property>*
- Transition animations are run in parallel
  - Can be wrapped into *SequentialAnimation*
  - Transition *reversible* flag might be needed
    - Runs sequential animations in reverse order

State examples

- *SequentialTransition* directory
  - Transitions quizz
- Mapping the *AnimationGrouping* example into state framework
  - *StateExample* directory



Advanced animation topics

BUILDING FLUID GUI

Layout animations

- The *anchors* of GUI Items can be changed while application is running
  - *AnchorChanges* element within a state
    - Re-anchors the item to another valid target
  - *AnchorAnimation* can be applied to state transitions list to animate the changes
    - Animates position and dimensions
- Some quirks involved
  - Example in *AnchorAnimations* directory

Layout animations

- In addition to anchor changes, the *parent-child* relationship of items can be changed
  - *ParentChange* element within a state
    - Changes the *parent* of an item
    - Optionally also the coordinates, size and transform
  - New relative coordinates
  - Requires re-anchoring within new parent
- Example in *ParentChange* directory

More animation objects

- *RotationAnimation* for angles
    - Configurable rotation direction
    - Uses shortest path by default
        - i.e. instead of going back from 359 to 0
- *SmoothedAnimation* for movement
    - For example translations
    - Can use *velocity* instead of *duration*
        - So speed doesn’t depend on distance moved
    - Easing curve built in

More animation objects

- *SpringAnimation* for spring-like movement
    - *spring*, *damping* and *mass*
- Some examples in *TransformAnimations* directory
    - Although, note that these animations are not in any way restricted to transformations

Easing curves

- Property and anchor animations may have an *easing curve*
  - Results in non-linear property change
  - Quite a lot of pre-defined curves
    - Check *PropertyAnimation.easing.type* for details
- Quick task:
  - Open *AnimationGrouping* example and add some easing curves

Script hooks

- *StateChangeScript* is run when a state is entered
  - Before state transitions are run

```
State {
    name: "state1"
    StateChangeScript {
        script: console.log("Entered state 1")
    }
```

- *ScriptAction* within *SequentialAnimation*
  - Can relocate a *StateChangeScript* call

Also, don't forget on<Property>Changed hook from first day slides

```
transitions: [
    Transition {
        SequentialAnimation {
            NumberAnimation { properties: "x,y"; duration: 500 }
            ScriptAction { scriptName: "changeScript" }
```

```
State {
    name: "state1"
    StateChangeScript {
        name: "changeScript"
        script: console.log("Played state 1 animation")
    }
```

Animation actions

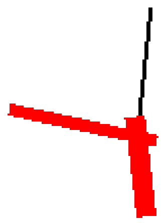
- *ScriptAction* can also run without states
  - Use *script* property instead of *scriptName*

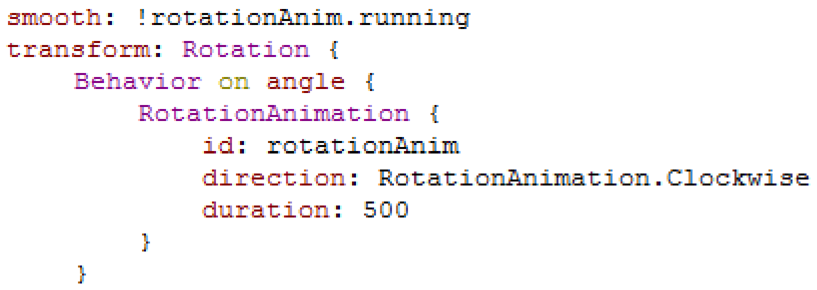
```
SequentialAnimation {
    NumberAnimation { properties: "x,y"; duration: 500 }
    ScriptAction { script: console.log("Played animation") }
}
```

- *PropertyAction* changes a property without performing animations
  - For example *bool* flags, z-value etc.

Animation notes

- Transformations (especially rotation) may produce jagged lines (*aliasing*)
    - Each *Item* has *smooth* property for anti-aliasing
- Smoothing is expensive operation
    - Might be good idea to try disabling smoothing for the duration of animations





```
smooth: !rotationAnim.running
transform: Rotation {
    Behavior on angle {
        RotationAnimation {
            id: rotationAnim
            direction: RotationAnimation.Clockwise
            duration: 500
        }
    }
```

See also *ClockExample*

Handling mouse and keyboard input

USER INTERACTION

Mouse and key events

- Mouse and keys are handled via *events*
  - *MouseEvent* contains position and button combination
    - Posted to *Item* under cursor
  - *KeyEvent* contains key that was pressed
    - Posted to *Item*, which has the *active focus*
  - If item doesn't handle it, event goes to parent
    - When *accepted* properties is set to *true*, the event propagation will stop
  - Events are *signal parameters*

Mouse input

- *MouseArea* element has already been used in most of the examples
  - Works for desktop and mobile devices
    - Although, some signals will not be portable
  - *pressed* property
    - Any mouse button (*pressedButtons* for filtering)
    - Finger-press on touch screen
  - Position of events:
    - *mouseX* and *mouseY* properties
    - *mouse* signal parameter

```
MouseArea {
    onClicked: {
        clickX = mouseX
        clickY = mouseY
    }
}
```

```
MouseArea {
    onClicked: {
        clickX = mouse.x
        clickY = mouse.y
    }
}
```

Mouse drag

- *MouseArea* can make an item *draggable*
  - Works with mouse and touch
- Draggable items may contain children with mouse handling of their own
  - The child items must be children of the *MouseArea* that declares dragging
    - *MouseArea* inherits *Item*, so may contain child items
    - *drag.filterChildren* property
- Example in *MouseDrag* directory

Keyboard input

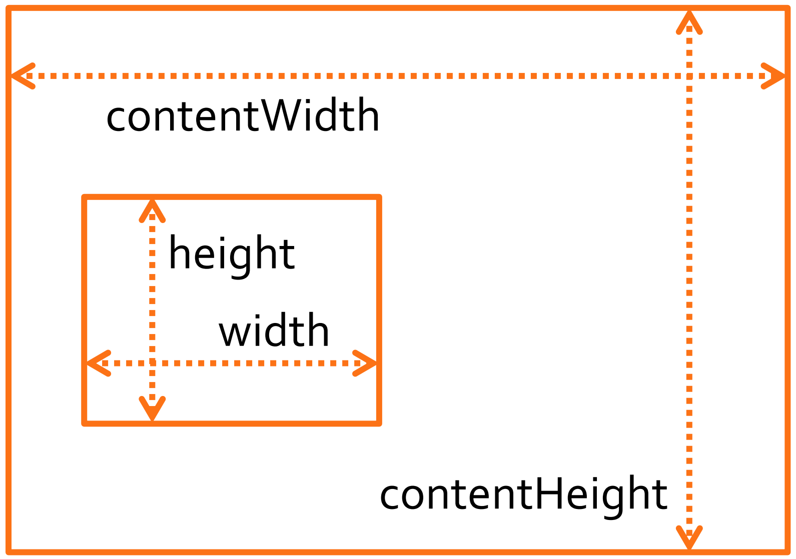
- Each *Item* supports keyboard input
  - *Keys* and *KeyNavigation* attached properties
    - *Keys.on<Key>Pressed* signals
    - *KeyNavigation.up* / *down* / *left* / *right* properties
  - Key events arrive to item with *activeFocus*
    - Can be forwarded to other items
    - Ignored if none of items is focused
  - Setting focus property to *true* to get focus

Keyboard input

- *FocusScope* element can create focus groups
  - Needed for re-usable components
    - Internals of component are not visible
  - Invisible item, similarly as *MouseArea*
    - One item within each *FocusScope* may have focus
    - Item within the *FocusScope*, which has focus gets key events
- Example in *KeyboardFocus* directory

Flickable element

- Scrollable container for other elements
  - Drag or flick to scroll
  - Scrollbars *not* built-in
    - ScrollBar example available in QML documentation

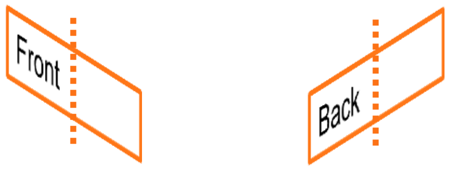


Flickable element

- *Flickable* mouse events
    - Drag events are intercepted by the flickable
    - Mouse clicks go to children
        - Similarly as MouseArea with drag enabled
    - Control via *interactive* and *pressDelay* properties
- Example in *FlickableExample* directory
    - Also contains *StateChangeScript* and *PropertyAction* examples

Flipable element

- *Flipable* is a two-sided container
  - Card with *front* and *back* items
  - Must use *Rotation* transform to see the back
    - Either via *x* or *y* axis, *z* won't help
    - Will not go upside-down via x-axis rotation
  - States and transitions not pre-implemented
    - Use for example *RotationAnimation* for transition
- Example in *FlipExample* directory



Models, views and delegates

DISPLAYING DATA

Data elements

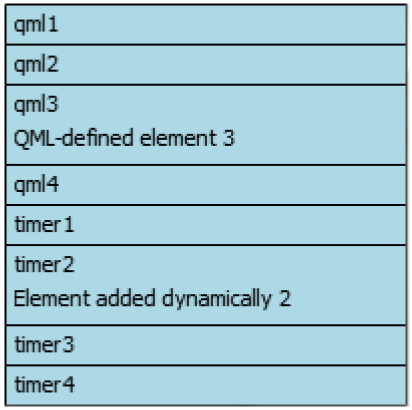
- Data elements are divided into three parts
  - *Model* contains the data
    - Each data element has a *role*
  - *View* defines the layout for the data elements
    - Pre-defined views: *ListView*, *GridView* and *PathView*
  - *Delegate* displays a single model element
    - Any *Item*-based component works

Data models

- *ListModel* for list of data elements
  - Define *ListElement* objects in QML code
    - *ListElement* consists of *roles*, not *properties*
    - Syntax is similar to QML properties (*name: value*)
    - But, cannot have scripts or bindings as value
  - Add JavaScript objects dynamically
    - Any dictionary-based (*name: value)* object will work
    - Works also with nested data structures

Data models

- *ListModel* is manipulated via script code
  - *append, insert, move, remove, clear*
  - *get, set, setProperty*
  - Changes to model are automatically reflected in the view(s) which display the model
    - Although, changes via *WorkerScript* need *sync*
- Example in *SimpleDataModel* directory



qml1

qml2

qml3
QML-defined element 3

qml4

timer1

timer2
Element added dynamically 2

timer3

timer4

Data models

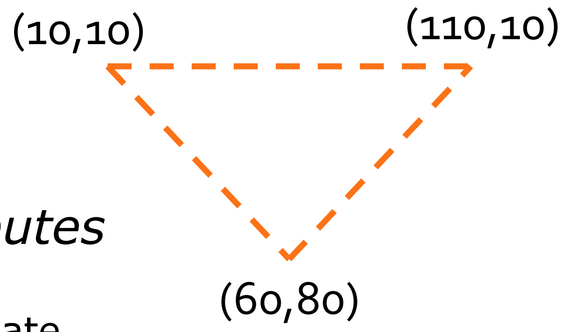
- Other model types
  - *XmlListModel* for mapping XML-data (for example from web services) into QML view
    - Uses *XPath* queries within list elements (*XmlRole*)
  - *FolderListModel* from QtLabs experimental
    - Displays local file system contents
  - *VisualItemModel* for GUI *Items*
  - *VisualDataModel*
    - Can visualize Qt/C++ *tree models*
    - May share GUI *delegates* across views

Data views

- QML has three views
  - *ListView* displays it’s contents in a list
    - Each element gets a row or column of its own
    - Compare to *Row* or *Column* positioners
  - *GridView* is two-dimensional representation
    - Compare with *Grid* positioner
  - *PathView* can be used to build 2-dimensional paths where elements travel

Path view

- The *PathView* component declares a *path* on which the model elements travel
  - *Path* consists of path segments
    - *PathLine*, *PathQuad*, *PathCubic*
    - Start and end point + control points
  - Each segment may have path *attributes*
    - Interpolated values forwarded to delegate
- Example in *PhotoExample* directory



Data view notes

- Note the lack of *tree view*
  - Probably not good for small screens
- *Repeater* was used in earlier example
  - Not a view, but can work with *model* and *delegate*
    - Or directly with GUI Items

Data views

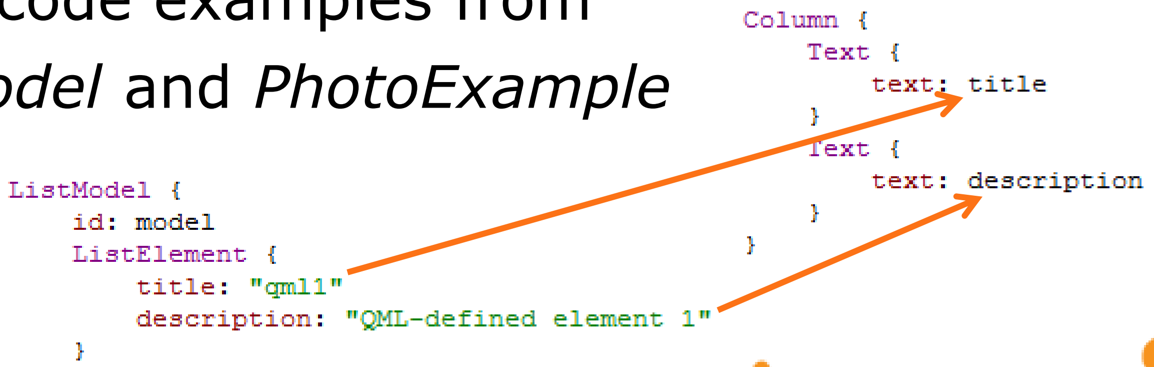
- Interaction with views
  - List and grid views inherint from *Flickable*
    - Content can be scrolled (no scrollbars though)
  - Path view uses drag and flick to move the items around the path
  - Delegates may implement mouse handlers
    - Same rules as with *Flickable* nested mouse areas

GUI delegates

- A *delegate* component maps a model entry into a GUI *Item*
  - In *VisualItemModel* each entry is GUI item
- Delegate objects are created and destroyed by the view as needed
  - Saves resources with lots of items
    - Remember dynamic object management slides at beginning of this day
  - Cannot be used to store any state

GUI delegates

- The delegate may access the list model roles by name
  - If role name is ambiguous, use *model* attached property
  - Special *index* role also available
- See delegate code examples from *SimpleDataModel* and *PhotoExample*

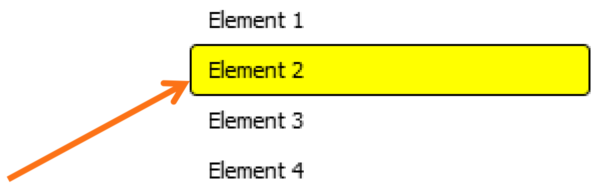


```
Column {
    Text {
        text: title
    }
    Text {
        text: description
    }
}
```

```
ListModel {
    id: model
    ListElement {
        title: "qml1"
        description: "QML-defined element 1"
    }
```

View selection

- Each view has *currentIndex* property
  - *ListView* and *GridView* have *currentItem*
  - Represents the selected element
- View has *highlight* delegate
  - Draws something *under* the current item
  - Highlight moves with *SmoothedAnimation*
    - Can be customized with *highlightFollowsCurrentItem*
- Example in *ViewHighlight* directory

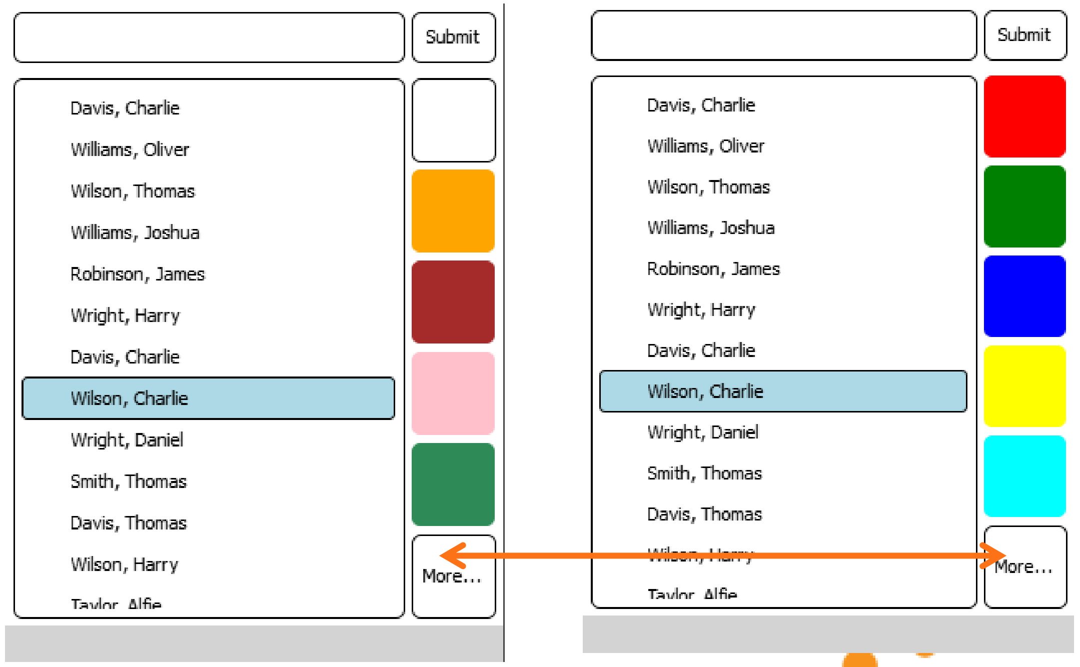


Adding states and transitions

FLUID GUI EXERCISES

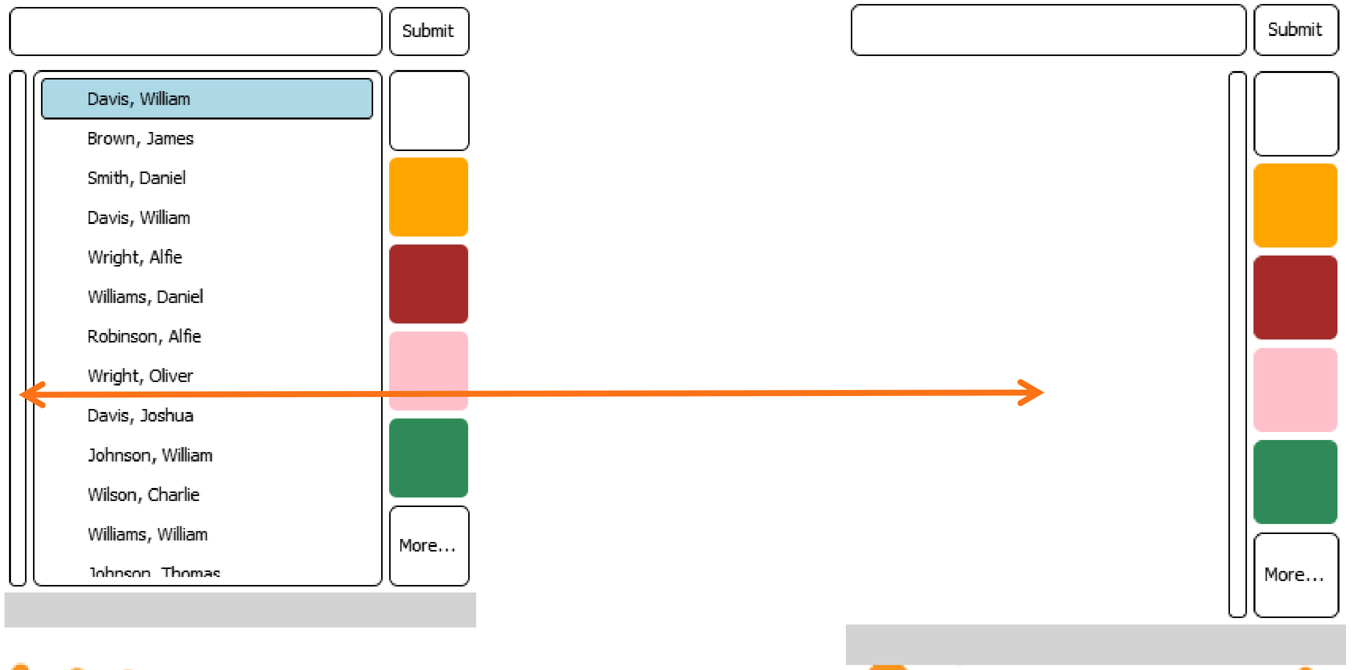
States and transitions

- Replace one of the original colors with a button, which flips the color list over and reveals more colors



States and transitions

- Add an area to left side, which slides in when mouse is clicked on it
  - Slides back when clicked again

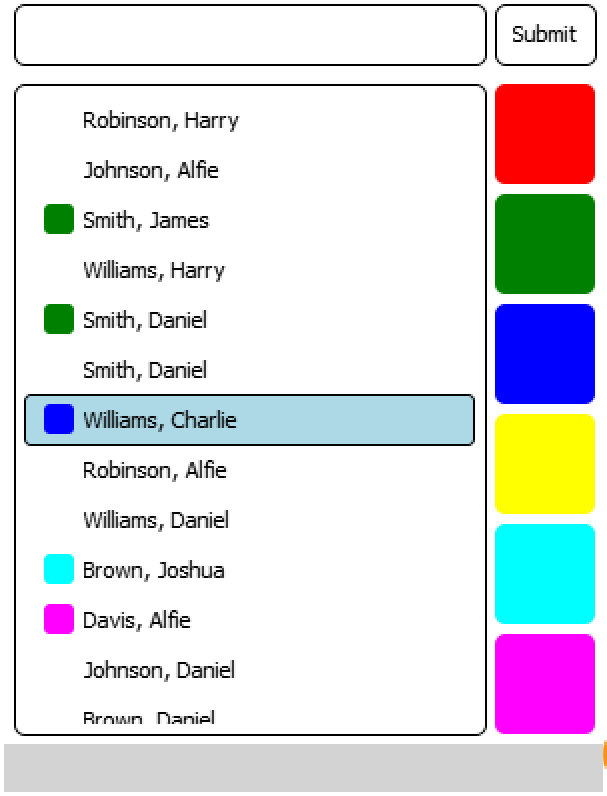


Implementing a model and view

DATA MODEL EXERCISE

Data model exercise

- Add a *ListModel* to the central area of day 1 exercise
  - Fill with random names
    - Generator example in *CourseDay2/ListArea.qml*
  - Add selection support to model
  - When a color on right side is clicked, tag the name with that color
    - Fade-in / fade-out the tag rectangle



Robinson, Harry
Johnson, Alfie
Smith, James
Williams, Harry
Smith, Daniel
Smith, Daniel
Williams, Charlie
Robinson, Alfie
Williams, Daniel
Brown, Joshua
Davis, Alfie
Johnson, Daniel
Brown, Daniel

<symbio>

SERIOUS ABOUT SOFTWARE