Element-based Views Without Model Access

This article shows how to implement element-based class for each of the Interview views that manages without an external model.

Basic Idea

We will treat each entry in these view widgets is an instance of an *element* or *item*. For each of the three widgets there are separate item classes that do not have a common parent class. This also means that the data in a view widget in each case cannot be used in the other two widget types without additional processing. The item classes are each named according to the view widget to which they belong: QListWidgetItem is used as an entry in a list view, QTreeWidgetItem represents an entry in a tree view, and QTableWidgetItem is responsible for displaying entries in a table.

The List View

We will insert a few names into a list widget. We create the entries by QListWidgetItem.By passing this to the view widget as the second argument, it takes over control of the item and inserts it.

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QListWidget listWidget;
    new QListWidgetItem(QObject::tr("Symbian"), &listWidget);
    new QListWidgetItem(QObject::tr("MeeGo"), &listWidget);
    new QListWidgetItem(QObject::tr("Series 40"), &listWidget);
    listWidget.showFullScreen();
    return app.exec();
}
```

There are two ways of instantiating a QListWidgetItem. We can call the item constructor,which expects a string or an icon, followed by a string as a parameter. A suitable view widget can be passed optionally as the third parameter, into which the item is inserted. In this example we use an alternative constructor that gets by without specifying an icon.

List view

The Tree View

QTreeWidget is used as a base class to get the tree structure as a view widget, in which the number of columns is fixed from the beginning. And this is being done with `setColumnCount()`. `setHeaderLabels()` is used to define the header. `addChild()` is used to insert the child entries.

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QTreeWidget treeWidget;
    treeWidget.setColumnCount(1);
    QStringList headerLabels;
    headerLabels << "Qt Tree View";
    treeWidget.setHeaderLabels(headerLabels);
    QTreeWidgetItem *parent =
    new QTreeWidgetItem(&treeWidget,
    QStringList(QObject::tr("Nokia Software Platform")));
    parent->addChild(new QTreeWidgetItem
    (QStringList(QObject::tr("Maemo"))));
    parent->addChild(new QTreeWidgetItem
    (QStringList(QObject::tr("S60/Symbian"))));
    parent->addChild(new QTreeWidgetItem
    (QStringList(QObject::tr("Series 40"))));
    treeWidget.expandItem(parent);
```


(C) Copyright Nokia 2013. All rights reserved.

```
    treeWidget.showFullScreen();
    return app.exec();
}
```

Before we display the widget, we first expand the parent item **Nokia Software Platform** by calling the `expandItem()` slot. Otherwise the user would have to do this with the **+** icon in front of the item.

Tree view

The Table View

The third and last ready-to-use view widget, QTableWidget, is based on QTableView and uses QTableWidgetItem as an item class. The size of the table can be con-veniently specified in the constructor. Items are inserted here with the `setItem()` method, which expects a column and row number and the item itself as arguments.

The following example creates a 7x7 table, in which each cell contains the product of its column and row coordinates:

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QTableWidget tableWidget(7,7);
    for (int row=0;row<tableWidget.rowCount(); row++)
    for (int col=0;col<tableWidget.columnCount(); col++)
    tableWidget.setItem(row, col,
    new QTableWidgetItem(QString::number(row*col)));
    tableWidget.showFullScreen();
    return app.exec();
}
```

Table view

Source Code

The full source code presented in this article is available here File:Adressbook.zip

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.