Qt

Обзор классов

Лекция 2

Черновик

Кафедра ИВТ и ПМ

2019

Outline

Прошлые темы

- Что такое фреимворк?
- Для чего код организуют именно в фреимворки?
- Назовите примеры фреимворков и их назначение.
- Охарактеризуйте фреимворк Qt.

Прошлые темы

- Что такое API?
- Что такое сигналы и слоты?
- Для чего они используются?
- Что такое парадигма программирования?
- Что такое событийно-ориентированное программирование?
- Для чего предназначены классы QCoreApplication и QApplication?

Outline

Outline

Виджеты

QWidget - основной класс для всех элементов интерфейса пользователя.

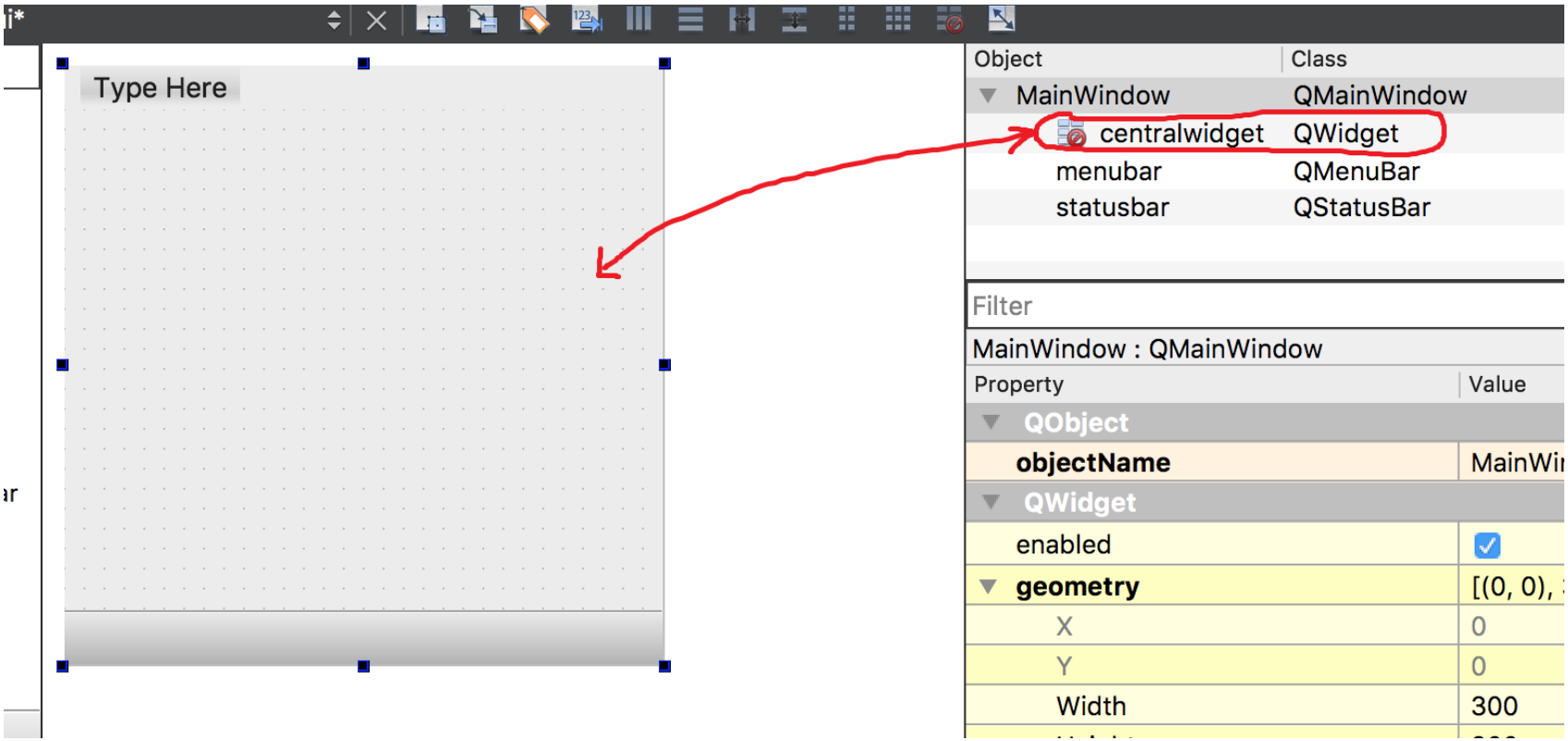
Он имеет базовые свойства виджета (размеры, цвет, шрифт и т.д) принимает события мыши и клавиатуры, рисует самого себя на экране, ...

QWidget

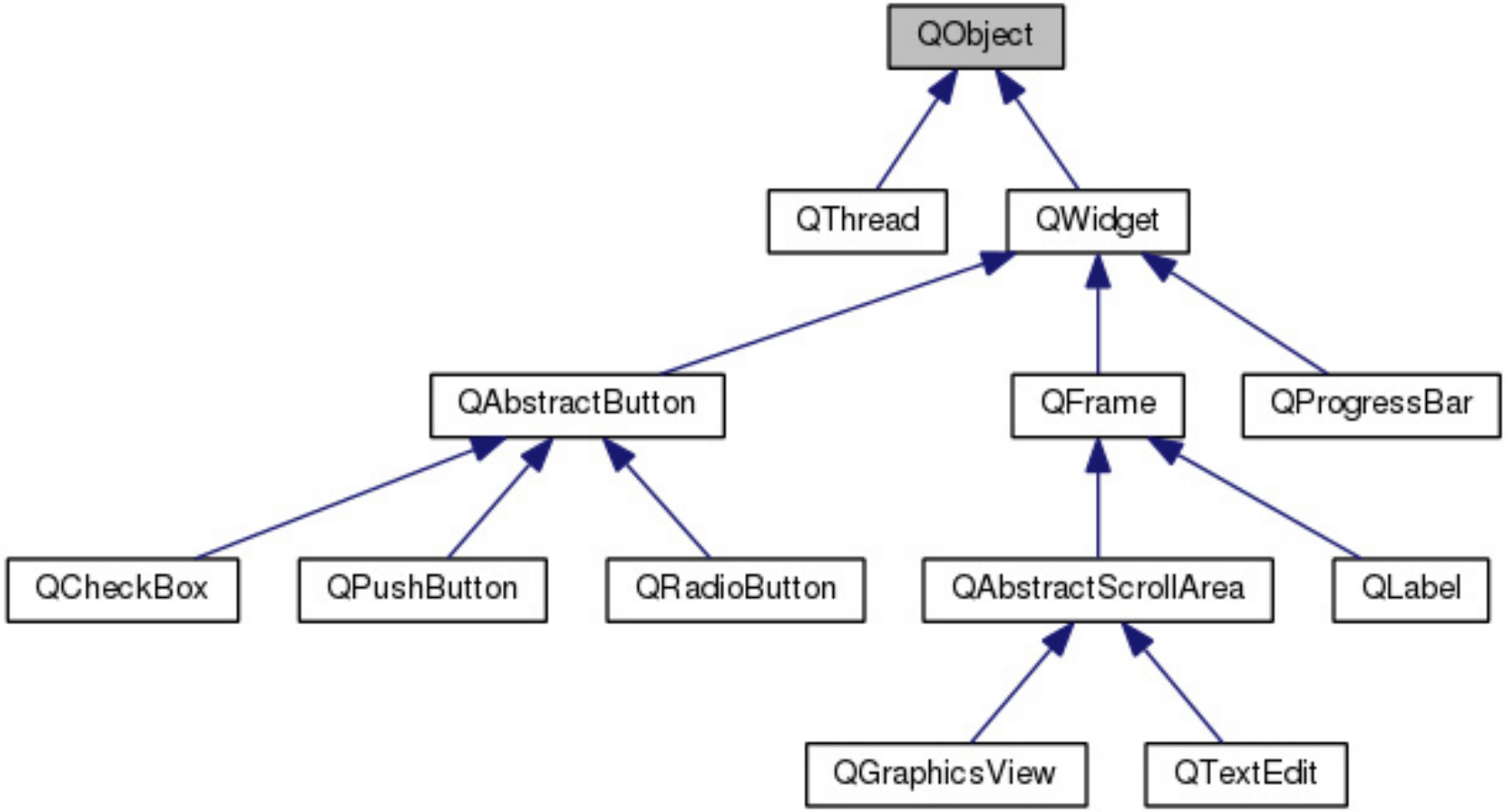
Большинство методов QWidget (так и остальных классов Qt) задающих свойства имеют названия `setProperty` и `property` для задания и получения свойства соответственно.

Свойства QWidget в дизайнере QtCreator

Свойство	Значение
QObject	
objectName	centralWidget
QWidget	
enabled	☑
geometry	[(0, 39), 736 x 487]
sizePolicy	[Preferred, Preferred, 0, 0]
minimumSize	0 x 0
maximumSize	16777215 x 16777215
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Унаследованная
font	A [Ubuntu, 11]
cursor	Arrow
mouseTracking	☐
tabletTracking	☐
focusPolicy	NoFocus
contextMenuPolicy	DefaultContextMenu
acceptDrops	☐
toolTip	
toolTipDuration	-1
statusTip	
whatsThis	
accessibleName	
accessibleDescription	
layoutDirection	LeftToRight
autoFillBackground	☐
styleSheet	
locale	Russian, Russia
inputMethodHints	ImhNone


Type Here
Object
Class
MainWindow
QMainWindow
centralwidget
QWidget
menubar
QMenuBar
statusbar
QStatusBar
Filter
MainWindow : QMainWindow
Property
Value
QObject
objectName
QWidget
enabled
geometry
X
Y
Width
300

QWidget



Все классы виджетов наследуются от QWidget, поэтому они имеют много общих методов и полей.

Основные свойства QWidget

- Размер

```
width();     // ширина в пикселях
height();    // высота в пикселях

// задать размер виджета
resize( int w, int h);

// задать и зафиксировать размер
setFixedSize(w, h)
```

Основные свойства QWidget

- **enabled** : bool
  Свойство отвечающее за "включение выключение"элемента интерфейса.

```
bool isEnabled() const\\
void setEnabled(bool)
```

- **visible** : bool
  - виден ли элемент интерфейса пользователю.

```
        bool          isVisible() const\\
virtual void          setVisible(bool visible)\\
```

Основные свойства QWidget

- Фокус ввода с клавиатуры
  focus : bool

```
bool hasFocus() const
void setFocus()

// Можно ли устанавливать фокус ввода?
// Каким способом устанавливать фокус ввода?
focusPolicy() const
void setFocusPolicy(Qt::FocusPolicy policy)
```

Виджеты

- Один виджет может содержать другие виджеты
- При создании окна с несколькими элементами интерфейса один из виджетов должен быть главным. По умолчанию этот виджет называется central_widget. Он как раз и содержит другие виджеты
- Причём виджет может содержать другой не только визульно, но и хранить указатели на те виджеты, которые он содержит
- Если виджеты создаётся в дизайнере форм Qt Creator'а, то эта связь устанавливается автоматически во время генерации cpp файла из ui файла формы.
- Такая иерархия позволяет удаляя основной виджет (экземпляр класса QWidget) автоматически удалить и остальные виджеты, которыми владеет основной

Виджеты

- Если виджеты создаются вручную, то может понадобится у подчинённых виджетов при вызове конструктора передать параметром (parent) указатель на основной виджет.

```
QWidget::QWidget(QWidget *parent = Q_NULLPTR)
```

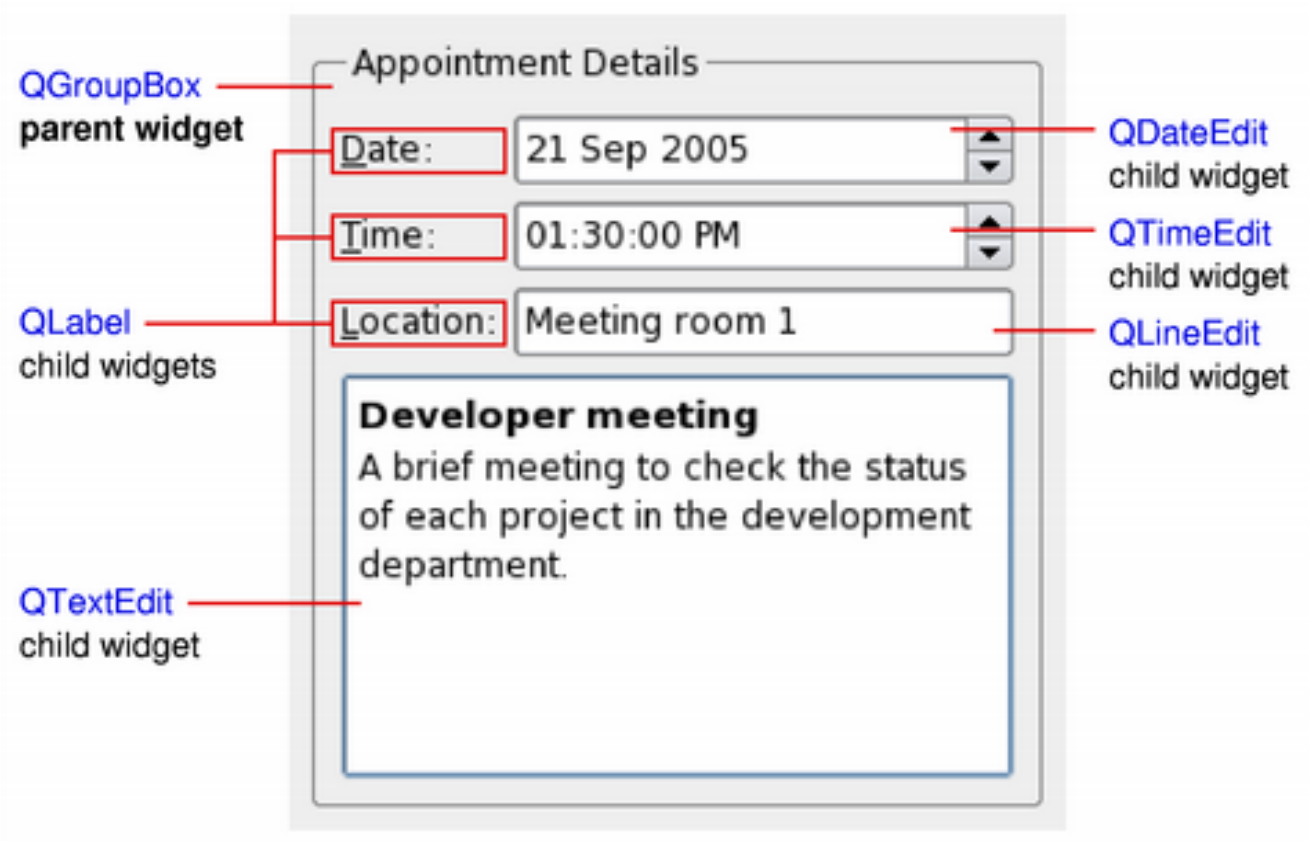
- В своём конструторе дочерний виджет, по ссылке на родительский даст последнему информацию о себе. Чтобы родительский виджет мог потом его удалить
- Если подчинённые виджеты добавляются сначала на компоновщик[1] (layout), а уже потом на основной виджет. То они автоматически становятся дочерними виджетами по отношению к основному виджету.
- Таким образом все виджеты находящиеся в окне, в конечном итоге агрегируются в основной виджет.

[1] компановщик отвечает за взаимное расположение виджетов внутри окна

Виджеты

```
QWidget w;

QPushButton *b = new QPushButton("PushMe");
QPushButton *b2 = new QPushButton("Show Label");
QLabel *l = new QLabel("I am Label");

// компоновщик
QVBoxLayout *layoyt = new QVBoxLayout();
layoyt->addWidget(b);
layoyt->addWidget(l);
layoyt->addWidget(b2);
w.setLayout(layoyt);

qDebug() << w.children(); // дочерние виджеты для w
// (QVBoxLayout(0xa8f4f0), QPushButton(0xd3e7c0),
// QLabel(0xc70c70), QPushButton(0xc728d0))
```

Основным виджетом может выступать либо пустой виджет, либо другие виджеты-контейнеры.
Например QTabWidget, QGroupBox и другие.



QWidget и компоновщики (layouts)

QWidget содержит свойства отвечающие за размер и положение элемента интерфейса пользователя, однако ручная работа с ними в большинстве случаев не рекомендуется.

За изменение размеров элемента интерфейса пользователя (виджетов) должен отвечать отдельный класс - компоновщик (layout), который будет автоматически менять ширину, высоту и положение виджета в зависимости от размеров окна.

В дизайнере форм можно задавать ограничения размера в контекстном меню виджета.

QWidget

Поля класса лучше всего изменять в дизайнере форм QtCreator'a.

При изменении свойств основного виджета (на котором расположены другие виджеты), аналогично изменяются и свойства всех дочерних. Так например можно изменить шрифт одновременно во всём окне.

По каждому из свойств доступна справка:
выделить свойство -> F1

Сигналы и обработчики событий

- Виджеты могут вызывать свои сигналы в ответ на некоторые события.
- Например в ответ на клик мышью, позиционирование курсора, появление, нажатие клавиши или изменение содержимого (если их содержимое может изменять пользователь)
- Эти сигналы вызываются автоматически
- При создании слотов (обработчиков событий) в дизайнере форм они автоматически соединятся с соответствующими сигналами
- Поэтому при вызове сигнала вызывается и слот

События QWidget

- При изменении некоторых свойств виджета вызываются *обработчики* соответствующего события.
- Некоторые обработчики событий

```
void QWidget::showEvent(QShowEvent *event)
void QWidget::hideEvent(QHideEvent *event)

// изменение размеров
void QWidget::resizeEvent(QResizeEvent *event);

// перерисовка виджета
void QWidget::paintEvent(QPaintEvent *event)

void QWidget::closeEvent(QCloseEvent *event)
```

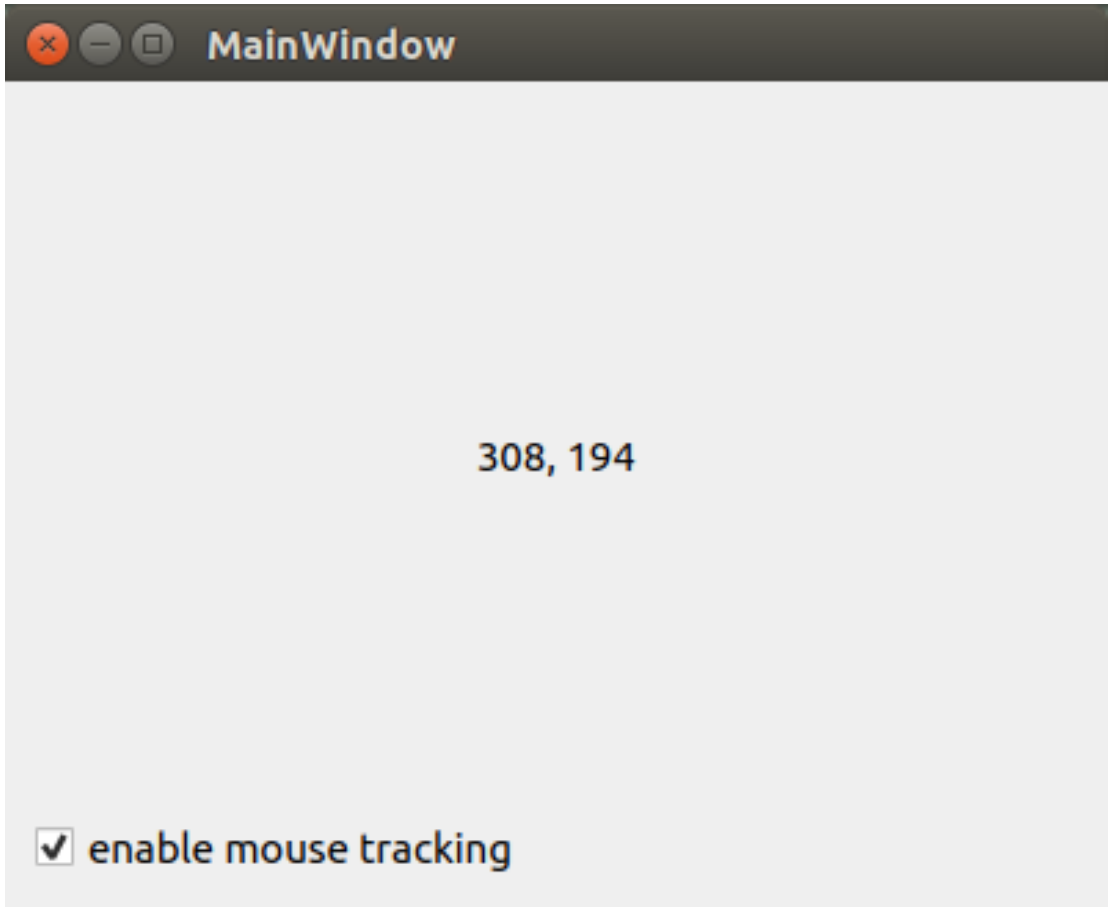
- По умолчанию обработчики не имеют реализации или не выполняют никакой работы, но их можно определить внутри класса.

Пример

Программа отображает позицию курсора если включено отслеживание мыши

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent), ui(new Ui::MainWindow){
    ui->setupUi(this);
// Включить отслеживание мыши для данного класса (главного окна)
      setMouseTracking(true);
//    Если это не сделать, то обработчик движения мыши будет
//    вызываться только если нажата одна из кнопок мыши
// Включить отслеживание мыши виджетом,
// хранящим всё содержимое главного окна
    ui->centralWidget->setMouseTracking(true);
    ui->checkBox->setChecked(
        ui->centralWidget->hasMouseTracking() );}
// обработчик события: движение мыши
void MainWindow::mouseMoveEvent(QMouseEvent *e){
    ui->label->setText( QString::number( e->x() ) + ", "
                + QString::number( e->y() ));}

void MainWindow::on_checkBox_stateChanged(int arg1){
```

Пример



События QWidget

Аналогично для событий генерируемых пользователем (клик или движение мыши, нажатие клавиши и т.д.) QWidget содержит виртуальные методы.

```
void QWidget::mouseMoveEvent(QMouseEvent *event)
void QWidget::mousePressEvent(QMouseEvent *event)
void QWidget::mouseDoubleClickEvent(QMouseEvent *event)

void QWidget::keyPressEvent(QKeyEvent *event)

void QWidget::contextMenuEvent(QContextMenuEvent *event)

void QWidget::dragEnterEvent(QDragEnterEvent *event)
```

При вызове обработчика ему передаётся в параметр объект описывающий соответствующее событие. Например координаты мыши или код нажатой клавиши.

События QWidget

Эти не специфические события для виджета нужно определять в производном от него классе вручную.

Обработчики для событий *других* виджетов, расположенных на данном окне создаются из контекстного меню конкретного виджета в дизайнере форм (go to slot...).

Outline

Основные элементы интерфейса пользователя

Часто используемые элементы интерфейса пользователя представлены классами:

- QLabel - надпись, также может отображать картинку;
- QLineEdit - однострочное текстовое поле ввода;
- QTexEdit - многострочное поле ввода;
- QSpinBox, QDoubleSpinBox - числовое поле ввода для целых и вещественных чисел соответственно
- QRadioButton - переключатель (позволяет выбор одного из нескольких вариантов)
- QCheckBox - флажок (галочка)
- QComboBox - Комбинированный список

Все эти элементы интерфейса могут быть использованы в Дизайнере форм Qt Creator.

Основные элементы интерфейса пользователя

- QListWidget - список
- QTableWidget - таблица
- QChartView - компонент для отображения графиков
- QGraphicsView - компонент для отображения графики
- QOpenGLWidget - компонент для рисования с помощью OpenGL
- QTextBrowser - текстовый браузер

Иерархия наследования

Свойство	Значение
QObject	
objectName	spinBox_n
QWidget	
QAbstractSpinBox	
wrapping	☐
frame	☑
alignment	AlignLeft, AlignVCenter
readOnly	☐
buttonSymbols	UpDownArrows
specialValueText	
accelerated	☐
correctionMode	CorrectToPreviousValue
keyboardTracking	☑
showGroupSeparator	☐
QSpinBox	
suffix	
prefix	
minimum	0
maximum	99
singleStep	1
value	3
displayIntegerBase	10

Все классы имеющие отношение к графическому интерфейсу построены на основе QWidget.
Это хорошо видно в разделе свойств объекта в дизайнере QtCreator.

QLabel

```
QString text() const; // Получение текста QLabel
void setText(const QString &); // Получение текста QLabel
```

QLabel

изображения в QLabel

```
// Загрузка изображения
// здесь изображение прикреплено к проекту
QPixmap pixmapTarget = QPixmap(":/image/icon/target.png");

// Масштабирование изображения
pixmapTarget = pixmapTarget.scaled(size-5, size-5,
        Qt::KeepAspectRatio, Qt::SmoothTransformation);


ui->label_image_power->setPixmap(pixmapTarget);
```

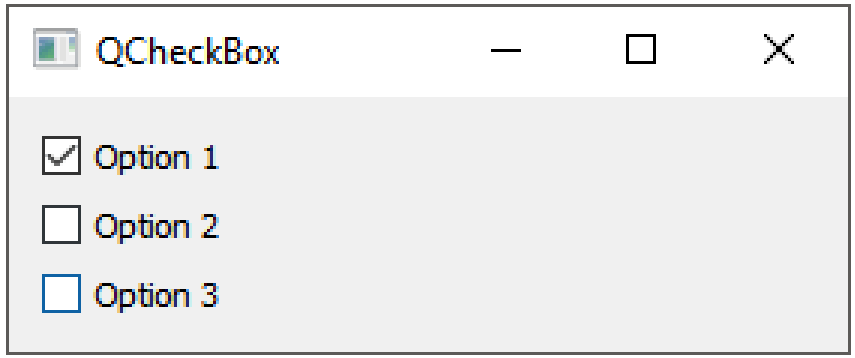
QPushButton

Сигналы

- `void clicked()`
- `void pressed()`
- `void released()`

Так же может быть полезным сделать кнопку неактивной с помощью метода

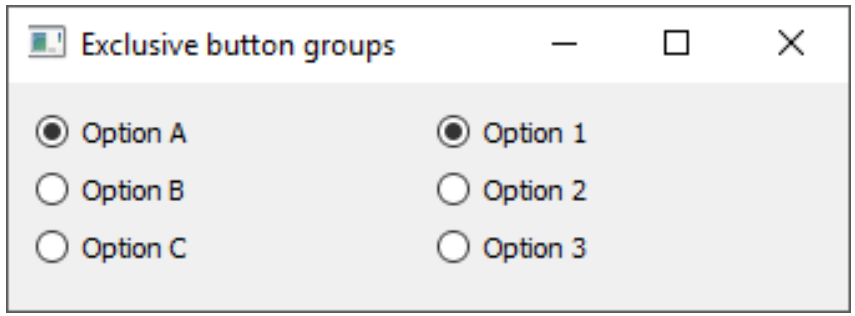
QCheckBox



Основное свойство флажка QCheckBox - state.

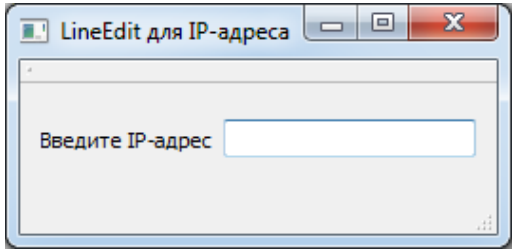
```
// получить состояние
bool isChecked() const
// задать состояние
void setChecked(bool)

// при изменении состояния вызывается сигнал
void QCheckBox::stateChanged(int state)
```

QRadioButton



- Переключатель QRadioButton также как и флажок может находится в двух состояниях
- Для проверки и задания состояния используются те же методы что и у QCheckBox
- На одной панели может быть нажат только один переключатель.
- Чтобы объединить переключатели на одну панель достаточно поместить их в один компоновщик. При нажатие на один переключатель, остальные будут отключатся

QLineEdit



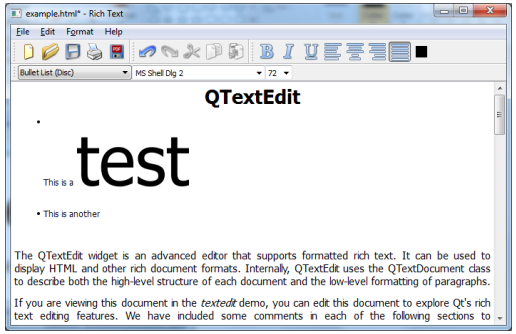
QLineEdit - однострочное поле ввода.

Методы

```
// получить содержимое
QString text() const
// задать содержимое
void setText(const QString &)
// сигнал вызываемый при изменении содержимого
void textChanged(const QString &text)
// сигнал вызываемый при нажатии на Enter
void QLineEdit::returnPressed()
```

Возможна проверка введённого текста по маске с помощью класса QValidator

QTextEdit



QTextEdit - многострочное поле ввода с поддержкой rich text и html, в том числе может содержать изображения в тексте

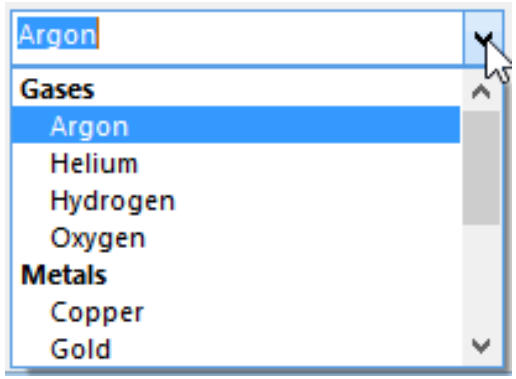
Методы

```
// получить содержимое в виде обычного текста
QString toPlainText() const
// задать текст
void setPlainText(const QString &text)

// получить содержимое в виде html
QString toHtml() const
// задать содержимое в виде html
void setHtml(const QString &text)
```

QTextEdit

- Для запись в файл содержимого QTextEdit используется класс QTextDocument
- Текст в представлении класса QTextDocument можно получить через свойство QTextDocument
- QTextDocument может сохранять документы в ODF и HTML формате
- QTextEdit поддерживает Drag and Drop, в том числе можно перетащить в это поля ввода изображение
- Для отображения простого текста вместо QTextEdit используется QPlainTextEdit

QComboBox



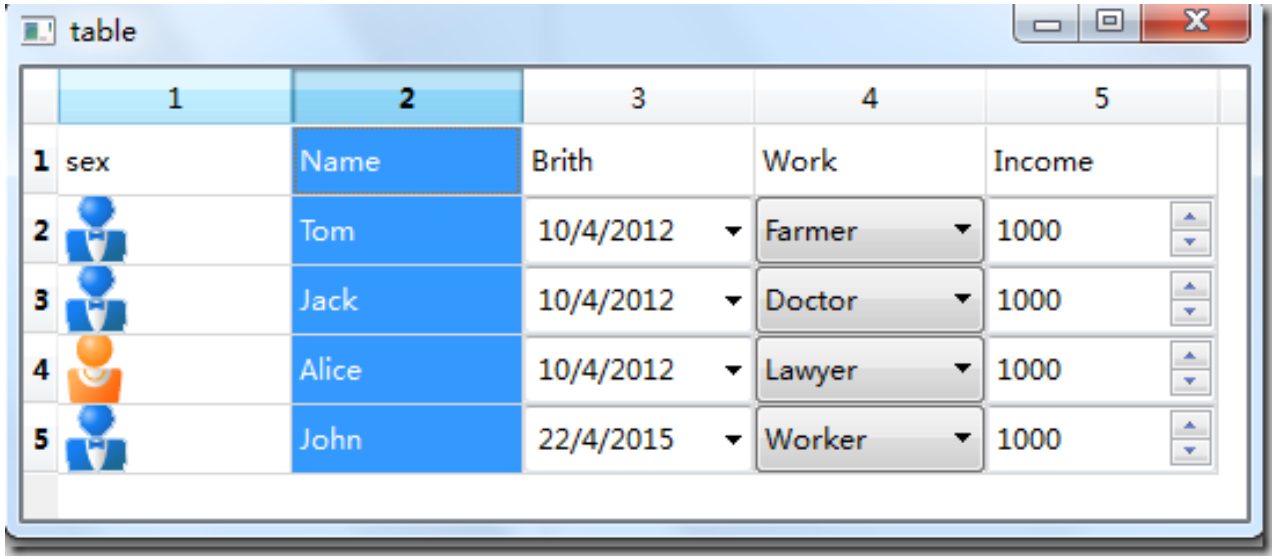
doc.qt.io/qt-5/qcombobox.html

QStatusBar

QListWidget

doc.qt.io/qt-5/qlistwidget.html

QTableWidget



- Таблица
- Может содержать редактируемый текст и виджеты
- Может быть связана с классом отвечающим за хранение данных (потомков QAbstractItemModel), таким образом хранение и представление данных будет разделено

https://doc.qt.io/qt-5/qtablewidget.html

QTableWidget

```
// задание числа строк
void QTableWidget::setRowCount(int rows)
// получение числа строк
int QTableWidget::rowCount() const
// задание числа столбцов
QTableWidget::setColumnCount(int columns)
// получение числа столбцов
int QTableWidget::columnCount() const
```

Если таблица будет иметь фиксированное число столбцов, каждый из которых имеет заголовок, то задать их удобнее в дизайнере форм

QTableWidget

- Каждая ячейка - объект.
- Для задания содержимого ячейки требуется создать объект
- Добавить его в таблицу

```
void QTableWidget::setItem(int row, int column,
                           QTableWidgetItem *item)
```

- После добавления экземпляра класса QTableWidgetItem в таблицу, она становится владльцем этого объекта
- Получить ячейку

```
QTableWidgetItem *QTableWidget::item(int row,
                                     int column) const
```

QTableWidget

Задание содержимого ячейки

```
for (int i=0; i<ui->tableWidget->rowCount(); i++)
    for (int j=0; j<ui->tableWidget->columnCount(); j++){
        int num = rand() % 100;
        QTableWidgetItem *item =
            new QTableWidgetItem( QString::number(num) );
        ui->tableWidget->setItem(i,j, item);
        }
```

- Необходимо создавать объекты для представления ячейки
- Если тип объекта не поменяется, то это должно происходить только один раз. Например в конструкторе окна, где расположена таблица.
- QTableWidget не предназначен для хранения данных, а предназначен для их отображения. Для хранения стоит предусмотреть свою структуру данных.

QTableWidget

Изменение содержимого ячейки

```
for (int i=0; i<ui->tableWidget->rowCount(); i++)
    for (int j=0; j<ui->tableWidget->columnCount(); j++){
        int num = rand() % 100;
        QTableWidgetItem *item = ui->tableWidget->item(i,j);
        item->setText( QString::number(num) );
    }
```

QTableWidgetItem

QTableWidgetItem - класс представляющий ячейку таблицы QTableWidget

```
// конструктор со строковым параметром
QTableWidgetItem(const QString &text, int type = Type)

// получение текстового содержимого
QString QTableWidgetItem::text() const
// задание текстового содержимого
void QTableWidgetItem::setText(const QString &text)

// задание шрифта
void QTableWidgetItem::setFont(const QFont &font)

// получение номера строки и столбца
int QTableWidgetItem::row() const
int QTableWidgetItem::column() const

// задание фона
void QTableWidgetItem::setBackground(const QBrush &brush)
```

QTableWidget

Виджеты в ячейках

```
// создание виджетов

for (int i=0; i<ui->tableWidget->rowCount(); i++){
    QCheckBox *cb =
            new QCheckBox( "checkbox #" + QString::number(i) );
    ui->tableWidget->setCellWidget(i,3, cb);

    QSpinBox *sp = new QSpinBox();
    sp->setValue( rand()%100 );
    ui->tableWidget->setCellWidget(i,1, sp);
}
```

QTableWidget

Виджеты в ячейках

```
// обращение к виджетам

for (int i=0; i<ui->tableWidget->rowCount(); i++){
  QCheckBox *cb =
    qobject_cast<QCheckBox*>( ui->tableWidget->cellWidget(i,3) );
  if (cb!=NULL)
    cb->setChecked( rand()% 2);

  QSpinBox *sp =
    qobject_cast<QSpinBox*>( ui->tableWidget->cellWidget(i,1) );
  if (sp!=NULL)
    sp->setValue( rand()%100 );
}
```

Outline

Элементно-ориентированный подход

- Для представления данных в табличном виде пользователю в Qt можно использовать виджет QTableWidget
- Этот виджет хранит и показывает данные
- QTableWidget строится на основе другого виджета - QTableView
- QTableWidget реализует **элементно-ориентированный подход**

Элементно-ориентированный подход

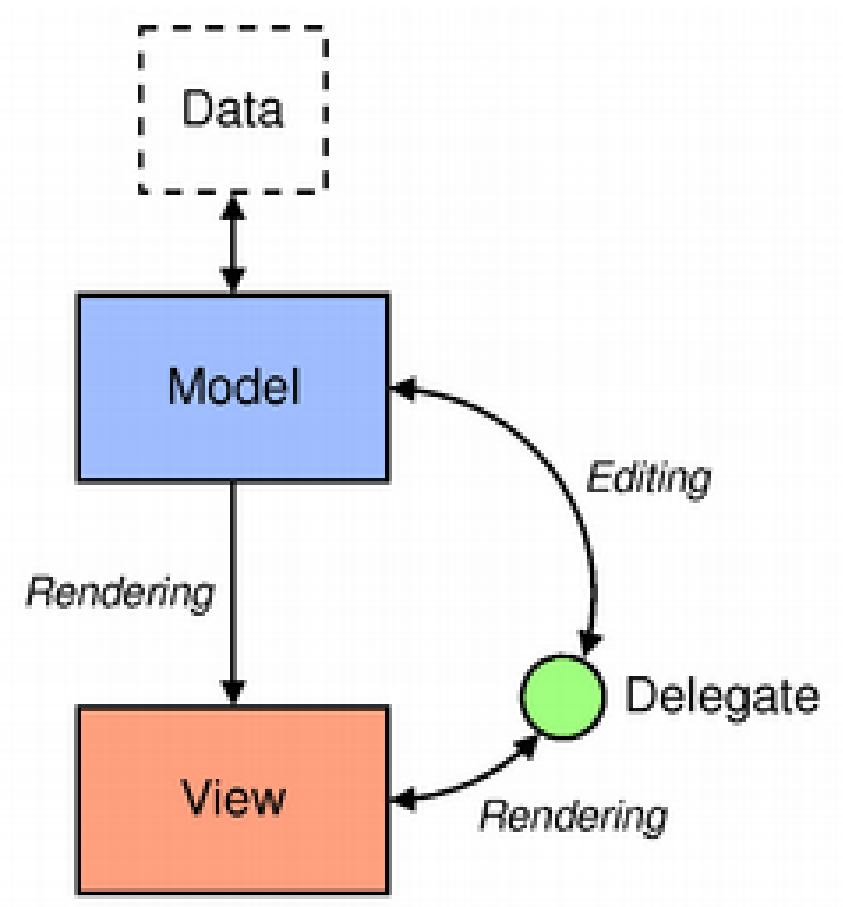
Недостатки

- Если данные должны быть представлены в нескольких виджетах, то будет создано несколько копий этих данных
- К тому же придётся следить за синхронизацией данных в разных виджетах
- Поэтому использование QTableWidget рекомендуется, когда данных не много и с ними не приводится сложных операций

Модель и представление

- Виджет QTableView способен *только представлять* (показывать и позволять редактировать) данные для пользователя
- Поэтому совместно с QTableView используется **модель**
- Модель хранит и обрабатывает данные
- Модель в Qt строится на основе класса QAbstractItemModel
- Такой подход подход называется **модель-представление**

Модель и представление



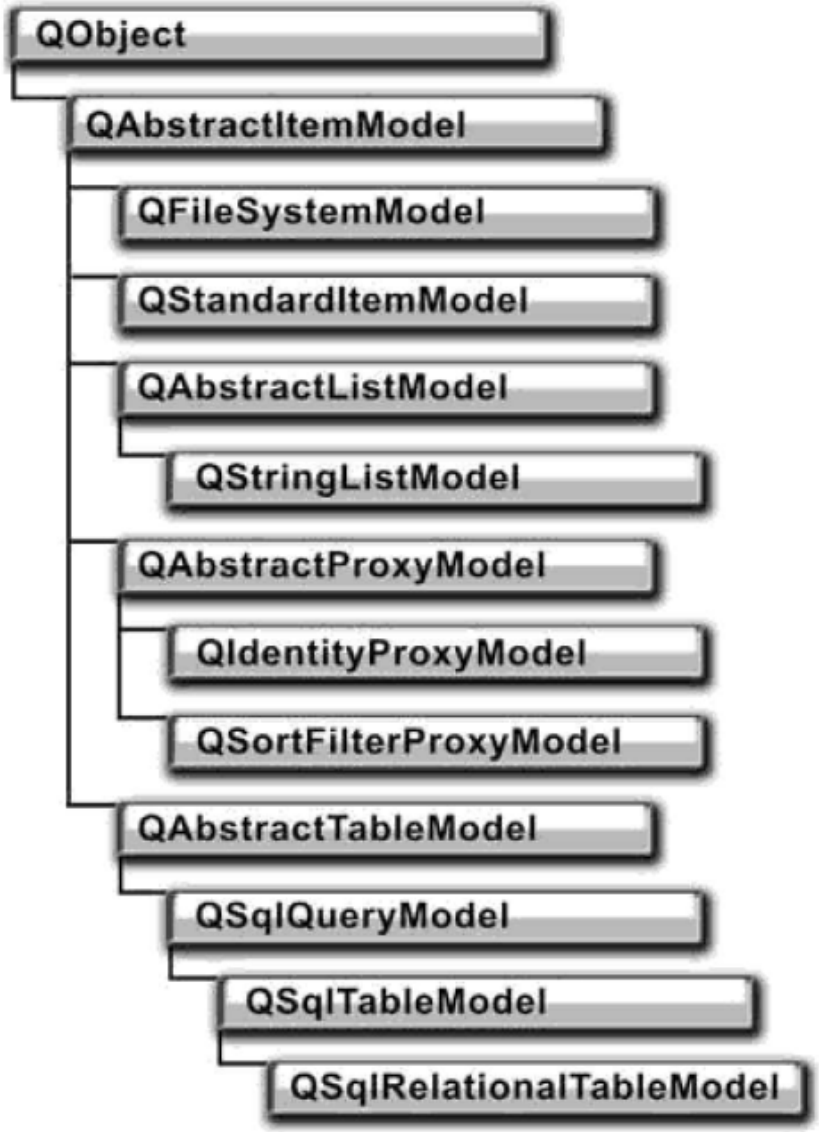
Подобные подходы используются во многих фреимворках для разных зыков программирования

Модель

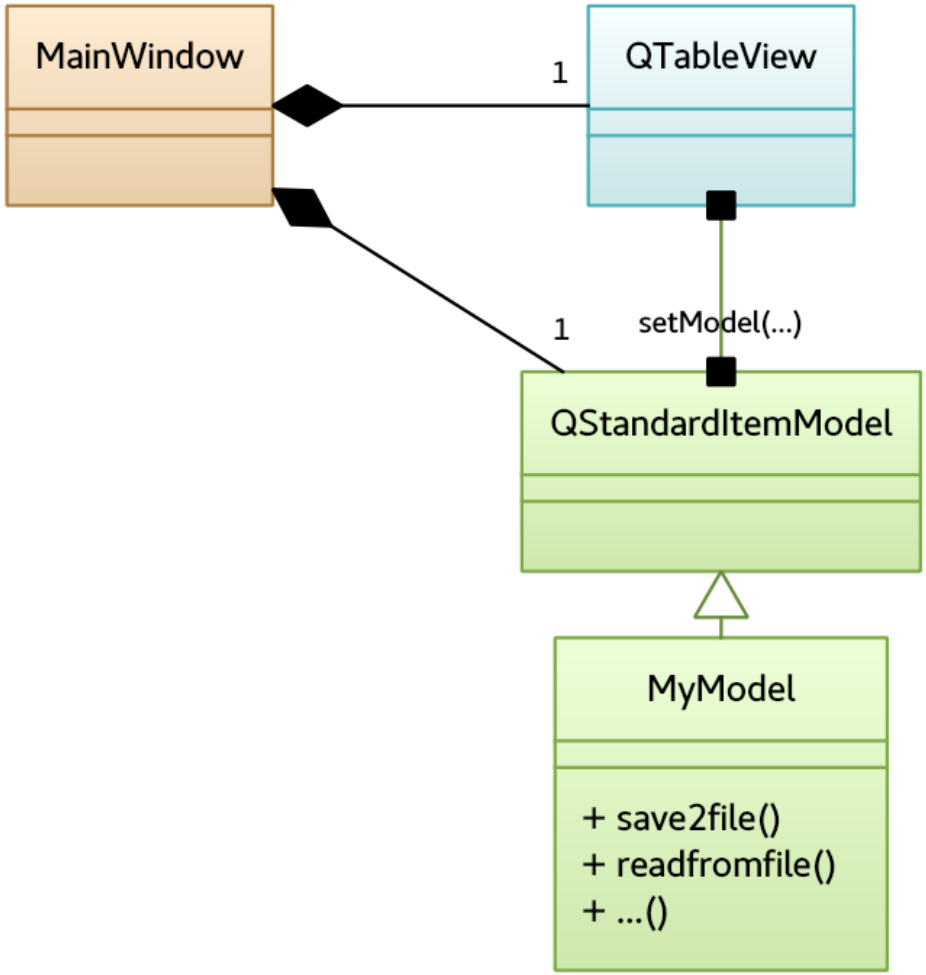
- Модель должна строится на основе класса QAbstractItemModel или его потомков
  (стоит помнить о необходимости реализации абстрактных методов)
- В классе модели должна быть описана бизнес-логика и другие методы
  Например сохранение и загрузка данных в файл
- Потомки этого класса могут использоваться и самостоятельно[2]
  - QStandardItemModel (наиболее общий класс из приведенных в этом списке)
  - QStringListModel
  - QDirModel
  - QFileSystemModel
  - QSqlQueryModel

[2] evileg.com/ru/post/158/

Классы моделей

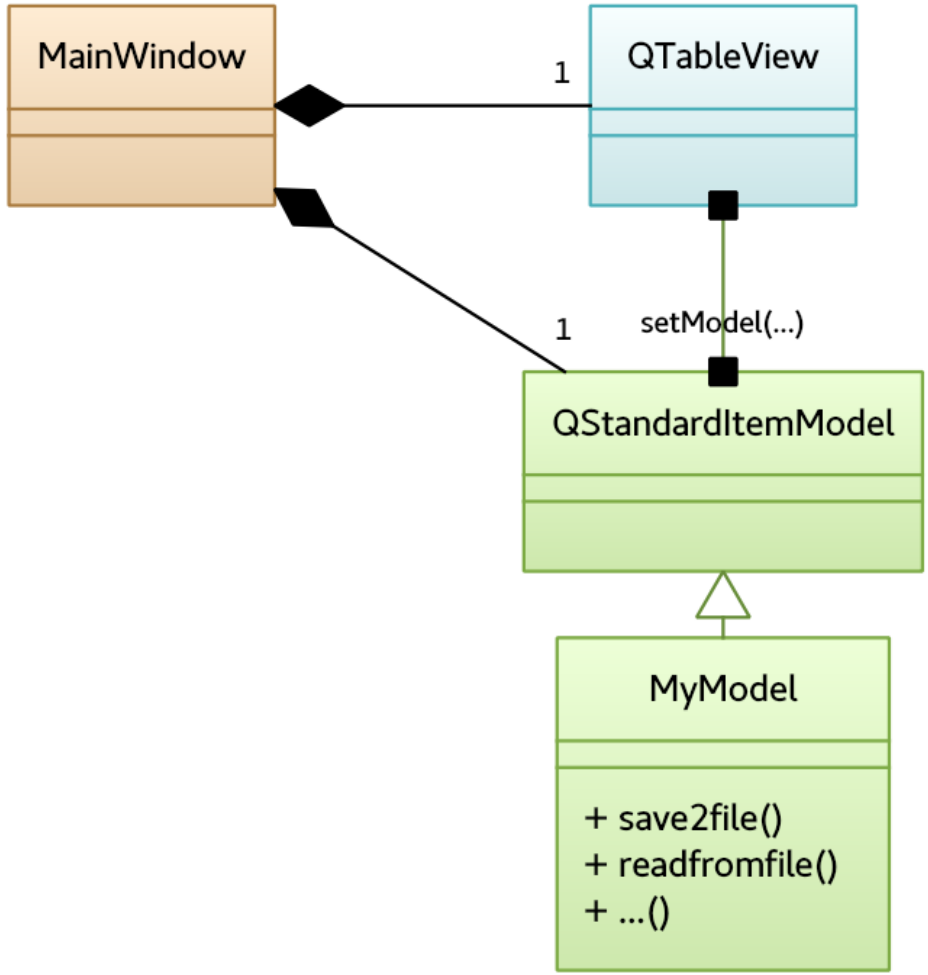


Приложение



Пример диаграммы классов приложения. Для модели создан отдельный класс на основе QStandardItemModel.

Приложение



Пример диаграммы классов приложения. Для модели создан отдельный класс на основе QAbstractModel. Это более гибкий подход, но QAbstractModel представляет только интерфейс.

Модель и представление

- Виджет QTableView и класс модели QAbstractItemModel связываются с помощью метода

```
void QTableView::setModel(QAbstractItemModel *model)
```

- Все изменения на QTableView сделанные пользователем сразу отражаются на модели
- Аналогично любое изменение данных в модели QAbstractItemModel автоматически обновляет виджет QTableView

Модель - представление

Пример

- Далее приводится примеры кода программы - файловой базы данных.
- Столбцы таблицы: имя контакта, номер телефона
- Для представления данных используется QTableView
  `ui->tableView`
- В качестве модели - QStandardItemModel
  объявлен в классе главного окна как
  `QStandardItemModel *model`

Модель - представление

Пример: связывание модели и таблицы, заполнение модели

```
// пример создания модели и заполнение её данными

// создание модели. Как правило это происходит один раз
model = new QStandardItemModel(this);
// модель сразу же ассоциируется с представлением - виджетом
ui->tableView->setModel(model);

// запишем данные в модель...
QList<QStandardItem *> items_list;
// QList - одна строка таблицы
// QStandardItem - один элемент в строке таблицы
items_list.append( new QStandardItem("Лёха (должен 100 рублей)") );
items_list.append( new QStandardItem("+79991234567") );
// добавим строку с данными в модель
model->appendRow( items_list );

// виджет теперь показывает добавленные данные
```

Модель - представление

Пример: обращение к ячейкам таблицы

```
// пример получения данных из модели

QList<QStandardItem *> items_list2;
QStringList str_list;  // сохраним данные сюда

// прочитаем первую строку
for (int j = 0; j < model->columnCount(); j++){
    // цикл по элементам строки
    QStandardItem *it = model->item(0, j);
    items_list2.append(it);
    // можно получить текстовые значения ячеек
    QString cell;
    if (cell != nullptr)
        cell = it->text();
    // при необходимости данные сохранятся в список из QString
    // например для последующей записи в текстовый файл
    str_list.append(cell);
        }
```

Модель - представление

Пример

- Пользователь может редактировать данные прямо в таблице tableView
- Для добавления данных пользователем в таблицу стоит предусмотреть например кнопку
- При нажатии кнопки в модель будет добавляется строка аналогично слайду 61 только с пустыми строками в каждой ячейке

- Код по работе с моделью следует поместить в методы отдельного класса
- Сам класс создать унаследовавшись от QStandardItemModel

Модель

Пример: удаление выделенных строк

```
// удаление выделенных строк из  QStandardItemModel

 QModelIndexList rows = // получим список выделенных строк
    this->ui->tableView->selectionModel()->selectedRows();

    for (QModelIndex i: rows){
        this->model->removeRow(i.row());
    }
```

Если для модели используется класс QAbstractItemModel то метод удаления removeRow потребуется реализовать самостоятельно

Модель

Пример: сортировка и поиск

```
// сортировка по второму столбцу
    this->model->sort(1);


// поиск вхождения подстроки по всем элементам первого столбца
    QList<QStandardItem *> items =
        this->model->findItems("Лёха", Qt::MatchContains, 0);
    if (items.size()){
        // выделение строки с первым найденым результатом
        this->ui->tableView->selectRow(items.at(0)->row());
    }
```

В этом примере элементы будут сортироваться как строки

Модель - представление

Документация

https://doc.qt.io/qt-5/model-view-programming.html - Model/View Programming

Outline

Основные классы

- Qt содержит множество классов как для создания элементов интерфейса пользователя, так и для хранения данных, работы с сетью, изображениями и т.п.
- Классы используемые для хранения данных совместимы с аналогичными из STL и во много похожи на них
- Во многом классы из Qt удобнее для программиста, чем классы из STL

Рекомендации

- Перед решением задачи и написанием собственного кода следует проверить документацию на наличие подходящих классов.
- Перед использованием класса следует познакомится с его документацией.
- Если не существует готовых решений, то следует изучить лучшие практики (best practice).

Справка Qt

F1 - вызов справки по классу (или функции), на который установлен курсор.

Справка по классу обычно состоит из

- общего описания класса
- Properties - списка свойств (полей класса и методов доступа к ним),
- Public Functions - открытых методов,
- Public Slots - открытых методов, которые вызываются в ответ на события.
- Закрытых методов
- Detailed Description - подробного описания класса, в котором могут быть приведены примеры его использования.

Основные классы Qt

Когда использовать STL, а когда аналогичные классы Qt?

Проблема бананов, обезьян и джунглей

Проблема с ОО-языками заключается в том, что они тянут за собой всё своё окружение. Вы хотели всего лишь банан, но в результате получаете гориллу, держащую этот банан, и все джунгли в придачу.

–Джо Армстронг, создатель Erlang

Когда использовать STL, а когда аналогичные классы Qt?

- В модулях приложения, которое и так использует Qt
  Например класс главного окна
- В модулях, которые потенциально не будут использованы вне Qt приложений
- Везде, где выгода от использования именно Qt классов превосходит недостатки из-за проблемы бананов, обезьян и джунглей
- При использовании классов Qt конечный размер приложения может сильно вырасти из-за необходимости распространять его с dll (so файлами на Linux) файлами Qt.

Core classes

Файл проекта:

```
QT += core
```

Почти все Qt классы (не только основные) содержатся в одноимённых заголовочных файлах.
Например QPoint:

```
##include <QPoint>
```

Заголовочные файлы в некоторых других классов (по большей части это разного рода виджеты) могут находится в отдельных каталогах фреимворка:

```
##include <QtWidgets/QLabel>
```

Заголовочный файл и модуль Qt указывается для каждого класса в документации.

Core classes

Классы предназначение для работы с данными, файловой системой, временем, исключением и т.п. содержатся в ядре фреймворка - модуле core.

Некоторые из **core classes**

- Для хранения данных: `QString` `QVector`, `QStringList`, `QStack`, `QSet`, `QPair`, `QMap`, `QList`, `QSize`, `QRect`, `QPoint`
- Для представления размеров и положения: QSize QRect QPoint
- Для для работы со временем QTime QDate
- ТаймерQTimer
- Исключения QException
- Регулярные выражения QRegExp
- Работа с URL QUrl
- Пути, папки файлы: QDir, QTextStream, QFile
- Логирование QMessageLogger

QString

Класс для хранения строк в Unicode кодировке.

```
QString str;

// Число -> строка
str.setNum(1234);          // str == "1234"|

// Строка -> число
float x = str.toFloat();

// Число -> строка. Статический метод
QString s = QString::number(42.05)

// Возвращает стробу без пробелов в начале и конце
str = str.trimmed();

// в std::string
std::string s = str.toStdString()
```

QDir

Информация о каталогах и их структуре.

```
// Упрощает работу с путями к файлам
QDir directory("Documents/Letters");
QString path = directory.filePath("contents.txt");
QString name = directory.dirName();
QString absPath = directory.absoluteFilePath("contents.txt");

// текущая папка
QDir cdir = QDir::current();
QString cdir_path = QDir::currentPath();
// Папка пользователя
QDir home =  QDir::home();
```

QDir

```
// проверка на существование
if ( dir.exists() ){... }

// Список имён файлов
QStringList dd = d.entryList();

// сменить папку (текущая папка для прогр. не меняется)
home.cd("another-dir");

// изменить текущую папку программы
if ( QDir::setCurrent("another-dir") ){
    // папка поменялась
}
```

QFile

Чтение и запись данных в файлы.

```
QFile f("myfile");

if ( !f.open(QFile::WriteOnly) )
    // не удалось открыть файл ;
;
char *data = "some data 12345 \n";
f.write(data, strlen(data));
f.close();

if ( !f.open(QFile::ReadOnly) )
    // не удалось открыть файл ;

char buf[1024];
qint64 r = f.readLine(buf, 1024);
if ( r!=-1 )
    // прочитано r байт
```

QTextStream

Упрощает работу с текстовыми файлами.

```
// создаём класс-файл
QFile data("output.txt");
// Открываем файл для записи
if (data.open(QFile::WriteOnly)) {

    // Для удобства записи разных типов данных
    // в текстовый файл
    // используем этот класс
    QTextStream out(&data);

    out << "Result: " << qSetFieldWidth(10) << left << 3.14
    // writes "Result: 3.14      2.7       "
    }
```

QTextStream

Упрощает работу с текстовыми файлами.

```
QFile data("output.txt");
if (data.open(QFile::ReadOnly)) {
    QTextStream in(&data);
    QString str = in.readLine();
    // уберём лишние пробелы в начале и в конце
    str = str.trimmed();
    // заменим повторяющиеся пробелы на один
    unsigned n;
    do{ n = str.length();
        str = str.replace("  ", " ");
    } while (n!=str.length());

    // разделим строку по пробелам
    QStringList sl = str.split(" ");
    x = sl[1].toFloat();
    x = sl[3].toFloat();
```

QTimer

```
QTimer *timer = new QTimer;
QObject::connect(timer, &QTimer::timeout,
                 [](){qDebug() << ".";});
timer->setInterval(1500); // в миллисекундах
timer->start();


// Таймер с одиночным срабатыванием
QTimer::singleShot(200, объект, SLOT(метод));
// после срабатывания будет вызван
// метод указанного объекта
```

В примере использована лямбда функция, но соединить сигнал таймера timeout можно с любой функцией или методом.

Outline

Другие классы

- QNetworkAccessManager - работа с сетевыми запросами
  https://ru.stackoverflow.com/questions/516754

Ссылки и литература

- Qt Википедия
- OpenSource версия
- Qt wiki

Книги:

- Qt 5.X. Профессиональное программирование на C++. Макс Шлее. 2015 г. 928 с. Книга периодически обновляется с выходом новых версий фреймворка Qt.
- Qt. Профессиональное программирование. Разработка кроссплатформенных приложений на C++. Марк Саммерфилд