Qt Quick – From bottom to top

Timo Strömmer, Feb 11, 2011

Contents – Day 2

<symbio>

Shared data objects

CORE FEATURES

Shared data objects

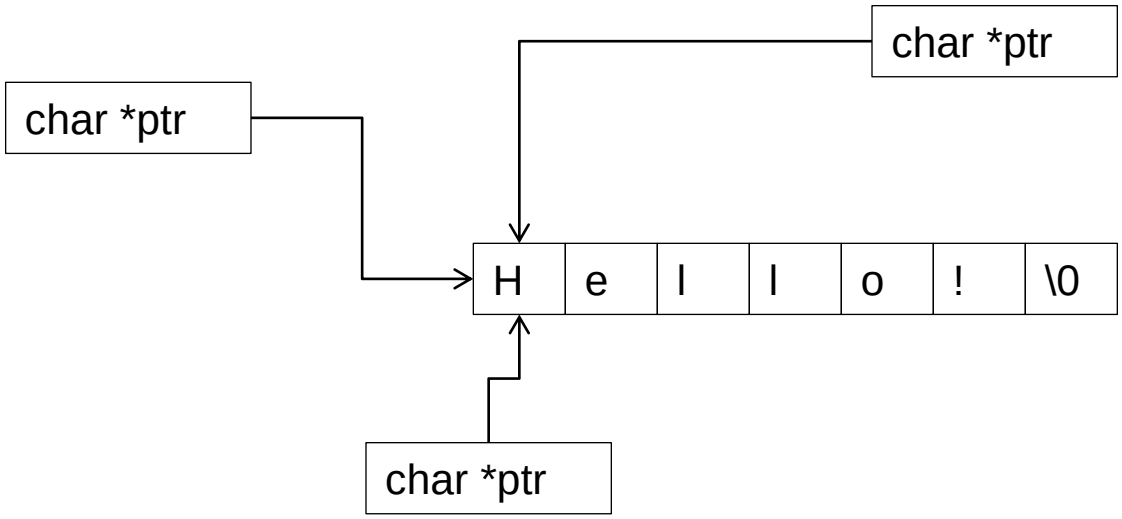
<symbio>

- A *shared data object* doesn't store the object data by itself
  - Instead, data is *implicitly shared*
    - With copy-on-write semantics
  - Easier to use that just pointers
    - The object can be thought as *simple value type*
- Examples:
  - Strings, images, collections

Implicit sharing

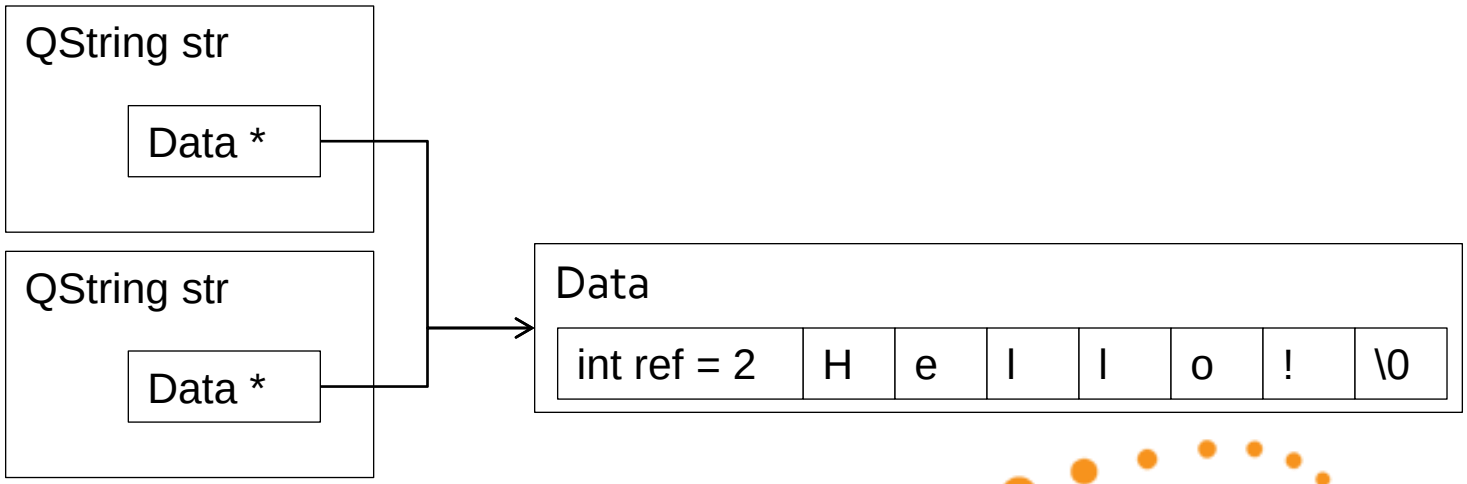
<symbio>

- In normal C++ an object is allocated and a pointer to it is passed around
  - Care must be taken that object is not deleted while it's still being pointed to



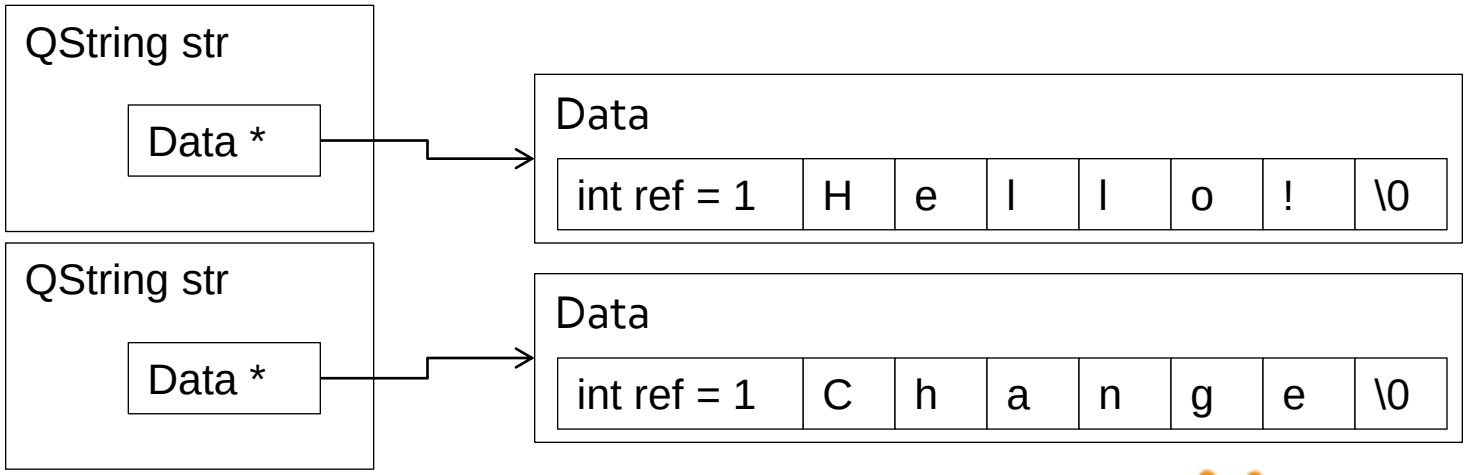
Implicit sharing

- In implicit sharing, a *reference counter* is associated with the data
  - Data pointer is wrapped into a *container* object, which takes care of deleting the data when reference count reaches 0



Implicit sharing

- Implicitly shared objects can be treated as simple values
  - Only the pointer is passed around



Terminology

- Copy-on-write
  - Make a shallow copy until something is changed
- Shallow copy
  - Copy just the pointer, not actual data
- Deep copy
  - Create a copy of the data

Strings

<symbio>

- Two types of string
  - UNICODE strings (QString)
  - Byte arrays (QByteArray)
- In general, QString should be used
  - UNICODE, so can be localized to anything
  - Conversion between the two types is easy, but might have unexpected performance issues

Strings and implicit sharing

- Strings are implicitly shared, so in general, should be treated as a value
  - Returned from functions like value
  - Stored into objects as values
  - Function parameters should use constant reference, not value
    - *const QString &*

```
QString HelloWorld::hello() const
{
    QString str = "Hello World";
    return str;
}
```

```
void HelloWorld::setDescription(const QString &desc)
{
    description = desc;
}
```

String operations

- In Qt, a string can be changed
  - Thus, differs from java immutable strings
  - Modifying a string in-place is more efficient (especially with *reserve()* function)
    - However, some care must be taken to avoid changes in unexpected places

```
void HelloWorld::changeString(QString &str)
{
    str += "_changed";
}

QString HelloWorld::createNewString(const QString &str)
{
    return str + "_changed";
}
```

String operations

- QString supports various operators
  - ’+’, ’+=’, ’>’, ’<’, ’<=’, ’>=’, ’==’, ’!=’
  - Also work with literals
  - Character access with []

```
QString str1 = "hello";
QString str2 = "world";

if (str1 == "hello" && str2 != "wolrd") {
    qDebug("True");
}
```

Generic containers

- List containers
  - *QList*, *QLinkedList*, *QVector*, *QStack*, *QQueue*
    - Usually *QList* is best for ordinary tasks
  - *QStringList* for strings
- Associative containers
  - QSet, QMap, QHash, QMultiMap, QMultiHash
  - QMap for sorted, QHash for unsorted items

List containers

- Lists are index-based, starting from 0
  - Fast access if index is known, slow to search
- Adding and removing items
  - append, insert, '+=', '<<'
- Accessing items
  - at, '[]'

```
QList<QString> strings;
strings.append("1");
strings << "2" << "3" << "4";
strings.insert(2, "2");
strings.removeOne("2");

qDebug("%s", qPrintable(strings[2])); // 3
```

Foreach statement

- Can be used to iterate over lists
- Takes a shallow copy of the container
  - If original container is modified while in loop, the one used in the loop remains unchanged

```
QString hello = "Hello World !!!";
QStringList strList = hello.split(" ");
foreach (QString str, strList) {
    qDebug("Part: %s", qPrintable(str));
}
```

Associative containers

<symbio>

- Associative containers are used to map keys to values
  - In QSet, key and value are the same
    - QSet<String>
  - Other containers have separate keys and values
    - QHash<QString,QString>
  - Normal versions have one-to-one mapping, *multi*-versions accept multiple values for single key
    - QMultiMap<QString, QObject *>

<symbio>

Object model

CORE FEATURES

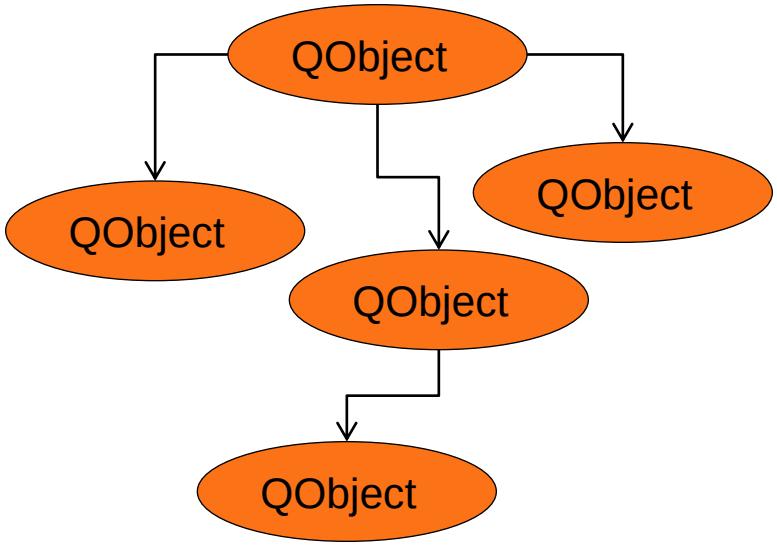
Object model

<symbio>

- Usual Qt program is based around a tree-based hierarchy of objects
  - Helps with C++ memory management
  - Based on *QObject* class
  - Do not confuse with class inheritance

Object model

- A QObject may have a *parent* object and number of *child* objects
- Object without parent is called a *root* object
- When an object is deleted, it will also delete all it's children



Object model and GUI

- All GUI components inherit from QWidget, which in turn inherits from QObject
  - Thus, GUI widgets are also arranged into tree hierarchy
  - The root widget is a *window*

  - Enabling / disabling or showing / hiding a widget will also affect its children

Signals & slots

CORE FEATURES

Signals and slots

- Qt way of making callback functions simple
  - Example cases
    - What happens when user presses a GUI button
    - What happens when data arrives from network
  - Similar semantics as with Java listeners
- A *signal* is *emitted*, which results in a function call to all *slots* that have been connected to the signal
  - i.e. *onSignal: slot()* in QML code

Signals and slots

- Code to support signal-slot connections is generated by the *moc* tool when project is compiled
- Special keywords are used, which are interpreted by *moc*
  - Q_OBJECT, signals, slots, emit

Special keywords

- Q_OBJECT keyword must be added to every class that inherits from QObject base class
  - Tells *moc* to parse the class contents
  - QtCreator complains if missing

```
class Emitter : public QObject {
public:
    void doSomething() { emit changed(); }
signals:
    void changed();
};
```

you forgot the Q_OBJECT macro

Special keywords

<symbio>

- *signals* keyword is used to start a block of signal definitions
  - Signal functions are not implemented. Instead, the code for them is generated by *moc*
  - Signals can have parameters as any normal function
    - A slot that is connected to signal must have matching parameter count and types

```
signals:
    void helloSignal();
    void signalWithParams(const QString &data, qint32 value);

public slots:
    void hello();
```

Special keywords

- *slots* keyword starts a block of slot definitions
  - Each slot is a normal C++ function
    - Can be called directly from code
  - Normal visibility rules apply when called directly from code
    - However, signal-slot connections will ignore visibility and thus it's possible to connect to private slot from anywhere

```
public slots:
    void publicSlot();

protected slots:
    void protectedSlot();

private slots:
    void internalSlot();
```

Special keywords

<symbio>

- *emit* keyword is used to send a notification to all slots that have been connected to the signal
    - Object framework code loops over the slots that have been connected to the signal and makes a regular function call to each

Connecting signals to slots

<symbio>

- Connections are made with *QObject::connect* static functions
  - No access control, anyone can connect anything
  - Class headers are not needed if signal and slot function signatures are known
- Component-based approach
  - Components provide services
  - Controller makes the connections between components

Signals and slots

- Comparing Qt and Java

```
class Emitter : public QObject {
    Q_OBJECT
public:
    void doSomething() { emit changed(); }
signals:
    void changed();
};

class Observer : public QObject {
    Q_OBJECT
public slots:
    void notifyChange() {
    }
};

class Manager : public QObject {
    Q_OBJECT
public:
    Manager() : emitter(new Emitter(this)),
                observer(new Observer(this)) {
    }
    void connectObjects() {
        QObject::connect(emitter, SIGNAL(changed()),
                         observer, SLOT(notifyChange()));
        emitter->doSomething();
    }
private:
    Emitter *emitter;
    Observer *observer;
};
```

```
import java.util.ArrayList;

interface ChangeEventListener {
    void notifyChange();
}

class Emitter {
    private ArrayList<ChangeEventListener> listeners =
        new ArrayList<ChangeEventListener>();
    void addChangeListener(ChangeEventListener listener) {
        listeners.add(listener);
    }
    void doSomething() { changed(); }
    private void changed() {
        for (int i = 0; i < listeners.size(); i++) {
            listeners.get(i).notifyChange();
        }
    }
}

class Observer implements ChangeEventListener {
    public void notifyChange() {
    }
}

public class Manager {
    private Emitter emitter = new Emitter();
    private Observer observer = new Observer();
    void connectObjects() {
        emitter.addChangeListener(observer);
        emitter.doSomething();
    }
}
```

Object properties

CORE FEATURES

Object properties

<symbio>

- All QObject-based classes support *properties*
  - A property is *QVariant* type, which is stored in a dictionary that uses C-style zero-terminated character arrays as keys
    - i.e. name-value pair
  - Properties can be *dynamic* or *static*
    - *Dynamic properties are assigned at run-time*
    - *Static properties are defined at compile time and processed by the meta-object compiler*

Object properties

- Static properties are declared into class header using *Q_PROPERTY* macro

```
class AnimatedPixmap : public QObject, public QGraphicsPixmapItem
{
    Q_OBJECT
    Q_PROPERTY(qreal rotation READ rotation WRITE setRotation NOTIFY rotationChanged)
```

- The above statement defines a property
  - Type is *qreal*, name is *rotation*
  - When read, *rotation* function is called
  - When modified, *setRotation* function is called
  - Changes are notified via *rotationChanged* signal

<symbio>

QML / C++ integration

HYBRID PROGRAMMING

Exporting objects to QML

<symbio>

- Objects are registered with *qmlRegisterType* template function
  - Object class as template parameter
  - Function parameters:
    - *Module name*
    - Object version number (major, minor)
    - Name that is registered to QML runtime

Details about modules from:
http://doc.trolltech.com/4.7-snapshot/qdeclarativemodules.html

```
qmlRegisterType<LocationNotifier>("LocationExample", 1, 0, "LocationNotifier");
```

Using exported classes

- The exported classes can be used as any other QML component
  - The module needs to be imported

```
qmlRegisterType<LocationNotifier>("LocationExample", 1, 0, "LocationNotifier");
```

```
class LocationNotifier : public QObject
{
    Q_OBJECT

    Q_PROPERTY(QString text READ text WRITE setText NOTIFY textChanged)
```

```
import Qt 4.7
import LocationExample 1.0

Rectangle {
    width: 200
    height: 200

    LocationNotifier {
        id: notifier
    }

    Text {
        x: 66; y: 93
        text: notifier.text
    }
}
```

Qt objects in QML

<symbio>

- Visibility at QML side
  - QObject *properties* become element properties
    - *on<Property>Changed* hook works if the NOTIFY signal is specified at C++ side
      - Also note that C++ signal name doesn't matter
  - QObject *signals* can be hooked with *on<Signal>*
  - QObject *slots* can be called as functions

Graphics items

<symbio>

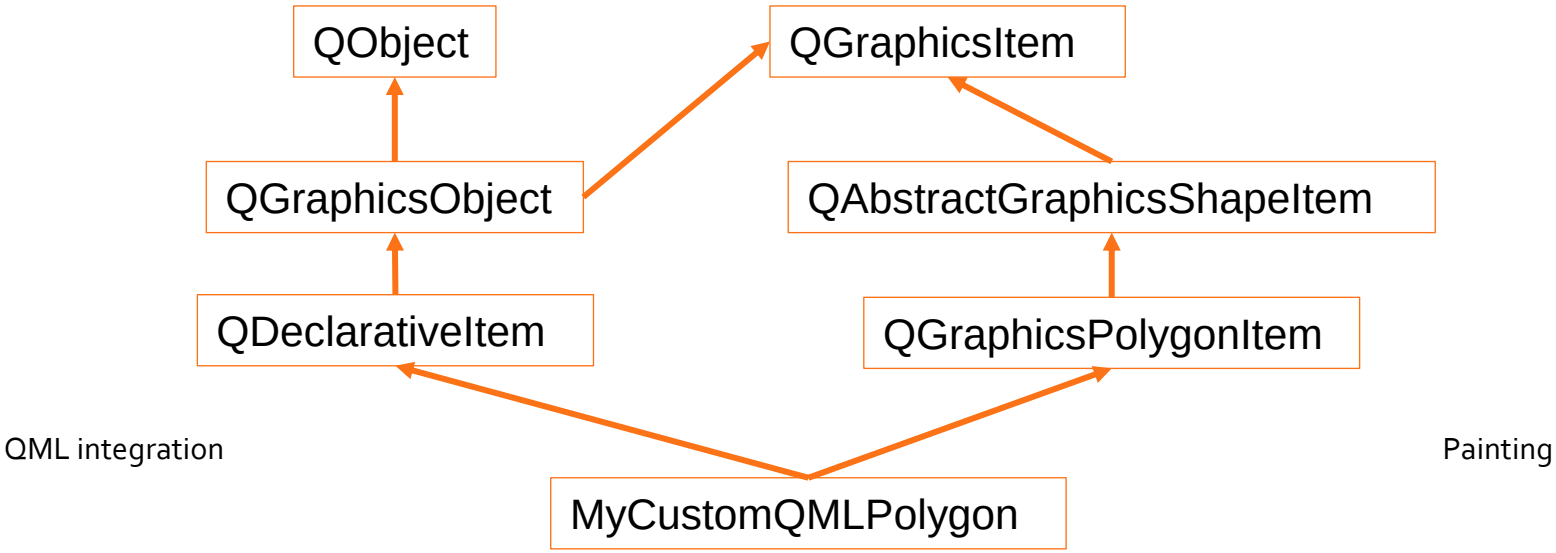
- Qt graphics items
  - *QGraphicsItem* base class
    - Not based on *QObject* for performance reasons
  - Items are added to *graphics scene*
- A QML *Item* is based on *QDeclarativeItem*
  - Inherits *QGraphicsObject* from *Qt graphics framework*

http://doc.trolltech.com/4.7-snapshot/qdeclarativeitem.html

  - Custom painting can be done on C++ side

Graphics items

- Multiple inheritance hierarchies



- Example in *HelloQMLApp* directory

<symbio>

Overview of QML animations

BUILDING FLUID GUI

Animations overview

<symbio>

- *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

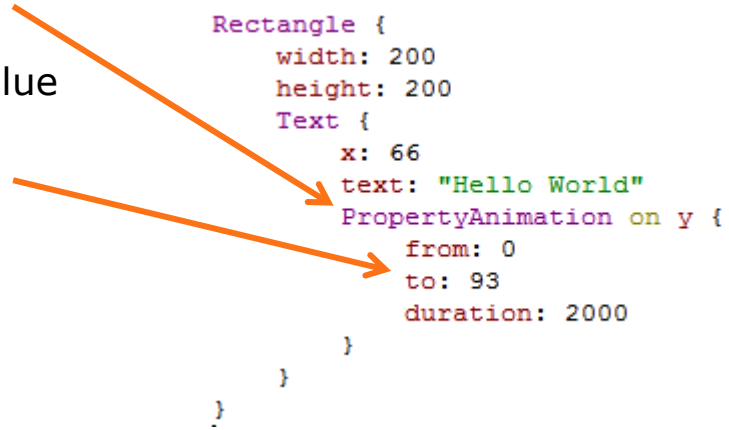
- All animations inherit from *Animation* base component
  - Basic properties (just like *Item* for GUI)
- Properties for declarative use:
  - *running*, *paused, loops, alwaysRunToEnd*
- Can also be used imperatively:
  - *start, stop, pause, resume, restart, complete*

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*

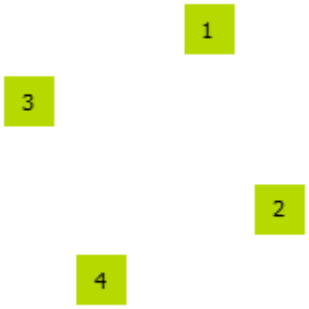
Animation grouping

- 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

<symbio>

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



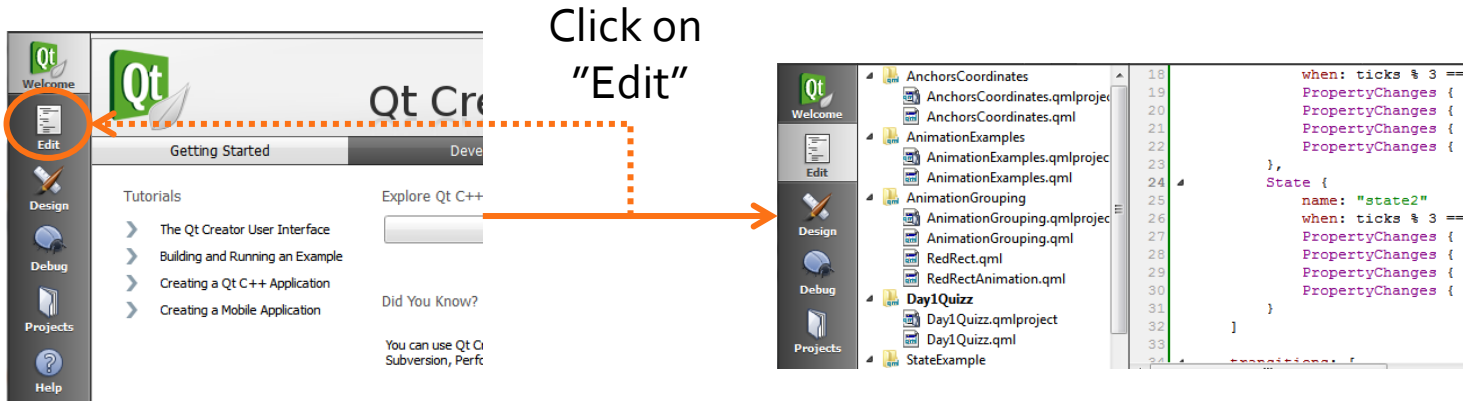
<symbio>

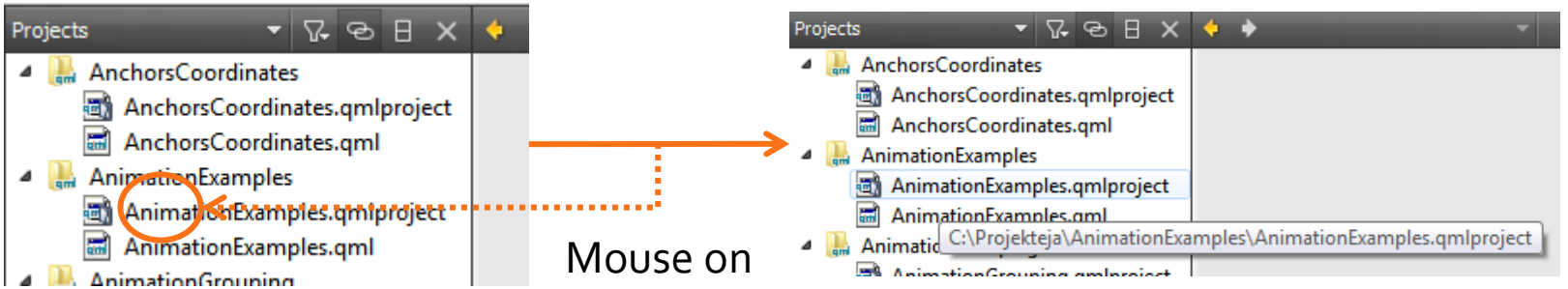
GUI states and animated state transitions

BUILDING FLUID GUI

GUI states

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





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

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

  - Each *State* object inherits properties from default state and declares the differences
    - *PropertyChanges* element

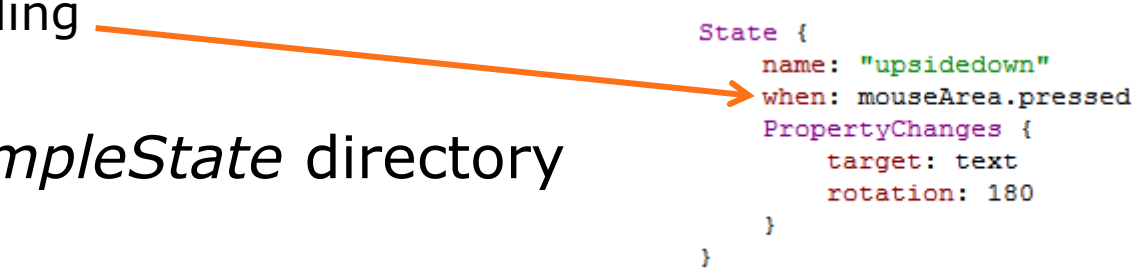
```
states: State {
    name: "shorter"
    PropertyChanges {
        target: rect;
        height: 100
    }
}
```

  - A state may inherit properties from another state instead of the default
    - *extend* property

```
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

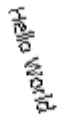


```
State {
    name: "upsidedown"
    when: mouseArea.pressed
    PropertyChanges {
        target: text
        rotation: 180
    }
}
```

- Example in *SimpleState* directory

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

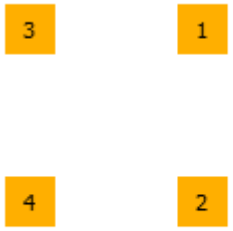


State transitions

- 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
- Mapping the *AnimationGrouping* example into state framework
  - *StateExample* 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

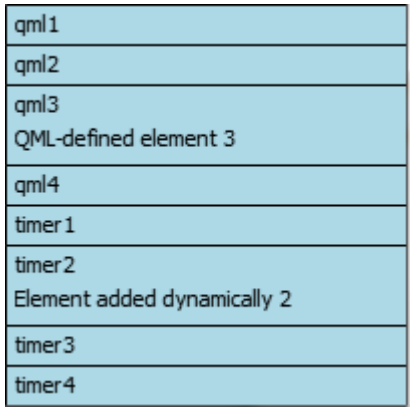
Data models

<symbio>

- *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



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

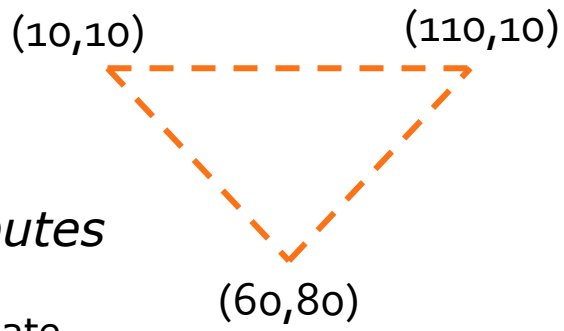
Data views

<symbio>

- 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 views

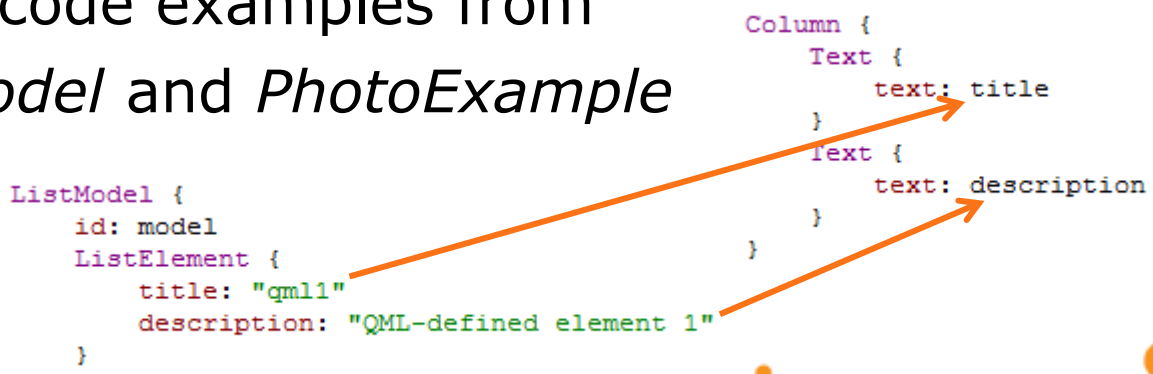
- Interaction with views
  - List and grid views inherint from *Flickable*
    - Content can be scrolled (no scrollbars though)
  - Path view uses drag and flick to move the items around the path
  - Delegates may implement mouse handlers
    - Same rules as with *Flickable* nested mouse areas

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
  - Cannot be used to store any state
    - Thus, state must be stored in the model

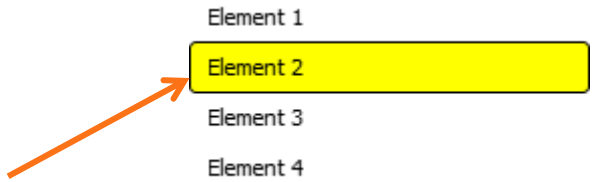
GUI delegates

- 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*



View selection

- Each view has *currentIndex* property
  - *ListView* and *GridView* also 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

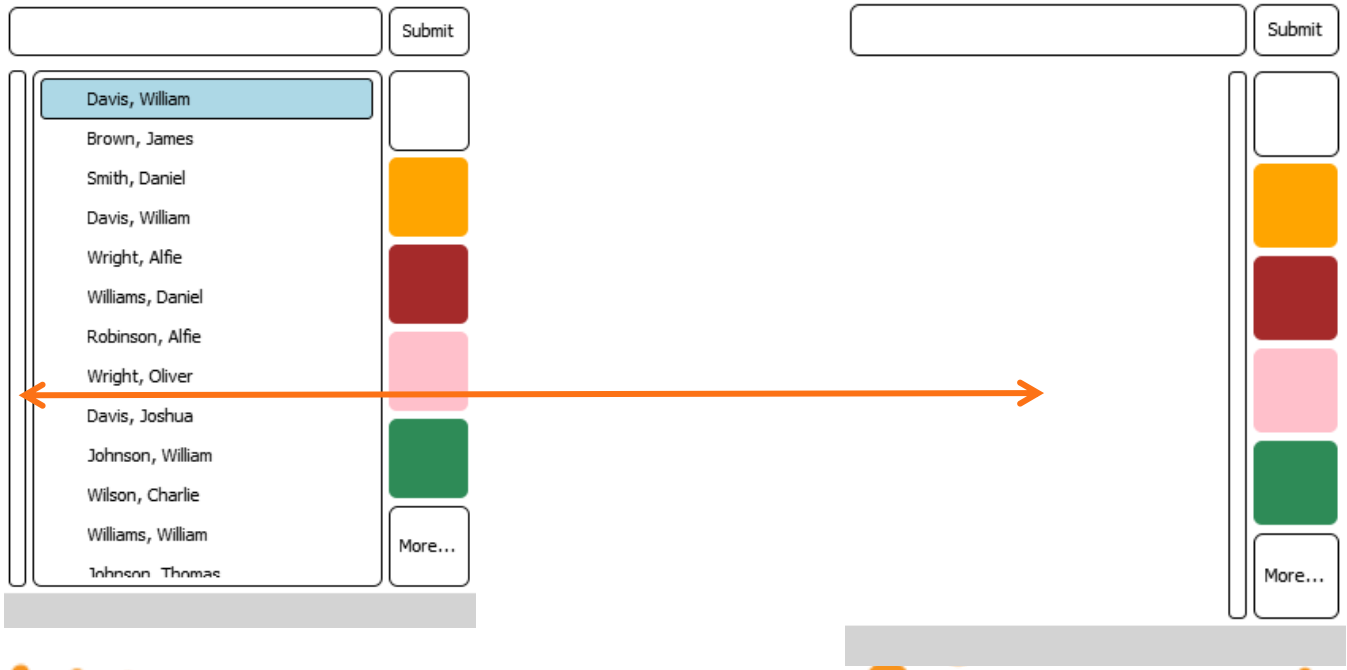
FLUID GUI EXERCISES

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



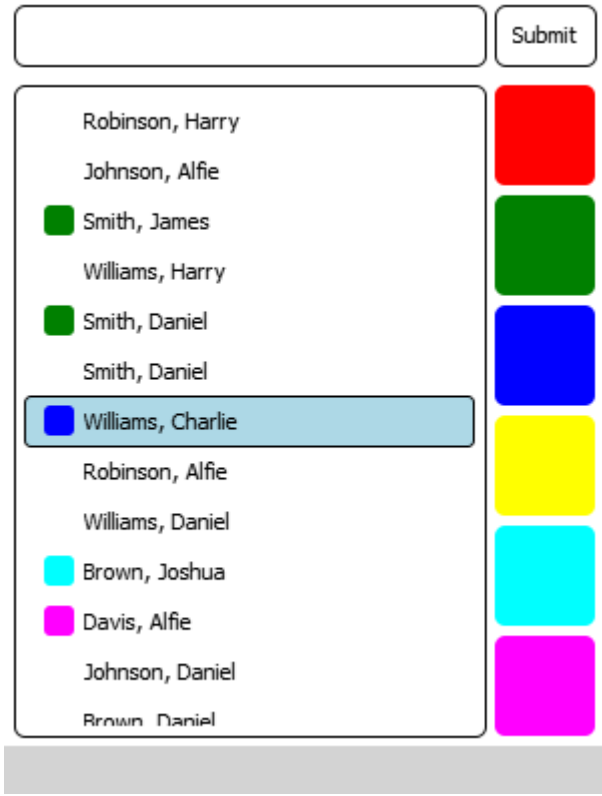
<symbio>

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




symbio
SERIOUS ABOUT SOFTWARE