Exercises Lecture 8 – The Model View Framework

Aim: This exercise explores the standard model view classes, as well as the support classes used to customize the look and feel of views.

Duration: 1h

Viewing a model

This exercise comes with a source code package. The contents are a number of starting point projects for the different steps of this exercise. Please extract the package and use the *standardmodel* project as the starting point for this step.

The application creates an instance of `QStandardItemModel` and an instance of `QTableView`. The goal is to populate the model with data of a simple todo list and assign the model to the table view. Each todo item is contained on a row. Each row is then composed from three columns: the name of the activity, its deadline and the percentage (integer value in a range of 0-100) of completeness.

Field Name	Field Type
Name	string
Deadline	date
Completeness	integer (0-100)

Start by calling the `QStandardItemModel::setHorizontalHeaderLabels()` method to set titles for the columns. The list of labels are provided in the source code.

The next step is to add three rows of tasks according to the table below. Each row is built from one `QStandardItem` object (one for each column) which are added to the model using the `appendRow` method.

Name	Deadline	Completeness
Clean house	5/3 2011	0
Buy groceries	10/4 2011	50
Exercise	1/1 2011	25

Creating a standard item containing a text string is easy. To get a proper representation of other data types such as integers and dates, one has to use the `setData` method as shown below:

```
    QList<QStandardItem*> row;
    QStandardItem *item;
...
    row.clear();
...
    item = new QStandardItem();
    item->setData(QDate(2011, 4, 10), Qt::DisplayRole);
    row << item;
...
    model->appendRow(row);
```

Testing the application now, you will realize that you can edit the contents of the model. This might be good or bad, but it is easy to control.

For each `QStandardItem` of the first column, set the flags `ItemIsSelectable` and `ItemIsEnabled` using the `setFlags` method. This will make them selectable and enabled, but not editable. For editable items, the `ItemIsEditable` flag must be set as well. This flag is enabled by default for `QStandardItem`

instances.

Additional roles

In this step, you will extend the model from the first step with more data using different roles. As the project has been restructured slightly, use the *multirolesmodel* project as the starting point.

In this step the todo list application will be enhanced with visual hints. For instance, by changing the background color of items as well as the font used.

For QStandardItem objects, it is possible to use dedicated methods to set the different roles (e.g. `setForeground()` for setting the item color). In order to familiarize yourself with the actual roles, you are encouraged to use the `setData()` method instead. This lets you specify the data for each role in a more specific manner.

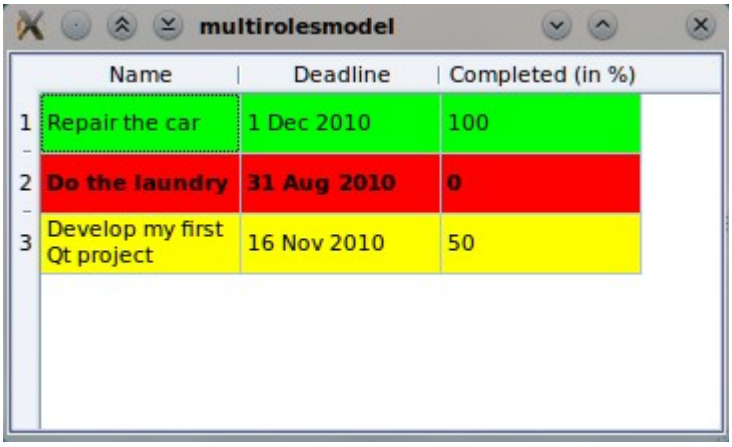
Start by setting a more detailed description for each item as the ToolTipRole role of the first column items. When running the application, try hovering the first column items to verify that a tool tip popup appears.

The next step is to implement and setup the `updateEntry()` method so that the background color of each row reflects the completeness of the task in question. Completed items are green, started items yellow and items that are now started are marked as red.

First, implement the `updateEntry` method. The method takes a single item as argument. You can retrieve the row of the item using the `item->row()` method. You can then call `m_model->item(` *`row, column`* `)` to get access to the rest of the row.

Using the `data` method of the completeness column, you can access the `DisplayRole` data. Use the `toInt` method to convert the returned `QVariant` to an integer.

When you have determined which color to use, set the `BackgroundColorRole` for all items of the row using the `setData` method.



As the program stands now, the `updateEntry` method is called for all new items, but not when items are changed. You can fix this by converting `updateEntry` to a slot (move it from the `private` section to the `private slots` section) and adding the `Q_OBJECT` macro to the class. Then connect the model's `itemChanged` signal to the `updateEntry` slot.

When you have the updateEntry slot working, there is one more alteration to implement. Given the due date, compare it to QDate::currentDate. If the task is overdue, apply a bold font to the item, otherwise, apply a normal font.

You can get the date of an item using the `toDate` function on the `QVariant` returned from the `data` method.

You get the default font of the system by creating a `QFont` instance. You can then make it bold or not using the `setBold` method.

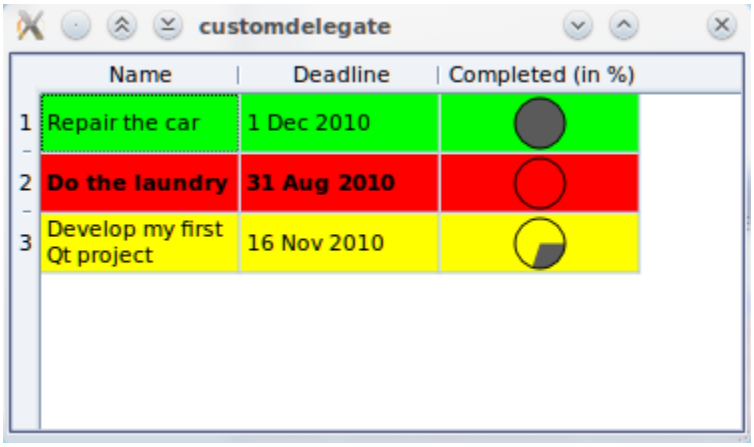
```
QFont font;
font.setBold(isOverdue);
```

Add all this to the `updateEntry` slot and verify that it works.

Custom rendering

Until now, you have relied on the standard features of the graphics view framework. In this step, you will customize the view by implementing a delegate of your own. This will let you visualize the completeness value using custom painting.

Use the *customdelegate* project as the starting point. The goal is to visualize the completeness as a pie chart. This is done using a custom delegate. There is already a skeleton for the delegate in the starting point project that draws the outline of the pie.



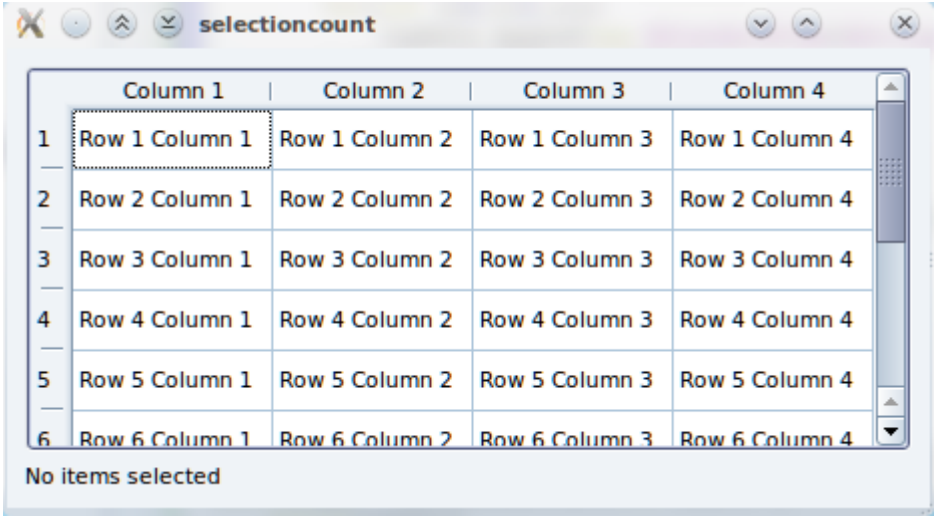
Your task is to extend the paint method of the delegate to draw a pie corresponding to the completeness value. The rectangle to draw within is given, but you have to setup a pen and brush.

To actually draw the pie, you need to calculate the angle to be drawn – multiply 360 degrees by the percentage (i.e. 20% will be 72 degrees). You can use `qRound()` to round the value to the nearest integer.

```
qreal spanInDegrees = 3.6*index.data().toInt();
painter->drawPie(rect, 0, -qRound(16*spanInDegrees));
```

Tracking selections

For the final step of this exercise, use the *selectioncount* project as your starting point. When you run the project, you will see a model with rows and columns and a label under the view that contains text “No items selected”.



The goal is to make the text of the label reflect the current selection in the model. In the project, there is a method called `updateLabelText`. Your task is to call that method with a proper value.

To do that, you need to connect to the `selectionChanged()` signal from the view's selection model to a slot of your own. When your slot gets called you will be able to query the selection model for `selectedIndexes()` and call the `updateLabelText()` slot with that value.

Start by adding a slot to the Widget class. It does not have to accept any arguments. Then connect it to the `selectionChanged` signal from the view's `selectionModel`.

Next, implement your slot to call `updateLabelText` with the value from the selection model's `selectedIndexes` method.

Solution Tips

Step 2

To color the whole row you need to set the background color for each column separately.

Having added the `Q_OBJECT` macro to your class, you have to re-run qmake on the project for Qt to recognize the change.

Step 4

To get the selection model use `m_view->selectionModel()`. The selection model is only available when there is a model set on the view.