Qml Presenting Data

Qt Essentials - Training Course

Produced by Nokia, Qt Development Frameworks

Material based on Qt 4.7, created on January 18, 2011

http://qt.nokia.com

Module: Presenting Data

- Arranging Items
- Data Models
- Using Views
- XML Models
- Views Revisited

Objectives

Can manipulate and present data:

- Familiarity with positioners and repeaters
    - rows, columns, grids, flows
    - item indexes
- Understanding of the relationship between models
    - pure models
    - visual models
    - XML models
- Ability to define and use list models
    - using pure models with repeaters and delegates
    - using visual models with repeaters
- Ability to use models with views
    - using list and grid views
    - decorating views
    - defining delegates

Module: Presenting Data

Arranging Items

Positioners and repeaters make it easier to work with many items

- Positioners arrange items in standard layouts
  - in a column: `Column`
  - in a row: `Row`
  - in a grid: `Grid`
  - like words on a page: `Flow`
- Repeaters create items from a template
  - for use with positioners
  - using data from a model
- Combining these make it easy to lay out lots of items

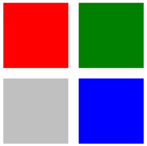
Positioning Items

```
import QtQuick 1.0

Grid {
    x: 15; y: 15; width: 300; height: 300

    columns: 2; rows: 2; spacing: 20

    Rectangle { width: 125; height: 125; color: "red" }
    Rectangle { width: 125; height: 125; color: "green" }
    Rectangle { width: 125; height: 125; color: "silver" }
    Rectangle { width: 125; height: 125; color: "blue" }
}
```



- Items inside a positioner are automatically arranged
  - in a 2 by 2 `Grid`
  - with horizontal/vertical spacing of 20 pixels
- `x`, `y` is the position of the first item
- Like layouts in Qt

Demo qml-presenting-data/ex-arranging-items/grid-rectangles.qml

Repeating Items

```
import QtQuick 1.0

Rectangle {
    width: 400; height: 400; color: "black"

    Grid {
        ...
        Repeater { ... }
    }
}
```

- The `Repeater` creates items
- The `Grid` arranges them within its parent item
- The outer `Rectangle` item provides
  - the space for generated items
  - a local coordinate system

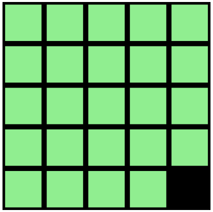
Repeating Items

```
import QtQuick 1.0

Rectangle {
    width: 400; height: 400; color: "black"

    Grid {
        x: 5; y: 5
        rows: 5; columns: 5; spacing: 10

        Repeater { model: 24
                   Rectangle { width: 70; height: 70
                               color: "lightgreen" } }
    }
}
```



- Repeater takes data from a model
  - just a number in this case
- Creates items based on the template item
  - a light green rectangle

Demo qml-presenting-data/ex-arranging-items/repeater-grid.qml

Indexing Items

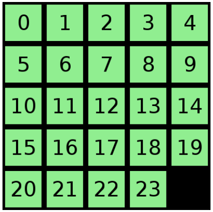
```
import QtQuick 1.0

Rectangle {
    width: 400; height: 400; color: "black"

    Grid {
        x: 5; y: 5
        rows: 5; columns: 5; spacing: 10

        Repeater { model: 24
                   Rectangle { width: 70; height: 70
                               color: "lightgreen"

                               Text { text: index
                                      font.pointSize: 30
                                      anchors.centerIn: parent } }
        } ...
```



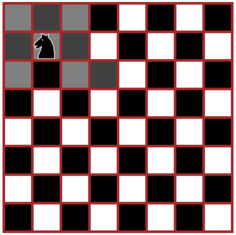
- `Repeater` provides an `index` for each item it creates

Demo qml-presenting-data/ex-arranging-items/repeater-grid-index.qml

Positioner Hints and Tips

- Anchors in the `Row`, `Column` or `Grid`
  - apply to all the items they contain

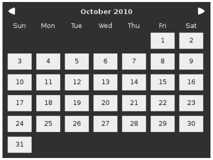
Lab – Chess Board



- Start by creating a chess board using a `Grid` and a `Repeater`
  - use the `index` to create a checker pattern
- Use the `knight.png` image to create a piece that can be placed on any square
  - bind its `x` and `y` properties to custom `cx` and `cy` properties
- Make each square clickable
  - move the piece when a suitable square is clicked
- Make the model an `Array` that records which squares have been visited
- Make the board and piece separate components

Lab – Calendar

- Start by creating a grid of squares using a `Grid` and a `Repeater`
  - put the grid inside an `Item`
  - use the `index` to give each square a number
- Place a title above the grid
- Ensure that the current date is highlighted
- Use the `left.png` and `right.png` images to create buttons on each side of the title
- Make the buttons navigate to the next and previous months
- Add a header showing the days of the week



Module: Presenting Data

- Arranging Items
- Data Models
- Using Views
- XML Models
- Views Revisited

Models and Views

Models and views provide a way to handle data sets

- Models hold data or items
- Views display data or items
    - using delegates

Models

Pure models provide access to data:

- `ListModel`
- `XmlListModel`

Visual models provide information about how to display data:

- Visual item model: `VisualItemModel`
  - contains child items that are supplied to views
- Visual data model: `VisualDataModel`
  - contains an interface to an underlying model
  - supplies a delegate for rendering

See Data Models Documentation

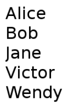
List Models

- List models contain simple sequences of elements
- Each `ListElement` contains
  - one or more pieces of data
  - defined using properties
  - *no information* about how to display itself
- `ListElement` does not have pre-defined properties
  - all properties are custom properties

```
ListModel {
    ListElement { ... }
    ListElement { ... }
    ...
}
```

Defining a List Model

```
ListModel {
    id: name_model
    ListElement { name: "Alice" }
    ListElement { name: "Bob" }
    ListElement { name: "Jane" }
    ListElement { name: "Victor" }
    ListElement { name: "Wendy" }
}
```



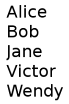
```
Alice
Bob
Jane
Victor
Wendy
```

- Define a `ListModel`
  - with an `id` so it can be referenced
- Define `ListElement` child objects
  - each with a `name` property
  - the property will be referenced by a delegate

Demo qml-presenting-data/ex-models-views/list-model-list-view.qml

Defining a Delegate

```
Component {
    id: name_delegate
    Text {
        text: name
        font.pixelSize: 32
    }
}
```

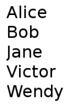


```
Alice
Bob
Jane
Victor
Wendy
```

- Define a `Component` to use as a delegate
  - with an `id` so it can be referenced
  - describes how the data will be displayed
- Properties of list elements can be referenced
  - use a `Text` item for each list element
  - use the value of the `name` property from each element
- In the item inside a `Component`
  - the `parent` property refers to the view
  - a `ListView` attached property can also be used to access the view

Using a List Model

```
Column {
    anchors.fill: parent
    Repeater {
        model: name_model
        delegate: name_delegate
    }
}
```



```
Alice
Bob
Jane
Victor
Wendy
```

- A `Repeater` fetches elements from `name_model`
  - using the delegate to display elements as `Text` items
- A `Column` arranges them vertically
  - using anchors to make room for the items

Working with Items

- ListModel is a dynamic list of items
- Items can be appended, inserted, removed and moved
  - **append** item data using JavaScript dictionaries:

    ```
    bookmarkModel.append({"title": lineEdit.text})
    ```

  - **remove** items by index obtained from a `ListView`:

    ```
    bookmarkModel.remove(listView.currentIndex)
    ```

  - **move** a number of items between two indices:

    ```
    bookmarkModel.move(listView.currentIndex,
                       listView.currentIndex + 1, number)
    ```

List Model Hints

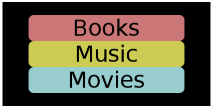
- **Note:** Model properties cannot shadow delegate properties:

```
ListModel {
    ListElement { text: "Alice" }
}

Component {
    Text {
        text: text // will not work
    }
}
```

Defining a Visual Item Model

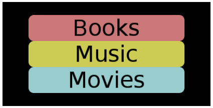
```
VisualItemModel {
    id: labels
    Rectangle { color: "#cc7777"; radius: 10.0
                width: 300; height: 50
                Text { anchors.fill: parent
                       font.pointSize: 32; text: "Books"
                       horizontalAlignment: Qt.AlignHCenter } }
    Rectangle { color: "#cccc55"; radius: 10.0
                width: 300; height: 50
                Text { anchors.fill: parent
                       font.pointSize: 32; text: "Music"
                       horizontalAlignment: Qt.AlignHCenter } }
    ...
}
```



- Define a `VisualItemModel` item
  - with an `id` so it can be referenced

Defining a Visual Item Model

```
VisualItemModel {
    id: labels
    Rectangle { color: "#cc7777"; radius: 10.0
                width: 300; height: 50
                Text { anchors.fill: parent
                       font.pointSize: 32; text: "Books"
                       horizontalAlignment: Qt.AlignHCenter } }
    Rectangle { color: "#cccc55"; radius: 10.0
                width: 300; height: 50
                Text { anchors.fill: parent
                       font.pointSize: 32; text: "Music"
                       horizontalAlignment: Qt.AlignHCenter } }
    ...
}
```



- Define child items
  - these will be shown when required

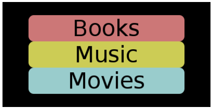
Using a Visual Item Model

```
import QtQuick 1.0

Rectangle {
    width: 400; height: 200; color: "black"

    VisualItemModel {
        id: labels
        ...
    }

    Column {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        Repeater { model: labels }
    }
}
```



- A `Repeater` fetches items from the `labels` model
- A `Column` arranges them vertically

Module: Presenting Data

Views

- `ListView` shows a classic list of items
  - with horizontal or vertical placing of items
- `GridView` displays items in a grid
  - like an file manager's icon view

List Views

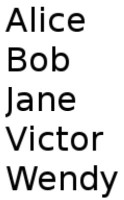
Take the model and delegate from before:

```
ListModel {
    id: nameModel
    ListElement { name: "Alice" }
    ListElement { name: "Bob" }
    ListElement { name: "Jane" }
    ListElement { name: "Victor" }
    ListElement { name: "Wendy" }
}
```

```
Component {
    id: nameDelegate
    Text {
        text: name;
        font.pixelSize: 32
    }
}
```

List Views

```
ListView {
    anchors.fill: parent
    model: nameModel
    delegate: nameDelegate
    clip: true
}
```


Alice
Bob
Jane
Victor
Wendy

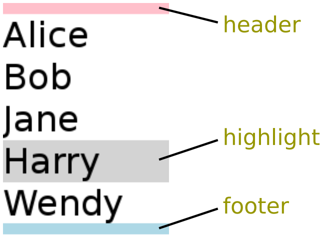
- Like Qt views
  - you must set a model
- Unlike Qt views
  - there is no default delegate
  - you must create and set a delegate or nothing will be displayed
- Unclipped views paint outside their areas
  - set the `clip` property to enable clipping
- Views are positioned like other items
  - the above view fills its parent

Demo qml-presenting-data/ex-models-views/list-model-list-view.qml

Decoration and Navigation

By default, `ListView` is

- undecorated
- a flickable surface
  - (can be dragged and flicked)

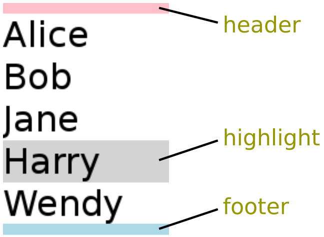


- To add decoration:
  - with a `header` and `footer`
  - with a `highlight` item to show the current item
- To configure for navigation:
  - set `focus` to allow keyboard navigation
  - `highlight` also helps the user with navigation
  - unset `interactive` to disable dragging and flicking

Demo qml-presenting-data/ex-models-views/list-view-decoration.qml

Decoration and Navigation

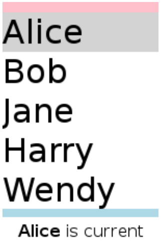
```
ListView {
   anchors.fill: parent
   model: nameModel
   delegate: nameDelegate
   focus: true
   clip: true
   header: Rectangle {
      width: parent.width; height: 10
      color: "pink"
   }
   footer: Rectangle {
      width: parent.width; height: 10
      color: "lightblue"
   }
   highlight: Rectangle {
      width: parent.width; color: "lightgray"
   }
}
```



Decoration and Navigation

Each ListView exposes its current item:

```
ListView {
    id: listView
    ...
}
Text {
    id: label
    anchors.bottom: parent.bottom
    anchors.horizontalCenter: parent.horizontalCenter
    text: "<b>" + listView.currentItem.text +
          "</b> is current"
    font.pixelSize: 16
}
```

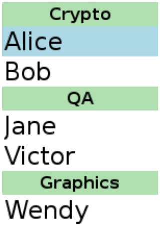


- Recall that, in this case, each item has a text property
  - re-use the listView's currentItem's text

Demo qml-presenting-data/ex-models-views/list-view-current-item.qml

Adding Sections

- Data in a `ListView` can be ordered by section
- Categorize the list items by
  - choosing a property name; e.g. `team`
  - adding this property to each `ListElement`
  - storing the section in this property



```
ListModel {
    id: nameModel
    ListElement { name: "Alice"; team: "Crypto" }
    ListElement { name: "Bob"; team: "Crypto" }
    ListElement { name: "Jane"; team: "QA" }
    ListElement { name: "Victor"; team: "QA" }
    ListElement { name: "Wendy"; team: "Graphics" }
}
```

Displaying Sections

Using the `ListView`

- Set `section.property`
  - refer to the `ListElement` property holding the section name
- Set `section.criteria` to control what to show
  - `ViewSection.FullString` for complete section names
  - `ViewSection.FirstCharacter` for alphabetical groupings
- Set `section.delegate`
  - create a delegate for section headings
  - either include it inline or reference it

Displaying Sections

```
ListView {
    model: nameModel
    ...
    section.property: "team"
    section.criteria: ViewSection.FullString
    section.delegate: Rectangle {
        color: "#b0dfb0"
        width: parent.width
        height: childrenRect.height + 4
            Text { anchors.centerIn: parent
                   font.pixelSize: 16
                   font.bold: true
                   text: section }
        }
    }
}
```

- The `section.delegate` is defined like the `highlight` delegate

Grid Views

Set up a list model with items:

```
ListModel {
    id: nameModel
    ListElement { file: "../images/rocket.svg"
                  name: "rocket" }
    ListElement { file: "../images/clear.svg"
                  name: "clear" }
    ListElement { file: "../images/arrow.svg"
                  name: "arrow" }
    ListElement { file: "../images/book.svg"
                  name: "book" }
}
```

- Define string properties to use in the delegate

Demo qml-presenting-data/ex-models-views/list-model-grid-view.qml

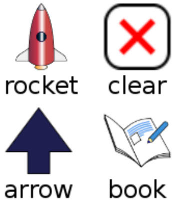
Grid Views

Set up a delegate:

```
Component {
   id: nameDelegate
   Column {
      Image {
         id: delegateImage
         anchors.horizontalCenter: delegateText.horizontalCenter
         source: file; width: 64; height: 64; smooth: true
         fillMode: Image.PreserveAspectFit
      }
      Text {
         id: delegateText
         text: name; font.pixelSize: 24
      }
   }
}
```

Grid Views

```
GridView {
    anchors.fill: parent
    model: nameModel
    delegate: nameDelegate
    clip: true
}
```



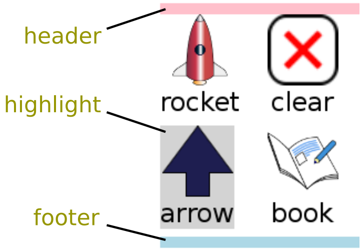
- The same as `ListView` to set up
- Uses data from a list model
  - not like Qt's table view
  - more like Qt's list view in icon mode

Decoration and Navigation

Like `ListView`, `GridView` is

- undecorated
- a flickable surface
    - (can be dragged and flicked)

- To add decoration:
    - define `header` and `footer`
    - define `highlight` item to show the current item

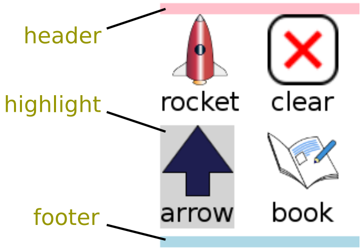


- To configure for navigation:
    - set `focus` to allow keyboard navigation
    - `highlight` also helps the user with navigation
    - unset `interactive` to disable dragging and flicking

Demo qml-presenting-data/ex-models-views/grid-view-decoration.qml

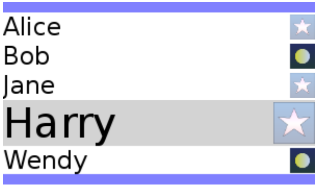
Decoration and Navigation

```
GridView {
    ...
    header: Rectangle {
        width: parent.width
        height: 10
        color: "pink"
    }
    footer: Rectangle {
        width: parent.width
        height: 10
        color: "lightblue"
    }
    highlight: Rectangle {
        width: parent.width
        color: "lightgray"
    }
    focus: true; clip: true
}
```



Lab – Contacts

- Create a `ListItemModel`, fill it with `ListElement` elements, each with
  - a `name` property
  - a `file` property referring to an image
- Add a `ListView` and a `Component` to use as a delegate



- Add `header`, `footer` and `highlight` properties to the view
- Add `states` and `transitions` to the delegate
  - activate the state when the delegate item is current
  - use a state condition with the `ListView.isCurrentItem` attached property
  - make a transition that animates the height of the item

Module: Presenting Data

XML List Models

- Many data sources provide data in XML formats
- `XmlListModel` is used to supply XML data to views
  - using a mechanism that maps data to properties
  - using XPath queries
- Views and delegates do not need to know about XML
  - use a `ListView` or `Repeater` to access data

Defining an XML List Model

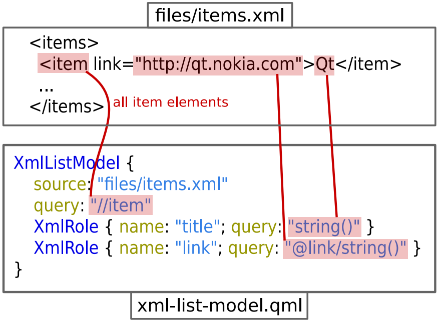
```
XmlListModel {
    id: xmlModel
    source: "files/items.xml"
    query: "//item"

    XmlRole { name: "title"; query: "string()" }
    XmlRole { name: "link"; query: "@link/string()" }
}
```

- Set the `id` property so the model can be referenced
- Specify the `source` of the XML
- The `query` identifies pieces of data in the model
- Each piece of data is queried by `XmlRole` elements

Demo qml-presenting-data/ex-models-views/xml-list-model.qml

XML Roles


files/items.xml

```
<items>
  <item link="http://qt.nokia.com">Qt</item>
  ...
</items>
```

```
XmlListModel {
    source: "files/items.xml"
    query: "//item"
    XmlRole { name: "title"; query: "string()" }
    XmlRole { name: "link"; query: "@link/string()" }
}
```

xml-list-model.qml

Result
title: "Qt"
link: "http://qt.nokia.com"

- `XmlRole` associates names with data obtained using XPath queries
- Made available to delegates as properties
  - `title` and `link` in the above example

Using an XML List Model

```
TitleDelegate {
    id: xmlDelegate
}

ListView {
    anchors.fill: parent
    anchors.margins: 4
    model: xmlModel
    delegate: xmlDelegate
}
```

- Specify the `model` and `delegate` as usual
- Ensure that the view is positioned and given a size
- `TitleDelegate` element is defined in `TitleDelegate.qml`
  - Must be defined using a `Component` element

Demo qml-presenting-data/ex-models-views/TitleDelegate.qml

Defining a Delegate

```
Component {
   Item {
      width: parent.width; height: 64

      Rectangle {
         width: Math.max(childrenRect.width + 16, parent.width)
         height: 60; clip: true
         color: "#505060"; border.color: "#8080b0"; radius: 8

         Column { Text { x: 6; color: "white"
                         font.pixelSize: 32; text: title }
                  Text { x: 6; color: "white"
                         font.pixelSize: 16; text: link } }
      }
   }
}
```

- `parent` refers to the view where it is used
- `title` and `link` are properties exported by the model

Module: Presenting Data

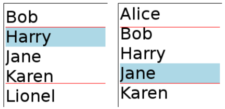
- Arranging Items
- Data Models
- Using Views
- XML Models
- Views Revisited

Customizing Views

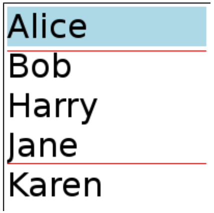
- All views are based on the `Flickable` item
- Key navigation of the highlighted item does not wrap around
  - set `keyNavigationWraps` to `true` to change this behavior
- The highlight can be constrained
  - set the `highlightRangeMode` property
  - `ListView.ApplyRange`
    tries to keep the highlight in a given area
  - `ListView.StrictlyEnforceRange`
    keeps the highlight stationary, moves the items around it

Customizing Views

```
ListView {
    ...
    preferredHighlightBegin: 42
    preferredHighlightEnd: 150
    highlightRangeMode: ListView.ApplyRange
    ...
}
```



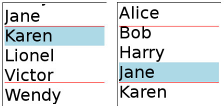
- View tries to keep the highlight within range
- Highlight may leave the range to cover end items
- `preferredHighlightBegin` and `preferredHighlightEnd` should
  - hold coordinates within the view
  - differ by the height/width of an item or more



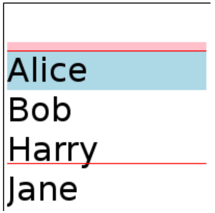
Demo qml-presenting-data/ex-models-views/list-view-highlight-range-apply.qml

Customizing Views

```
ListView {
    ...
    preferredHighlightBegin: 42
    preferredHighlightEnd: 150
    highlightRangeMode: ListView.StrictlyEnforceRange
    ...
}
```



- View always keeps the highlight within range
- View may scroll past its end to keep the highlight in range
- `preferredHighlightBegin` and `preferredHighlightEnd` should
  - hold coordinates within the view
  - differ by the height/width of an item or more



Demo qml-presenting-data/ex-models-views/list-view-highlight-range-strict.qml

Optimizing Views

- Views create delegates to display data
  - delegates are only created when they are needed
  - delegates are destroyed when no longer visible
  - this can impact performance
- Delegates can be cached to improvement performance
  - `cacheBuffer` is the maximum number of delegates to keep
  - trades memory usage for performance
  - useful if it is expensive to create delegates; for example
    - when obtaining data over a network
    - when delegates require complex rendering