# QML Programming Fundamentals and Beyond

## QML Models and Views

*Material based on Qt 5.12*

# Course Outline

Session 1: April 28, Introduction to QML
- About QML
- Properties
- Basic Types

Session 2: May 1, QML Item Placement
- How to correctly size and place items
- When to use Anchors, Layouts and Positioners

Session 3: May 5, Touch Interaction
- QML Signals
- Touch Events
- Single and Multi-Touch
- Swipe and Pinch Gestures

Session 4: May 8, States & Transitions
- Creating and defining states
- Using Transitions

Session 5: May 15, Custom Items & Components
- Creating your own Components
- Creating a Module

Session 6: May 19, Model / View
- Model / View
- QML Models
- QML Views

Session 7: May 22, C++ Integration
- Why expose C++ to QML
- Exposing C++ Objects
- Exposing C++ Classes

## About ICS

### *ICS Designs User Experiences and Develops Software for Connected Devices*

- Largest source of independent Qt expertise in North America since 2002
- Headquartered in Waltham, MA with offices in California, Canada, Europe
- Includes Boston UX, ICS' UX design division

- Embedded, touchscreen, mobile and desktop applications
- Exclusive Open Enrollment Training Partner in North America

Qt Service Partner

BOSTON UX

# UX/UI Design and Development for Connected Devices Across Many Industries

## Agenda

- Model / View
- Structuring models
- View types available

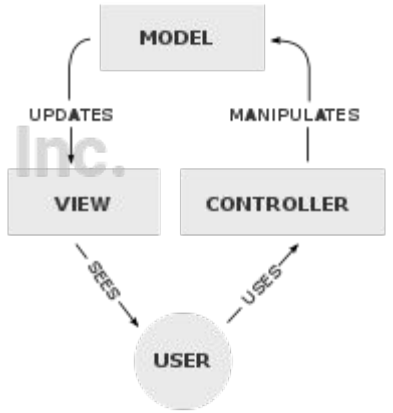
# What is Model / View?

Qt Model / View in a nutshell

- A model provides data for a view
- A view displays the data from the model
- Similar to the Model / View / Controller design pattern
  - But combine the controller with the view

Why Model / View?

- Isolate the business logic from the UI logic
- Create UI components that are independent
  - Easier Development
  - Easier Testing
  - Easier Maintenance
- More Reuse
  - Reuse your UI logic across multiple views
  - Models can be reused for different views



# Model / View Components

Model:

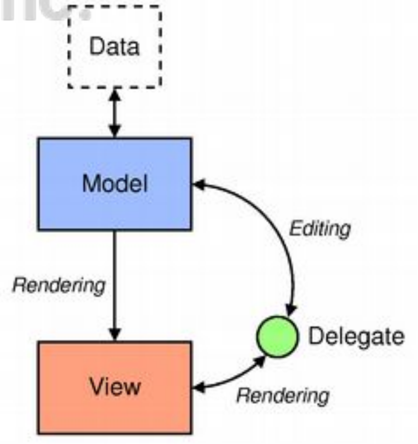
- Interface to business logic data
- Emits signals when data changes or is appended
- Many common model types are provided by Qt
    - C++: `QStringList`, `QVariantList`, `QAbstractItemModel` subclass
    - QML: an integer, JavaScript array, `ListModel`, `XmlListModel`

View:

- Displays the data structure
- Handles the user interaction with the data
- Qt provides many view types with model support
    - QML: `ListView`, `GridView`, `PathView`, `TableView` (new)
- Model items are accessible through the index property

Delegate:

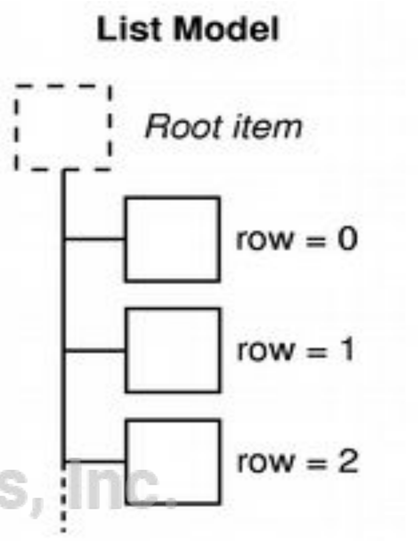
- Items in a view are rendered and edited by delegates
- The view expects a `Component` for its delegate property



# Model Types

- Standard QML model
  - Integer - read only, the model has no data roles
  - Object instances - properties of the object are the roles
  - JavaScript array - read only, no need for dynamic model updates
  - `ListModel` - simple hierarchy of QML types
- Additional QML models available in other modules
  - `QtLocation` - `CategoryModel`, `EditorialModel`, `ImageModel`, etc.
  - `QtDataVisulization` - `ItemModelBarDataProxy`, etc.
  - And More from `QtBluetooth`, `QtCharts`, `QtLabs`, `QtWebEngine`
- C++ models (must be implemented in C++ before use)
  - More complex *but* more fine-tuned control over data access
  - `QStringList`, `QVariantList`, `QObjectList`, `QAbstractItemModel` (and subclasses)
  - Subclass `QAbstractItemModel` for complex data-sets

# ListModel: Definition



A **ListModel** defines a free-form list data source

- A simple container of **ListElement** definitions
- Define a **ListModel**
    - With an **id** so that it can be referenced
- Define **ListElement** child objects
    - Each with a custom **name** data role
    - The property will be referenced by a delegate

```
ListModel {
    id: people
    ListElement { name: "Alice"; address: "Alice Address" }
    ListElement { name: "Bob" ; address: "Bob Address"}
    ListElement { name: "Jane" ; address: "Jane Address"}
    ListElement { name: "Victor" ; address: "Victor Address"}
    ListElement { name: "Wendy" ; address: "Wendy Address"}
}
```

# ListModel: How To Use

```
Column {
    anchors.fill: parent
    Repeater {
        model: people
        delegate: Text{
             text: name
         }
    }
}
```

```
Alice
Bob
Jane
Victor
Wendy
```

A **`Repeater`** fetches elements from ***`people`***

- Using the delegate to display model elements as Text items

A **`Column`** arranges them vertically

- Using anchors to make room for the items

# Role and Property Name Clash

```
ListModel {
     id: weirdList
     ListElement { text: "Alice" }
}
Column {
     anchors.fill: parent
     Repeater {
          model: weirdList
          delegate: Text{
                // Will not work;
                text: text
                //Use:  text: model.text
           }
     }
}
```

If a model property shares the same name as delegate properties, the model property needs to be qualified when invoked inside the delegate using model.<role>

## ListModel: Working With Items

A `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} )
    ```
  - **Insert** item data using JavaScript dictionaries at a specific index:
    ```
    bookmarkModel.insert(index, {"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, numberOfItems)
    ```

- Roles (item types) may be dynamic
  - Set `dynamicRoles` property to true
  - Strongly discouraged
  - Using dynamic roles is 4-6 times slower than using static ones
  - Use `QVariantMap` instead

# Integers As Models

An integer can be used as a model that contains a certain number of types.

- In this case, the model does not have any data roles
- Note: The limit on the number of items in an integer model is 100,000,000

```
Item {
   width: 100; height: 200
   Component {
       id: itemDelegate
       Text { text: "item number: " + index }
   }
   ListView {
       anchors.fill: parent
       model: 5
       delegate: itemDelegate
   }
}
```

## View Types

Standard views

- `GridView`
- `ListView`
- `TableView`

More view types provided in other modules

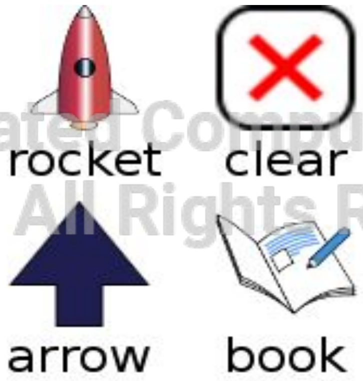
- Qt Location - `MapObjectView`, `MapItemView`
- QtQuick Controls 2 - `StackView`, `ScrollView`, `SwipeView`
- QtCharts - `ChartView`, `PolarChartView`
- And More...

# View Previews

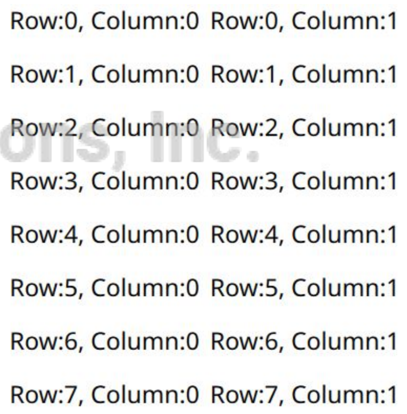
## ListView

Alice
Bob
Jane
Victor
Wendy

## GridView



## TableView



| | |
|---|---|
| Row:0, Column:0 | Row:0, Column:1 |
| Row:1, Column:0 | Row:1, Column:1 |
| Row:2, Column:0 | Row:2, Column:1 |
| Row:3, Column:0 | Row:3, Column:1 |
| Row:4, Column:0 | Row:4, Column:1 |
| Row:5, Column:0 | Row:5, Column:1 |
| Row:6, Column:0 | Row:6, Column:1 |
| Row:7, Column:0 | Row:7, Column:1 |

## Defining a Delegate

- Define a `Component` to use as a delegate
  - With an `id` so that it can be referenced
  - Describes how each model index will be displayed

- Properties of list elements can be referenced
  - Use a `Text` item for each list element
  - Use the value of the `name` data role from each list element

```
Component {
    id: nameDelegate
    Text {
        text: name
        font.pixelSize: 32
}}
ListView{
    model: nameModel
    delegate: nameDelegate
}
```

## Delegates, Contexts, and Attached Properties

- Each property is exposed in one context
  - defines how the property can be accessed together with the scope rules
- `Repeater`, `Instantiator`, and `View` types expose properties to delegate instances in sub-contexts
  - Allows the parent to expose properties that are visible in the sub-context only
  - `index` and `modelData` (if the model is a string or object list) roles available to the delegate
- Views also provide attached properties to delegates

```
Component {
    id: nameDelegate
    Text {
        property var listView: ListView.view
        text: name; font.pixelSize: 32
        color: (listView.currentIndex === index) ? "red" : "black"
    }
}
```

# ListView

- A ListView has model and delegate properties.
- Items can be laid horizontally or vertically
- ListViews are flickable since it inherits from Flickable type.

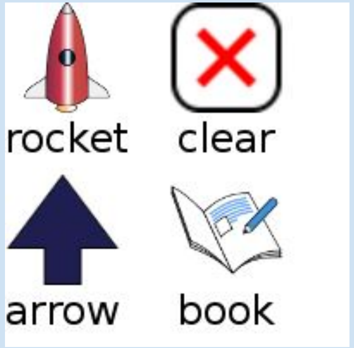
```
Item {
   width: 100; height: 200
   Component {
        id: itemDelegate
        Text { text: name }
   }
   ListView {
        anchors.fill: parent
        model: people
        delegate: itemDelegate
   }
}
```

# GridView

- The setup is the same as with **`ListView`**
- Uses data from a list model
  - Unlike **`TableView`**
  - Think of it as **`ListView`** in "icon mode"

```
ListModel {
    id: iconList
    ListElement { label: "rocket"; iconImage: "file://rocketImage.jpg" }
    ...
}

GridView {
    anchors.fill: parent
    model: iconList
    delegate: iconDelegate
    clip: true
}
```



## GridView

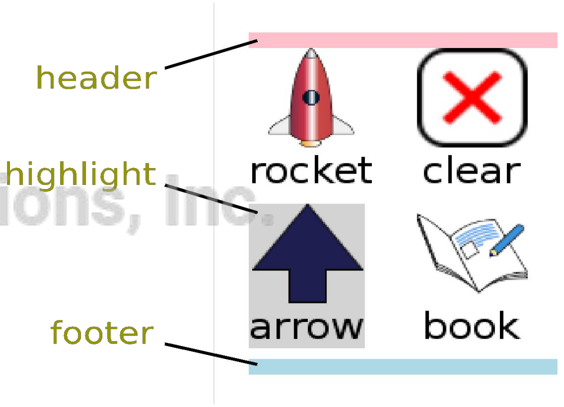
Set up the delegate:

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

# GridView: Decoration and Navigation

**GridView** is also undecorated and a flickable by default

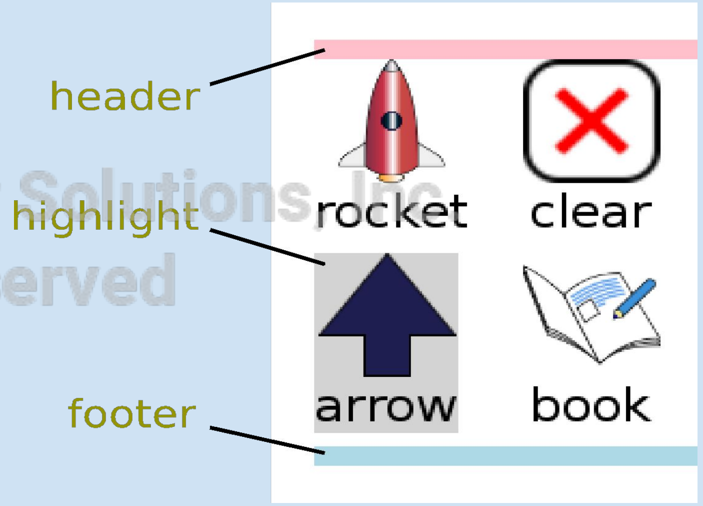
- To add decoration:
  - Define a **header** and **footer**
  - Define a **highlight** delegate to show the current item
- To configure for navigation
  - Set **focus** to allow keyboard navigation
  - **highlight** also helps the user with navigation
  - Unset **interactive** to disable dragging and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



# GridView: Decoration and Navigation

```
GridView {
    ...
    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"
    }
}
```



# TableView

- `TableView` is new as of Qt 5.12
  - `import QtQuick 2.12`
- Don’t confuse it with the QtQuick Controls 1 `TableView`
  - This element has known performance issues
- Only a subsection of the table is normally visible in the viewport
  - `TableView` inherits `Flickable`, if you flick
    - New rows and columns enter the viewport, and old ones are removed
    - Rows and columns that move out are reused
    - `TableView` supports models of any size without affecting performance
- A `TableView` can display `ListModel` data
  - But it will only populate the first column in a `TableView`
- To create models with multiple columns:
  - Create a model in C++ that inherits `QAbstractTableModel`, and expose it to QML

TIME FOR A LIVE DEMO
38
WHAT COULD GO WRONG?
memegenerator.net

# Q&A Session

If you have additional questions or feedback,
please contact us at QtTraining@ics.com

COMING SOON!

Hands-on Virtual Training:
Building an Embedded Device Application with Qt

Course begins July 14

More details and registration available early June