Qt Quick for Qt Developers

Presenting Data

The Qt Company

Based on Qt 5.4 (QtQuick 2.4)

Contents

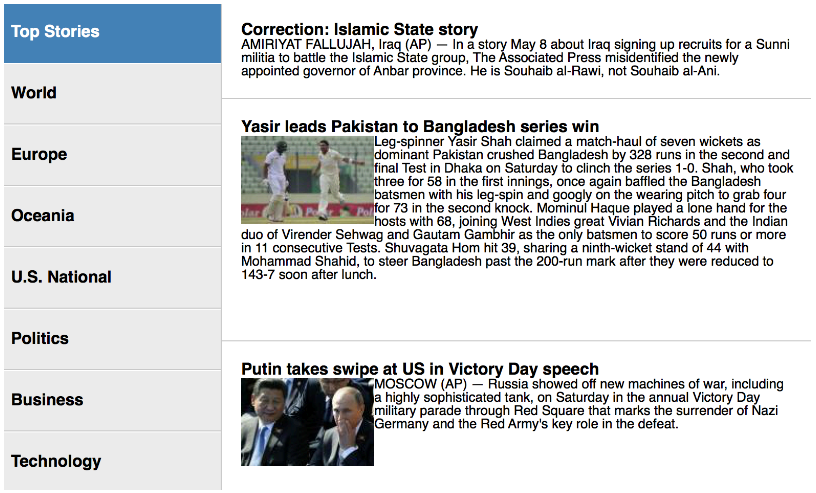
Objectives

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

Why Use Model/view Separation?

- Easily change the UI later
- Add an alternative UI
- Separation of concerns
- Leads to easier maintenance
- Easily change the data source
  - (XML? JSON? Other?)
- Allows the use of 'dummy' data during development
- Many Qt APIs to consume the common data structures



Correction: Islamic State story
AMIRIYAT FALLUJAH, Iraq (AP) — In a story May 8 about Iraq signing up recruits for a Sunni militia to battle the Islamic State group, The Associated Press misidentified the newly appointed governor of Anbar province. He is Souhaib al-Rawi, not Souhaib al-Ani.

Yasir leads Pakistan to Bangladesh series win
Leg-spinner Yasir Shah claimed a match-haul of seven wickets as dominant Pakistan crushed Bangladesh by 328 runs in the second and final Test in Dhaka on Saturday to clinch the series 1-0. Shah, who took three for 58 in the first innings, once again baffled the Bangladesh batsmen with his leg-spin and googly on the wearing pitch to grab four for 73 in the second knock. Mominul Haque played a lone hand for the hosts with 68, joining West Indies great Vivian Richards and the Indian duo of Virender Sehwag and Gautam Gambhir as the only batsmen to score 50 runs or more in 11 consecutive Tests. Shuvagata Hom hit 39, sharing a ninth-wicket stand of 44 with Mohammad Shahid, to steer Bangladesh past the 200-run mark after they were reduced to 143-7 soon after lunch.

Putin takes swipe at US in Victory Day speech
MOSCOW (AP) — Russia showed off new machines of war, including a highly sophisticated tank, on Saturday in the annual Victory Day military parade through Red Square that marks the surrender of Nazi Germany and the Red Army's key role in the defeat.

Demo: <Qt Examples>/declarative/demos/rssnews/rssnews.pro

Arranging Items

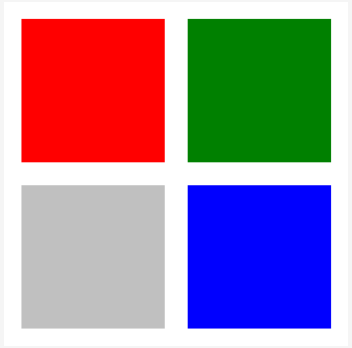
Arranging Items

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 layout lots of items

Positioning Items

```
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
  - Ina 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

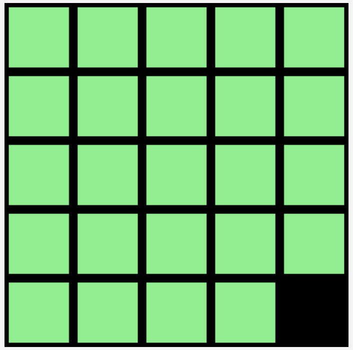
Repeating Items

```
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" }
        }
    }
}
```

- 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

```
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" }
        }
    }
}
```

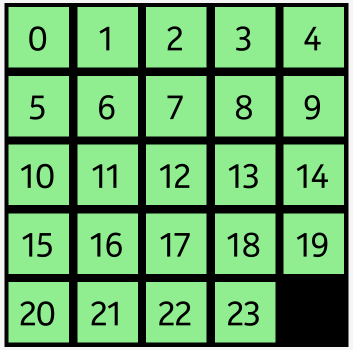


- `Repeater` takes data from a model
  - Just a number in this case
- Creates items based on the template item
  - A light green rectangle

Demo: qml-presenting-data/ex-arranging-items/repeater-gird.qml

Indexing Items

```
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

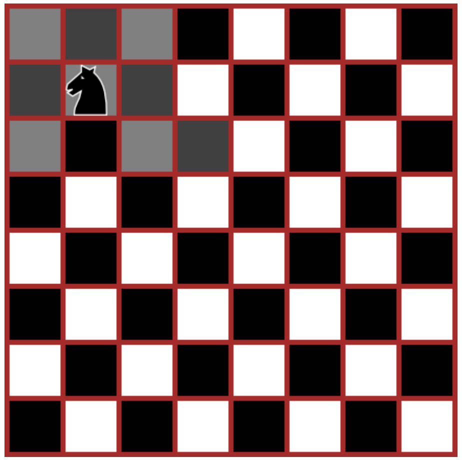
Demo: qml-presenting-data/ex-arranging-items/repeater-gird-index.qml

Positioner Hints and Tips

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

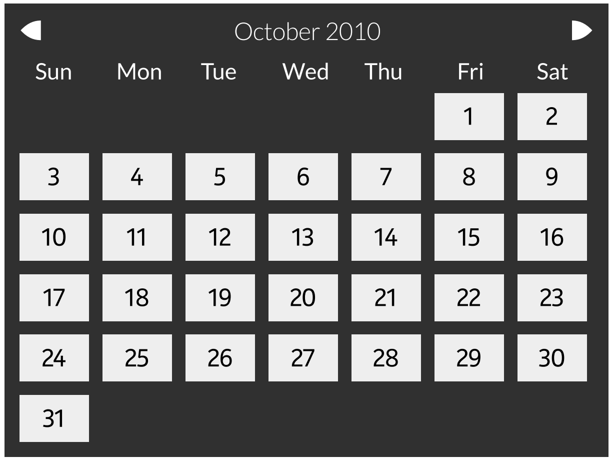
Lab – Chess Board

- 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 chess board 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


October 2010

Sun	Mon	Tue	Wed	Thu	Fri	Sat
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

Data Models

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: `ObjectModel` (replaces `VisualItemModel`)
  - Contains child items that are supplied to views
- Visual data model: `DelegateModel` (replaces `VisualDataModel`)
  - Contains an interface to an underlying model
  - Supplies a delegate for rendering
  - Supports delegate sharing between the views

See Documentation: Data Models

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 {
    id: nameModel
    ListElement { … }
    ListElement { … }
    ListElement { … }
}
```

Defining a List Model

```
ListModel {
    id: nameModel
    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: nameDelegate
    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: nameModel
        delegate: nameDelegate
    }
}
```

```
Alice
Bob
Jane
Victor
Wendy
```

- A `Repeater` fetches elements from `nameModel`
  - 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

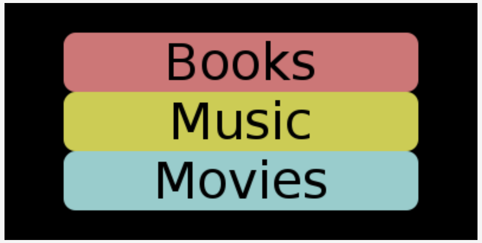
- **Note:** Model properties cannot shadow delegate properties:

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

Component {
    Text {
        text: text; // Will not work
    }
}
```

Defining an Object Model (Visual Item Model)

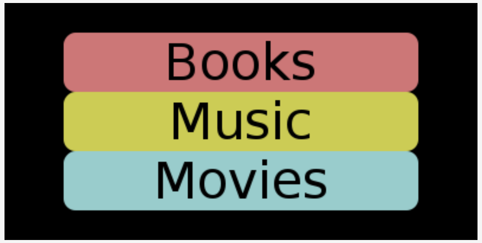
```
Rectangle {
    width: 400; height: 200; color: "black"
    ObjectModel {
        id: labels
        Rectangle { color: "#cc7777"; radius: 10.0
                    width: 300; height: 50
                    Text { anchors.fill: parent
                           font.pointSize: 32; text: "Books"
                           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 `ObjectModel` item
  - With an `id` so it can be referenced
  - Import `QtQml.Models 2.1`

Defining an Object Model (Visual Item Model)

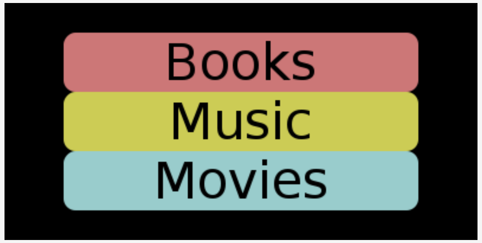
```
Rectangle {
    width: 400; height: 200; color: "black"
    ObjectModel {
        id: labels
        Rectangle { color: "#cc7777"; radius: 10.0
                    width: 300; height: 50
                    Text { anchors.fill: parent
                           font.pointSize: 32; text: "Books"
                           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 an Object Model (Visual Item Model)

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

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
}
```

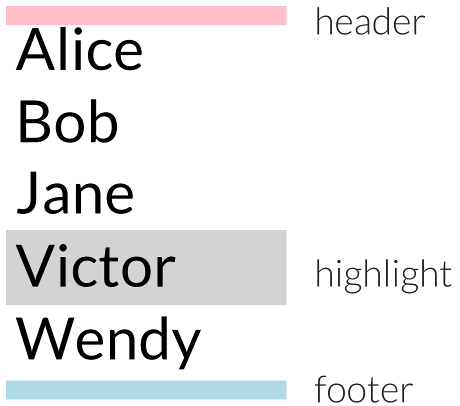
```
Alice
Bob
Jane
Victor
Wendy
```

- No default delegate
- 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

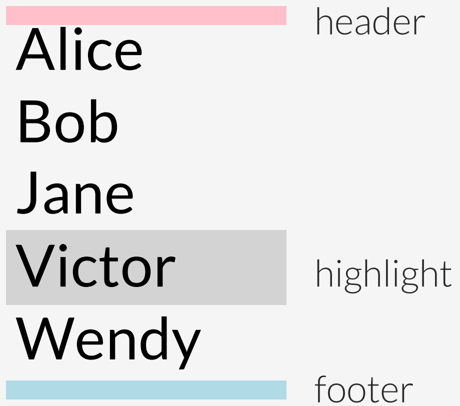
- By default, `ListView` is
  - Undecorated
  - A flickable surface (can be dragged and flicked)
- To add decoration:
  - With a `header` and `footer`
  - With a `highlight` item to show the current item
- To configure for navigation:
  - Set `focus` to allow keyboard navigation
  - Property `highlight` 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

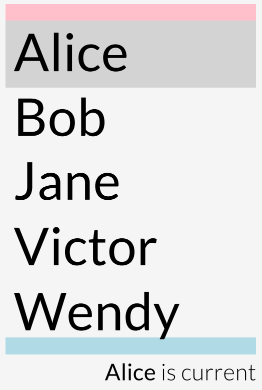
```
ListView {
    anchors.fill: parent
    model: nameModel
    delegate: nameDelegate
    focus: true
    clip: true
    header: Rectangle {
        width: parent.width; height: 10;
       color: "pink” }
    footer: Rectangle {
        width: parent.width; height: 10;
        color: "lightblue" }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



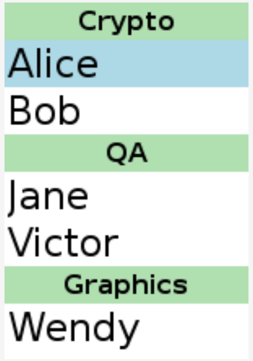
- Recall that, in this case, each item has a `text` property
  - re-use the listView's `currentItem's text`

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 name
  - `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.horizontalCenter: parent.horizontalCenter
               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

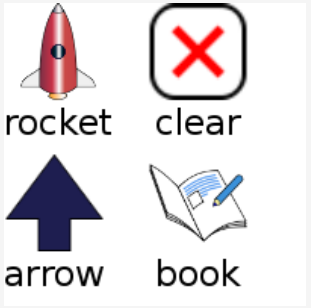
Grid Views

- 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
}
```

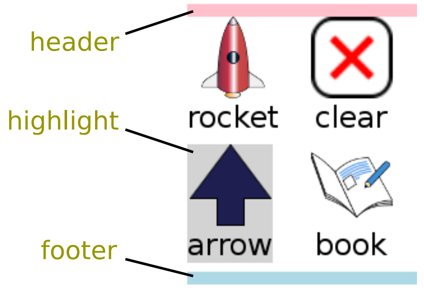


- The same as `ListView` to set up
- Uses data from a list model
  - Not like Qt's table view
  - More like Qt's list view in icon mode

Decoration and Navigation

Like `ListView`, `GridView` is

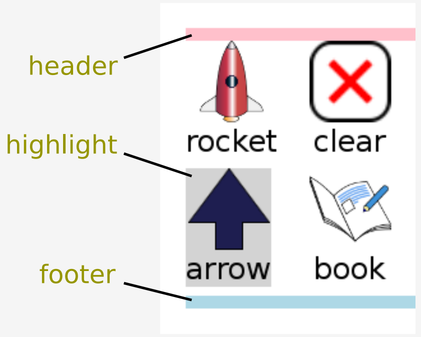
- Undecorated and a flickable surface
- To add decoration:
  - Define header and footer
  - Define highlight item to show the current item
- To configure for navigation:
  - Set focus to allow keyboard navigation
  - Highlight also helps the user with navigation
  - Unset interactive to disable dragging and flicking



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

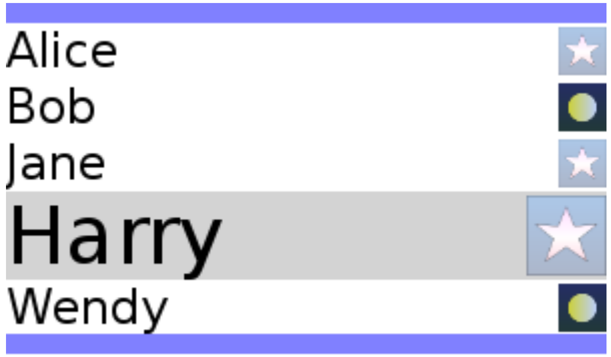
Decoration and Navigation

```
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



XML Models

XML List Models

- Many data sources provide data in XML formats
- Element `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

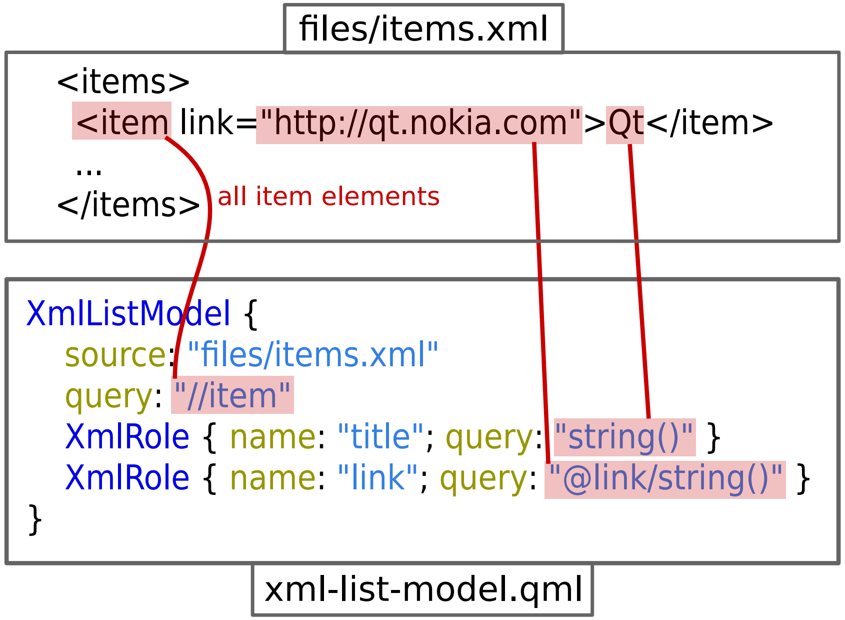
Defining an XML List Model

```
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



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

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

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

- Element `XmlRole` associates names with data obtained using XPath queries
- Made available to delegates as properties
  - 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
- Element `TitleDelegate` 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 }
            }
        }
    }
}
```

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

Views Revisited

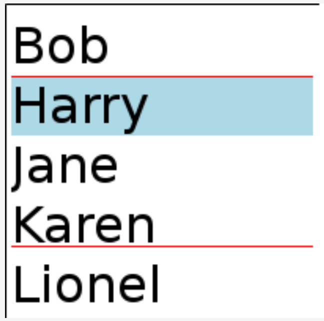
Customizing Views

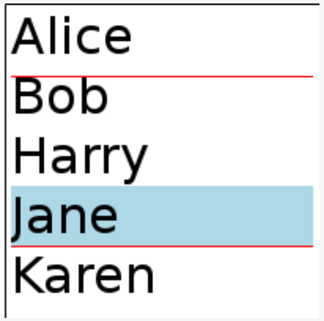
- All views are based on the `Flickable` item
- Key navigation of the highlighted item does not wrap around
  - Set `keyNavigationWraps` to true to change this behavior
- The highlight can be constrained
  - Set the `highlightRangeMode` property
  - Value `ListView.ApplyRange` tries to keep the highlight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

Customizing Views

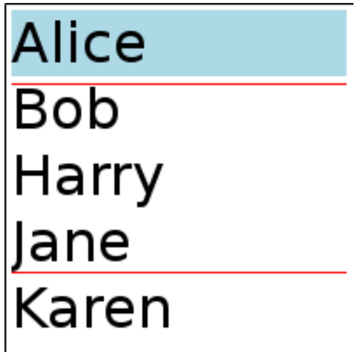
```
ListView {
    preferredHighlightBegin: 42
    preferredHighlightEnd: 150
    highlightRangeMode: ListView.ApplyRange

    …
}
```





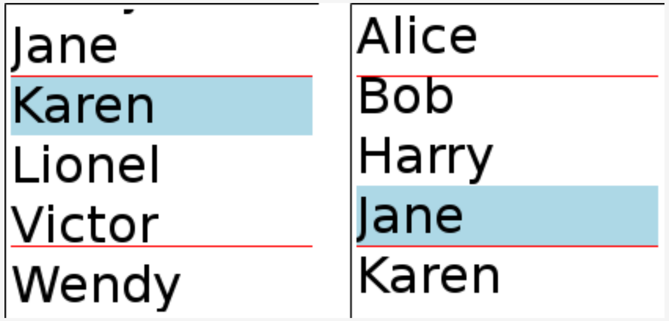
- View tries to keep the highlight within range
- Highlight may leave the range to cover end items
- Properties `preferredHighlightBegin` and `preferredHighlightEnd` should
  - Hold coordinates within the view
  - Differ by the height/width of an item or more



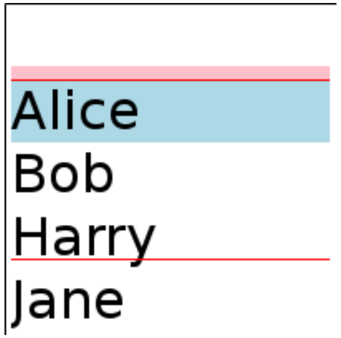
Demo: qml-presenting-data/ex-models-views/list-view-highlight-range-apply.qml

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
- Properties `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 improve performance
  - Property `cacheBuffer` is the maximum number of delegates to keep (calculated as a multiply of the height of the delegate)
  - 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