# Object-Oriented Programming

Iuliana Bocicor
*iuliana@cs.ubbcluj.ro*

Babes-Bolyai University

2020

## Overview

## Qt Item based widgets I

- QListWidget, QTableWidget, QTreeWidget
- Item widgets are populated with the entire content of a data set.
- Searches, edits are performed on the data held in the widgets.
- The data needs to be synchronized, written back to the data source (file, database, network).

# Qt Item based widgets II

## Advantages

- easy to understand;
- simple to use.

## Drawbacks

- does not scale well with very large data sets;
- does not work if we have multiple views of the same data set;
- requires data duplication.

## Model-View-Controller (MVC) I

Is a flexible approach to visualizing large data sets.

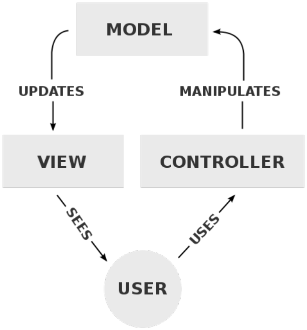


Figure source: https://en.wikipedia.org/wiki/Model-view-controller#/media/File:MVC-Process.svg

## Model-View-Controller (MVC) II

**Model**

- Represents and manages the data of the application domain.
- Is responsible for:
  - fetching the data that is needed for view;
  - writing back any changes (requests which come from the controller).

## Model-View-Controller (MVC) III

### View

- Presents the data to the user.
- Even if we have a large dataset, only a limited amount of data is visible. That is the only data that is requested by the view.

### Controller

- Mediates between the user and the view.
- Interprets user input and commands the model or the view to change as appropriate.
- Converts user actions (which come from the view) into requests to navigate or edit data.

## Model/View Architecture in Qt I

- Model/View is a technology used to separate data from their visual representation (views).
- The view and controller objects from MVC are combined.
- The way the data is stored is separated from the way the data is presented to the user.

## Model/View Architecture in Qt II

- Allows displaying the same data in different views.
- Implementing new types of views is possible, without changing the underlying data structures.
- You can find a more detailed tutorial at: `http://doc.qt.io/qt-5/modelview.html`.

## Model/View Architecture in Qt III

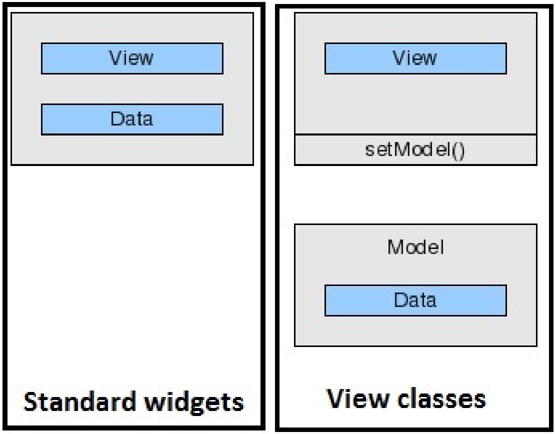


Figure source: http://doc.qt.io/qt-5.6/modelview.html

# Model/View Architecture in Qt IV

- Model/view architecture is very suitable for handling large data sets, complex data items, database integration, multiple data views.
- User input is handled with delegates.
- The *delegate* is used to provide fine control over how items are rendered and edited.
- Qt provides a default delegate for every type of view (which is sufficient for most applications).

## Model/View Architecture in Qt V

### How does it work?

- The model communicates with a source of data.
- The model must provide an *interface* for the views.
- The view obtains *model indexes* from the model - references to items of data.
- The delegate renders the items of data and communicates with the model when the data is edited.

## Model/View Architecture in Qt VI

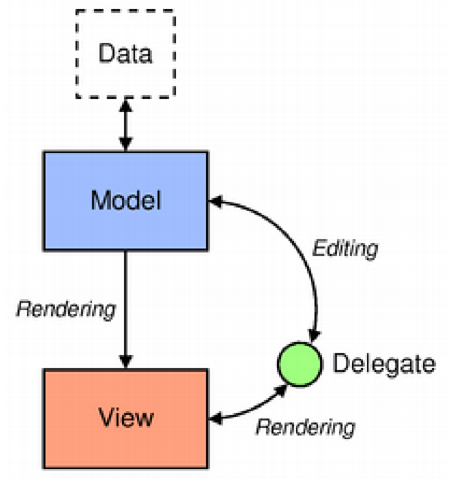


Figure source: http://doc.qt.io/qt-5.6/modelview.html

## Predefined classes for models, views, delegates

- Models, views and delegates are defined by *abstract classes* that provide common interfaces and sometimes defaut implementations.
- These abstract classes should be subclassed for specialized components.
- Models, views, and delegates communicate with each other using signals and slots.

## Models

- QAbstractItemModel is the class that defines an interface used by views and delegates to access data.
- All item models are based on this abstract class.
- This class provides a flexible interface, which can be used with views that represent data in the form of tables, lists, and trees.
- There are also QAbstractListModel and QAbstractTable-Model, which are more appropriate for models representing list of table-like data structures.

## Predefined models

Qt provides several predefined models for use with the view classes:

- QStringListModel - stores a list of strings.
- QStandardItemModel - stores arbitrary hierarchical data.
- QDirModel - encapsulates the local file system.
- QSqlQueryModel - encapsulates an SQL result set.
- QSqlTableModel - encapsulates an SQL table.
- QSqlRelationalTableModel - encapsulates an SQL table with foreign keys.
- QSortFilterProxyModel - sorts and/or filters another model.

## Views

- QAbstractItemView is the abstract base class for views.
- There are complete implementations for the following types of views:
  - QListView - displays a list of items.
  - QTableView - displays data from a model in a table.
  - QTreeView - shows model items of data in a hierarchical list.

## Demo I

### Genes List

- Displaying a list of genes using a list widget and then a list view with a predefined model (QStringListModel).
- Large data sets are displayed much faster.
- For $\sim$ 21000 genes: the list widget needs $\sim$ 9 seconds, while the view needs $\sim$ 2 seconds.

**DEMO**

Using predefined models - genes list (*Lecture11_demo_predefined_models*).

## Demo II

### Directory Tree View

- Recursively displaying the sub-folders of a folder using the predefined view QTreeView and the predefined model QDirModel.

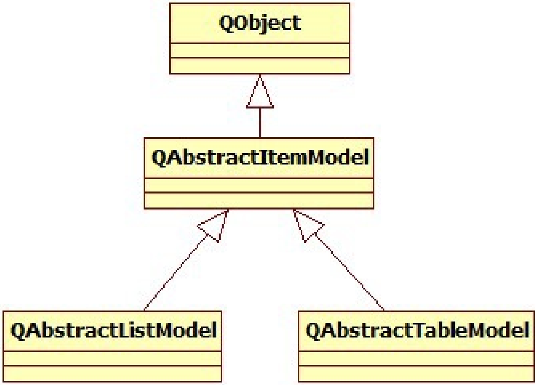
**DEMO**

Using predefined models - directory tree view (*Lecture11_demo - main.cpp - directory tree*).

## Custom models I

- QAbstractItemModel is the class representing the model for any Qt Item View Class.
- This is able to represent list data (rows), table data (rows, columns) or hierarchical data (tree structure: parents, children).
- To create a custom model, create a new class, which extends the appropriate Qt model class (QAbstractItemModel or QAbstractListModel or QAbstractTableModel).

## Custom models II



## Example - genes table model I

- Inherit from QAbstractTableModel.
- Provide implementation for at least the following three functions: rowCount, columnCount, data.
- The QModelIndex
  - is used to locate data in a model;
  - it is an index which refers to an item in a model and is used by views;
  - each index is located in a given row and column, and may have a parent index.

## Example - genes table model II

```
class GenesTableModel: public QAbstractTableModel
{
public:
    GenesTableModel(QObject* parent = NULL);
    ~GenesTableModel();

    // number of rows
    int rowCount(const QModelIndex &parent = QModelIndex
        {}) const override;

    // number of columns
    int columnCount(const QModelIndex &parent =
        QModelIndex{}) const override;

    // Value at a given position
    QVariant data(const QModelIndex &index, int role = Qt
        ::DisplayRole) const override;
};
```

## Controlling the text appearance - item roles I

- Items in a model can perform various *roles*.
- Each item in the model has a set of data elements associated with it, each with its own role.
- When asking for the item's data from a model, the role can be specified and thus we obtain the type of data that we want.
- There is a set of standard roles defined in Qt::ItemDataRole, which cover the most common uses for item data.

## Controlling the text appearance - item roles II

| **enum Qt::ItemDataRole** | **Description** | **Type** |
|---|---|---|
| Qt::DisplayRole | The data to be rendered in the form of text. | QString |
| Qt::EditRole | The data in a form suitable for editing in an editor. | QString |
| Qt::FontRole | The font used for items. | QFont |
| Qt::TextAlignmentRole | The alignment of the text. | Qt::AlignmentFlag |
| Qt::BackgroundRole | The background brush. | QBrush |
| Qt::ForegroundRole | The foreground brush (text colour). | QBrush |

## Table/Tree headers

- The model also controls the headers for a table/tree view.
- For this, the function headerData must be implemented.
- The QVariant class acts like a union for the most common Qt data types. A QVariant object holds a single value of a single type at a time.

```
QVariant headerData(int section, Qt::Orientation
    orientation, int role = Qt::DisplayRole) const
    override;
```

## Demo

**DEMO**

Implementing a custom model (*Lecture11_demo_custom_models*).

## Edit model values

- Implement the methods setData (will be called when a cell is edited) and flags (returns the item flags for a given index).
- When the data has been set, the model must let the views know that some data has changed. This is done by emitting the dataChanged() signal.

**DEMO**

Implementing a custom model (*Lecture11_demo_custom_models*).

## Multiple views for the same model

- Multiple views attached to the same model allow the user to interact with the data in different ways.
- Qt automatically keeps multiple views in sync, reflecting changes in the model.
- If the underlying data is changed, only the model needs to be changed; the views will behave correctly.
- Demo below: 3 different views (list view, table view, tree view) using the same model.

**DEMO**

Implementing a custom model (*Lecture11_demo_custom_models*).

## Filtering and sorting I

- The QSortFilterProxyModel class provides support for sorting and filtering data passed between another model and a view.
- The structure of the source model is transformed by mapping the model's indexes to new indexes.
- The given source model is restructured, without requiring transformations on the underlying data, and without duplicating the data in memory.

## Filtering and sorting II

- After an object QSortFilterProxyModel is created, use the setSourceModel() and set the QSortFilterProxyModel on the view.
- Use the sortingEnabled property of the QTableView and QTreeView to activate sorting by clicking on the header.

**DEMO**

Sorting (*Lecture11_demo_custom_models*).

## Populating models incrementally

- For large data sets, items should be added to the model in batches and only when they are needed by the view.
- Reimplement the methods fetchMore() and canFetchMore() from QAbstractItemModel.
- canFetchMore() is called by the view when it needs more items.

DEMO

Sorting (*Lecture11_demo_custom_models* - class PaginatedGenesTableModel).

## Delegates

- Delegates are used to render and edit individual items.
- They provide input capabilities and are also responsible for rendering individual items in some views.
- Usually, the default delegate is sufficient.
- However, the way that items of data are rendered and edited can be customized by using custom delegates.

## Defining custom delegates

- We can create our own delegate class and set it on the view that is supposed to use it.
- The standard interface for controlling delegates is defined in the QAbstractItemDelegate class.
- The default delegate implementation that is used by Qt's standard views is QStyledItemDelegate. This should be used as base class when implementing custom delegates.

**DEMO**

Custom delegates (*Lecture11_demo_custom_models* - PictureDelegate).