Unsere AddressItem's ...

Ein POCO (Plain Old C/C++ object)

```
class AddressItem
{
public:
    AddressItem();
    AddressItem(QString& firstName,QString& lastName,QString& adress,QString&
       city,QString& zipCode);

    QString m_firstName;
    QString m_lastName;
    QString m_address;
    QString m_city;
    QString m_zipCode;


    bool operator==(const AddressItem& rhs);

};

       ...
```

Dazu später mehr

… werden in einer Qlist verwaltet.
Unser TableModel erbt von QAbstractTableModel

```
class TableModel : public QAbstractTableModel
{
    Q_OBJECT
public:
    TableModel(QObject *parent=0);
    TableModel(QList< AddressItem > listOflistOfAddressItems, QObject *parent=0);

    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation, int role) const;
    Qt::ItemFlags flags(const QModelIndex &index) const;
    bool setData(const QModelIndex &index, const QVariant &value, int
      role=Qt::EditRole);
    bool insertRows(int position, int rows, const QModelIndex &index=QModelIndex());
    bool removeRows(int position, int rows, const QModelIndex
      &index=QModelIndex());
    QList< AddressItem > getList();

private:
    QList< AddressItem > m_listOflistOfAddressItems;

};
```

Einige Methoden müssen überschrieben werden

Hier sind die AddressItem's

Zum Filtern und Sortieren verwenden wir QSortFilterProxyModel

```
AddressOverviewWindow::AddressOverviewWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::AddressOverviewWindow)
{
        ...
    model = new TableModel(this);

    proxyModel = new QSortFilterProxyModel(model);
    proxyModel->setSourceModel(model);
    proxyModel->setFilterRegExp(QRegExp(QString(), Qt::CaseInsensitive,
      QRegExp::FixedString));
    proxyModel->setFilterKeyColumn(1);
    ui->AddressesTableView->setModel(proxyModel);

    void AddressOverviewWindow::filterColumnSelected(){
      proxyModel->setFilterKeyColumn(ui->filterComboBox->currentIndex());
    }

    void AddressOverviewWindow::filterAdresses(){
      proxyModel->setFilterRegExp(QRegExp(QString(ui->filterLineEdit->text()),
      Qt::CaseInsensitive, QRegExp::FixedString));
    }
        ...
```

Das TableModel ...

... wird im QSortFilterProxyModel verpackt

Die TableView verwendet das proxyModel

Adressen Hinzufügen

```
void AddressOverviewWindow::addEntry(QString firstName,QString lastName,QString
address,QString city,QString zipCode)

{
    QList< AddressItem > list = model->getList();
    AddressItem* addressItem= new
AddressItem(firstName,lastName,address,city,zipCode);

    if (!list.contains(*addressItem)) {
        model->insertRows(0, 1, QModelIndex());

        QModelIndex index = model->index(0, 0, QModelIndex());
        model->setData(index, firstName, Qt::EditRole);
            …
         index = model->index(0, 4, QModelIndex());
        model->setData(index, zipCode, Qt::EditRole);

        QTableView *temp = static_cast<QTableView*>(ui->AddressesTableView);
        proxyModel->sort(temp->horizontalHeader()->sortIndicatorSection());

    } else {
        QMessageBox::information(this, tr("Duplicate Name"),
                                 tr("The adress already exists."));
    }
}
```

*Deshalb wurde der **operator ==** überladen*

Hier werden die Spalten befüllt

Nach dem Einfügen wird sortiert

Adressen Speichern

```
void AddressOverviewWindow::saveFile()

{
    QString fileName = QFileDialog::getSaveFileName(this);
    if (!fileName.isEmpty()) {
        this->writeToFile(fileName);
    }
}
```

Öffnet den File Dialog

```
void AddressOverviewWindow::writeToFile(QString fileName) {
    QFile file(fileName);

    if (!file.open(QIODevice::WriteOnly)) {
        QMessageBox::information(this, tr("Unable to open file"), file.errorString());
        return;
    }

    QList< AddressItem > contacts = model->getList();
    QDataStream out(&file);
    out << contacts;
}
```

*Der **<< operator** von AdressItem muss als freie Funktion überladen werden*

Nochmal zurück zum AddressItem

```
class AddressItem

…

inline QDataStream& operator<<(QDataStream & out, const AddressItem & item ){
    out << item.m_firstName;
    out << item.m_lastName;
    out << item.m_address;
    out << item.m_city;
    out << item.m_zipCode;
    return out;
}

inline QDataStream& operator>>(QDataStream & in, AddressItem & item){
    in >> item.m_firstName;
    in >> item.m_lastName;
    in >> item.m_address;
    in >> item.m_city;
    in >> item.m_zipCode;
    return in;
}
```

Das Overloading funktioniert einfacher als vielleicht zu erwarten

Den brauchen wir fürs Lesen

Quellen

http://doc.qt.nokia.com/4.7-snapshot/itemviews-addressbook.html

http://doc.qt.nokia.com/4.7-snapshot/tutorials-addressbook.html

http://doc.qt.nokia.com/4.7-snapshot/modelview.html

http://doc.qt.nokia.com/4.7-snapshot/qsortfilterproxymodel.html

http://doc.qt.nokia.com/4.7-snapshot/qtableview.html

http://www.youtube.com/playlist?list=PL2D1942A4688E9D63&feature=plcp