# Qt Essentials - Model View Module

## Training Course

Visit us at `http://qt.digia.com`

Produced by Digia Plc.

*Material based on Qt 5.0, created on September 27, 2012*

Digia Plc.

## Module: Model/View

- Model/View Concept
- Showing Simple Data
- Proxy Models
- Custom Models

## Overview

**Using Model/View**

- Introducing to the concepts of model-view
- Showing Data using standard item models
- Understand the limitations of standard item models
- How to interface your model with a data backend
- Understand what are proxy models and how to use them

**Custom Models**

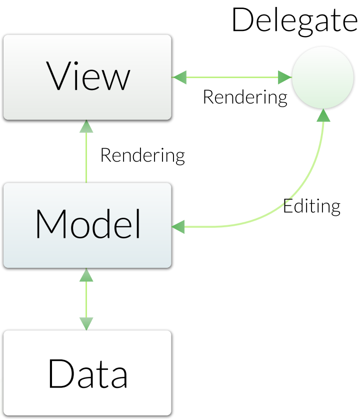
- Writing a simple read-only custom model.

- Model/View Concept
- Showing Simple Data
- Proxy Models
- Custom Models

## Why Model/View?

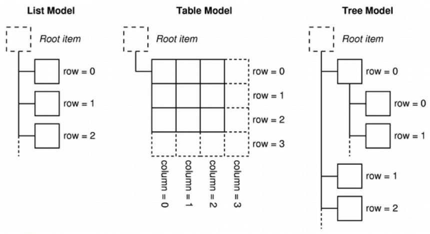
- **Isolated domain-logic**
  - From input and presentation
- **Makes Components Independent**
  - For Development
  - For Testing
  - For Maintenance
- **Foster Component Reuse**
  - Reuse of Presentation Logic
  - Reuse of Domain Model

## Model/View-Components



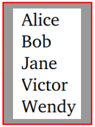
Demo modelview/ex-simple

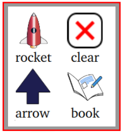
## Model Structures

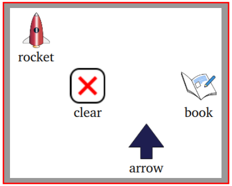


## Display the Structure - View Classes

- **QtQuick ItemView**
  - Abstract base class for scrollable views
- **QtQuick ListView**
  - Items of data in a list
- **QtQuick GridView**
  - Items of data in a grid
- **QtQuick PathView**
  - Items of data along a specified path



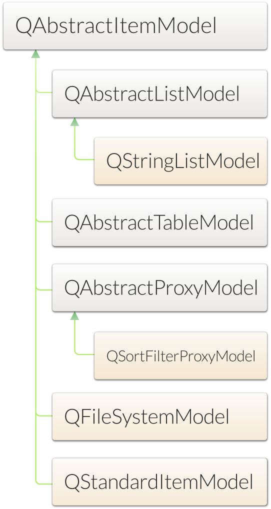




## Adapts the Data - Model Classes

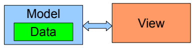
- **`QAbstractItemModel`**
  - Abstract interface of models
- **Abstract Item Models**
  - Implement to use
- **Ready-Made Models**
  - Convenient to use
- **Proxy Models**
  - Reorder/filter/sort your items

See Model Classes Documentation

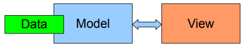


## Data - Model - View Relationships

- **Standard Item Model**
  - Data+Model combined
  - View is separated
  - Model is your data



- **Custom Item Models**
  - Model is adapter to data
  - View is separated



## Addressing Data - `QModelIndex`

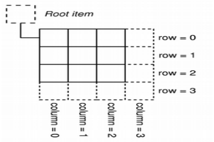
- Refers to item in model
- Contains all information to specify location
- Located in given row and column
  - May have a parent index

- **QModelIndex API**
  - `row()` - row index refers to
  - `column()` - column index refers to
  - `parent()` - parent of index
    - or `QModelIndex()` if no parent
  - `isValid()`
    - Valid index belongs to a model
    - Valid index has non-negative row and column numbers
  - `model()` - the model index refers to
  - `data( role )` - data for given role

## `QModelIndex` in Table/Tree Structures

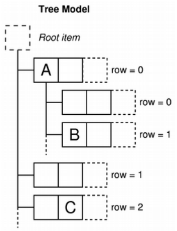
- **Rows and columns**
  - Item location in table model
  - Item has no parent (parent.isValid() == false)

```
indexA = model->index(0, 0, QModelIndex());
indexB = model->index(1, 1, QModelIndex());
indexC = model->index(2, 1, QModelIndex());
```

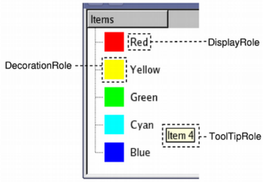


- **Parents, rows, and columns**
  - Item location in tree model

```
indexA = model->index(0, 0, QModelIndex());
indexC = model->index(2, 1, QModelIndex());
// asking for index with given row, column and parent
indexB = model->index(1, 0, indexA);
```



See Model Indexes Documentation



- **Item performs various roles**
  - for other components (delegate, view, ...)
- **Supplies different data**
  - for different situations
- **Example:**
  - `Qt::DisplayRole` used displayed string in view
- **Asking for data**

```
QVariant value = model->data(index, role);
// Asking for display text
QString text = model->data(index, Qt::DisplayRole).toString()
```

- **Standard roles**
  - Defined by `Qt::ItemDataRole`
  - See enum Qt::ItemDataRole Documentation

## Mapping Item Roles from/to QML

- Item Roles in C++

```
// Asking for display text
QString text = model->data(index, Qt::DisplayRole).toString()
```

- Item properties in QML

```
onCurrentIndexChanged: {
    var text = model.get(index).display
}
```

- Default mappings
  - `Qt::DisplayRole` in C++ is display in QML
  - `Qt::DecorationRole` in C++ is decoration in QML
- Add additional mappings by reimplementing `QAbstractItemModel::roleNames()`

## Exporting models to QML

- Export model instance
  - Create model instance in C++
  - Set as a context property on the view's engine

```
CustomModel *model = new CustomModel;
QQuickView view;
view.engine()->rootContext("_model", model);
```

  - Use in QML by id

```
ListView { model: _model }
```

- Export model type
  - Register custom model class with QML type system

```
qmlRegisterType<CustomModel>("Models", 1, 0, "CustomModel");
```

  - Use in QML like any other QML element

```
import Models 1.0
ListView {
    model: CustomModel {}
}
```

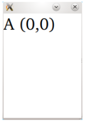
## Recap of Model/View Concept

- **Model Structures**
  - List, Table and Tree
- **Components**
  - Model - Adapter to Data
  - View - Displays Structure
  - Delegate - Paints Item
  - Index - Location in Model
- **Views**
  - `ListView`
  - `GridView`
  - `PathView`
- **Models**
  - `QAbstractItemModel`
  - Other Abstract Models
  - Ready-Made Models
  - Proxy Models
- **Index**
  - `row(),column(),parent()`
  - `data( role )`
  - `model()`
- **Item Role**
  - `Qt::DisplayRole`
  - Standard Roles in `Qt::ItemDataRoles`

- Model/View Concept
- Showing Simple Data
- Proxy Models
- Custom Models

## QStandardItemModel - Convenient Model

- QStandardItemModel
    - Classic item-based approach
    - Only practical for small sets of data


A (0,0)

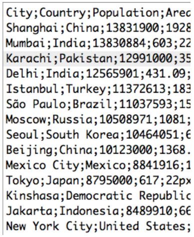
```
model = new QStandardItemModel(parent);
item = new QStandardItem("A (0,0)");
model->appendRow(item);
model->setItem(0, 1, new QStandardItem("B (0,1)"));
item->appendRow(new QStandardItem("C (0,0)"));
```

- *"B (0,1)" and "C (0,0)" - Not visible. (list view is only 1-dimensional)*

See QStandardItemModel Documentation | Demo modelview/ex-QStandardItemModel

## Meet the City Engine

- Our Demo Model
  - 62 most populous cities of the world
  - Data in CSV file
- Data Columns
  - *City | Country | Population | Area | Flag*
- Implemented as data backend
  - Internal implementation is hidden
  - Code in `CityEngine` class



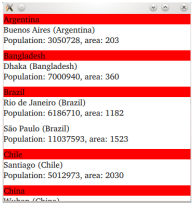
```
City;Country;Population;Area
Shanghai;China;13831900;1928
Mumbai;India;13830884;603;22
Karachi;Pakistan;12991000;35
Delhi;India;12565901;431.09;
Istanbul;Turkey;11372613;183
São Paulo;Brazil;11037593;15
Moscow;Russia;10508971;1081;
Seoul;South Korea;10464051;6
Beijing;China;10123000;1368.
Mexico City;Mexico;8841916;1
Tokyo;Japan;8795000;617;22px
Kinshasa;Democratic Republic
Jakarta;Indonesia;8489910;66
New York City;United States;
```

## Our Backend CityEngine API

```
public CityEngine : public QObject {
  // returns all city names
  QStringList cities() const;
  // returns country by given city name
  QString country(const QString &cityName) const;
  // returns population by given city name
  int population(const QString &cityName) const;
  // returns city area by given city name
  qreal area(const QString &cityName) const;
  // returns country flag by given country name
  QIcon flag(const QString &countryName) const;
  // returns all countries
  QStringList countries() const;
  // returns city names filtered by country
  QStringList citiesByCountry(const QString& countryName) const;
};
```

## Lab: Standard Item Model for CityEngine

- Implement `setupModel()` in `citymodel.cpp`
- Display cities grouped by countries



```
Argentina
Buenos Aires (Argentina)
Population: 3050728, area: 203
Bangladesh
Dhaka (Bangladesh)
Population: 7000940, area: 360
Brazil
Rio de Janeiro (Brazil)
Population: 6186710, area: 1182
São Paulo (Brazil)
Population: 11037593, area: 1523
Chile
Santiago (Chile)
Population: 5012973, area: 2030
China
Wuhan (China)
```

Lab modelview/lab-cities-standarditem

- Model/View Concept
- Showing Simple Data
- Proxy Models
- Custom Models

## Proxy Model - `QSortFilterProxyModel`

- `QSortFilterProxyModel`
  - Transforms structure of source model
  - Maps indexes to new indexes

```
view = new QQuickView(parent);
// insert proxy model between model and view
proxy = new QSortFilterProxyModel(parent);
proxy->setSourceModel(model);
view->engine()->rootContext()->setContextProperty("_proxy", proxy);
```

*Note:* Need to load all data to sort or filter

## Sorting/Filtering - `QSortFilterProxyModel`

- **Filter with Proxy Model**

```
// filter column 1 by "India"
proxy->setFilterWildcard("India");
proxy->setFilterKeyColumn(1);
```

- **Sorting with Proxy Model**

```
// sort column 0 ascending
proxy->sort(0, Qt::AscendingOrder);
```

- **Filter via `TextInputs` signal**

```
TextInput {
    onTextChanged: _proxy.setFilterWildcard(text)
}
```

Demo modelview/ex-sortfiltertableview

- Model/View Concept
- Showing Simple Data
- Proxy Models
- Custom Models

## Implementing a Model

- Variety of classes to choose from
  - **QAbstractListModel**
    - One dimensional list
  - **QAbstractTableModel**
    - Two-dimensional tables
  - **QAbstractItemModel**
    - Generic model class
  - **QStringListModel**
    - One-dimensional model
    - Works on string list
  - **QStandardItemModel**
    - Model that stores the data

- **Notice:** Need to subclass *abstract* models

```
class MyModel: public QAbstractListModel {
public:
  // return row count for given parent
  int rowCount( const QModelIndex &parent) const;
  // return data, based on current index and requested role
  QVariant data( const QModelIndex &index,
                 int role = Qt::DisplayRole) const;
};
```

Demo modelview/ex-stringlistmodel

## Step 2: Suppling Header Information

```
QVariant MyModel::headerData(int section,
                             Qt::Orientation orientation,
                             int role) const
{
  // return column or row header based on orientation
}
```

Demo modelview/ex-stringlistmodel-2

## Step 3: Enabling Editing

```
// should contain Qt::ItemIsEditable
Qt::ItemFlags MyModel::flags(const QModelIndex &index) const
{
 return QAbstractListModel::flags() | Qt::ItemIsEditable;
}

// set role data for item at index to value
bool MyModel::setData( const QModelIndex & index,
                const QVariant & value,
                int role = Qt::EditRole)
{
  ... = value; // set data to your backend
  emit dataChanged(topLeft, bottomRight); // if successful
}
```

Demo modelview/ex-stringlistmodel-3

## Step 4: Row Manipulation

```
// insert count rows into model before row
bool MyModel::insertRows(int row, int count, parent) {
   beginInsertRows(parent, first, last);
   // insert data into your backend
   endInsertRows();
}

// removes count rows from parent starting with row
bool MyModel::removeRows(int row, int count, parent) {
   beginRemoveRows(parent, first, last);
   // remove data from your backend
   endRemoveRows();
}
```

Demo modelview/ex-stringlistmodel-4

## Lab: City List Model

- Please implement a City List Model
- Given:
  - Start with solution of `modelview/lab-cities-standarditem`
- Your Task:
  - Rebase `CityModel` to `QAbstractListModel`
- **Optional**
  - Make the model editable
  - Enable adding/removing cities

Lab modelview/lab-cities-standarditem