ЛАБОРАТОРНА РОБОТА №10

Тема: Механізми сигналів і слотів. Класи QWidget та QDialog.

Мета: Навчитись застосовувати механізм сигналів і слотів фреймворку Qt, розробляти програмне забезпечення з використанням класів QWidget та QDialog.

КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Клас QWidget є базовим для всіх об'єктів користувальницького інтерфейсу.

Віджет - це елементарний об'єкт призначеного для користувача інтерфейсу: він отримує події миші, клавіатури та інші події від віконної системи і малює своє зображення на екрані. Кожен віджет має прямокутну форму, і всі вони відсортовані в порядку накладення (Z-order). Віджет обмежений своїм батьком і іншими віджетами, розташованими перед ним.

Віджет, що не вбудований в батьківський віджет, називається вікном. Зазвичай вікно має рамку і смугу з заголовком, хоча використовуючи відповідні прапори вікна можна створити вікно без такого зовнішнього оформлення. В Qt QMainWindow і різноманітні підкласи QDialog є найбільш поширеними типами вікон.

Кожен конструктор віджета приймає один або два стандартних аргументи:

QWidget * parent = 0 є батьком нового віджету. Якщо він дорівнює 0 (за замовчуванням), новий віджет буде вікном. Якщо немає, то він буде нащадком для parent, і буде підкорятися геометрії батька parent (за винятком випадку, коли ви призначите йому прапор Qt :: Window , як прапор вікна).

Qt :: WindowFlags f = 0 (там, де це можливо) встановлює прапори вікна; значення за замовчуванням підходить майже для всіх віджетів, але, наприклад, щоб отримати вікно без рамки, передбаченої системою, ви повинні використовувати спеціальні прапори.

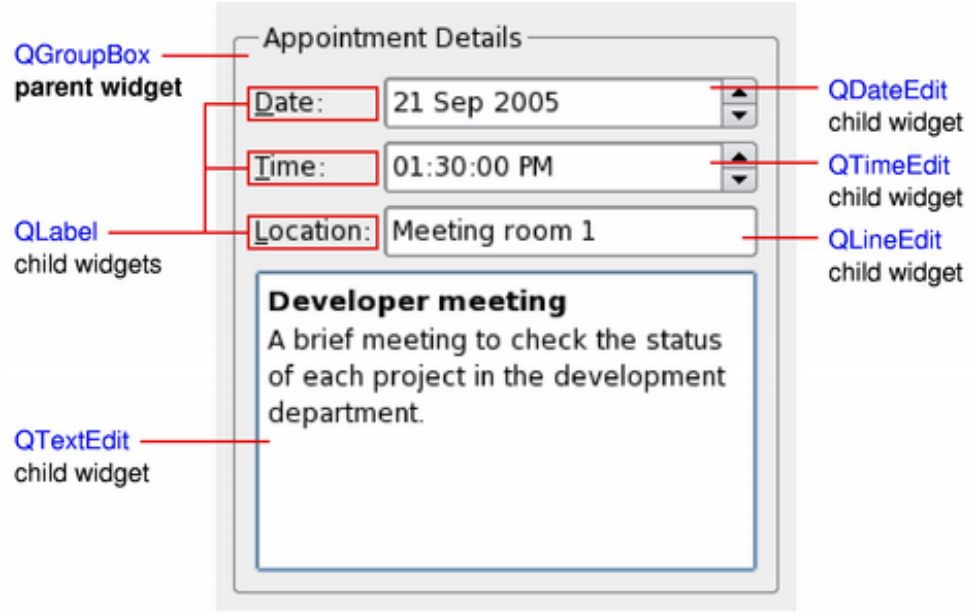
QWidget має багато функцій, але більшість з них не має безпосереднього застосування; наприклад, QWidget має властивість font, але ніколи не використовує його для самого себе. Існує багато підкласів, які володіють реальною функціональністю, наприклад, QLabel , QPushButton , QListWidget і QTabWidget .

Віджети верхнього рівня і віджети-нащадки

Віджет без батьківського віджета завжди є незалежним вікном (віджетом верхнього рівня). Для таких віджетів setWindowTitle () і setWindowIcon () встановлюють заголовок вікна і іконку.

Віджети, які не є вікнами, є дочірніми віджетами свого батьківського віджета. Більшість віджетів в Qt використовуються в основному як дочірні віджети. Наприклад, можна відобразити кнопку як віджет верхнього рівня, але більшість користувачів вважає за краще поміщати кнопку на інші віджети, такі як QDialog .

Батьківський віджет містить в собі різні дочірні віджети.



На діаграмі вище показаний віджет QGroupBox , який буде використовувати різні дочірні віджети, скомпоновані за допомогою QGridLayout . Рамкою у дочірніх віджетів QLabel показані їх справжні розміри.

Якщо ви хочете використовувати QWidget для зберігання дочірніх віджетів, це означає вам потрібно додати компоновщик в батьківський віджет. Для отримання додаткової інформації дивіться Управління компонуванням .

Комбіновані віджети

Якщо віджет використовується як контейнер, який об'єднує кілька дочірніх віджетів, він називається комбінованим віджетом. Такий віджет може бути створений на основі віджета, який має деякі візуальні властивості - як, наприклад, QFrame , - а його дочірні віджети розмістити в ньому за допомогою компоновщика. Вище на малюнку показаний комбінований віджет, створений за допомогою Qt Designer .

Комбінований віджет може бути створений шляхом створення підкласу від стандартного віджету, такого як QWidget або QFrame , і додавання необхідних компонувальник і дочірніх віджетів в конструктор підкласу. Багато з прикладів, що поставляються разом з Qt , використовують цей підхід, як і ті, що даються в Навчальних посібниках Qt.

Призначені для користувача віджети і відображення

Так як QWidget є підкласом QPaintDevice, його підкласи можуть самі відображати свій вміст, що зручніше, ніж було раніше, коли використовувалися операції малювання разом з екземпляром класу QPainter . Цей підхід відрізняється від методу полотна, що використовує Каркасом графічного представлення , де додаток додає елементи в область відображення, і вони малюють себе самі.

Кожен віджет виконує всі операції з малювання зі своєї функції paintEvent (). Вона викликається, коли виникає необхідність перемалювати віджет, через деяких зовнішніх змін або на вимогу програми.

Перевага розміри і політика розмірів

Коли створюється новий віджет, майже завжди корисно перевизначити sizeHint () для забезпечення бажаних розмірів віджету за замовчуванням, а також встановити коректну політику розміру в допомогою setSizePolicy ().

За замовчуванням, визначення розмірів комбінованих віджетів, у яких не визначені кращі розміри, буде відбуватися відповідно до простором, яке потрібно його дочірнім віджетів.

Політика розмірів дозволяє вам забезпечити хорошу поведінку для системи компоновки таким чином, що інші віджети можуть містити і легко управляти вашим віджетом. За замовчуванням політика розмірів має на увазі, що найкраще, якщо більшість віджетів матиме свій бажаний розмір.

Події

Віджети отримують події, які породжуються типовими діями користувача. Qt посилає події віджету через виклики спеціальних функцій обробників подій з аргументом у вигляді підкласу від QEvent , що містить інформацію про подію.

Якщо ваш віджет є всього лише контейнером для дочірніх віджетів, то вам, швидше за все, не знадобиться реалізовувати ніяких обробників подій. Якщо вам потрібно відловити клацання миші в дочірньому віджеті, викличте його функцію underMouse () всередині функції mousePressEvent () віджета.

Вам потрібно буде надати поведінку і вміст для ваших віджетів, проте ось короткий огляд подій, які відносяться до QWidget, починаючи з найпоширеніших:

- paintEvent () виникає, коли є необхідність в перемальовуванні віджета. Будь-віджет, що відображає призначений для користувача контент, повинен його реалізувати такий обробник. Перемальовування, що використовує QPainter , може відбуватися тільки в paintEvent () або у функціях, що викликаються з paintEvent ().
- resizeEvent () виникає, коли віджет змінив розміри.
- mousePressEvent () виникає, якщо натиснута кнопка миші в той момент, коли миша перебувала всередині віджета, або якщо він захоплений мишкою за допомогою grabMouse (). Натискання кнопки миші без відпускання - це фактично те ж саме, що і виклик grabMouse ().
- mouseReleaseEvent () виникає, коли кнопка миші відпускається. Віджет отримає подія відпускання кнопки миші, якщо до цього він отримав подія натискання кнопки миші. Це означає, що Услі користувач натиснув кнопку миші всередині вашого віджета, потім перемістив мишку куди-небудь і там відпустив кнопку, то ваш віджет отримає подія відпускання миші. Існує один виняток: якщо при натисканні кнопки з'являється спливаюче меню, воно негайно перехопить події від миші.
- mouseDoubleClickEvent () виникає, коли користувач двічі клацне по віджету. Якщо користувач використовував подвійне клацання, віджет отримає подія натискання кнопки, подія відпускання кнопки і на закінчення подія про другому натисканні. (Кілька подій про переміщення миші можуть прийти, якщо користувач під час операції не втримав мишку на місці.) Неможливо відрізнити одинарний клацання від подвійного поки не прийде другий клацання. (Це одна з причин, чому багато книги про розробку користувальницького інтерфейсу

рекомендують, щоб подвійне клацання був краще, ніж одинарний, при перемиканні між різними операціями.)

У віджетів, які дозволяють введення з клавіатури, потрібно перевизначити трохи більше функцій:

- keyPressEvent () виникає, коли клавіша була натиснута, і ще раз, коли клавіша утримується для автоповтора. Натискання клавіші Tab і поєднання Shift + Tab передається віджету тільки, якщо він не використовує механізм зміни фокусу. Для перехоплення натискання цих клавіш ви повинні перевизначити QWidget :: event ().
- focusInEvent () виникає, коли віджет отримує фокус введення з клавіатури (якщо ви викликали setFocusPolicy ()). Добре написані віджети показують, що вони отримали фокус введення з клавіатури в ясній і простій формі.
- focusOutEvent () виникає, коли віджет втрачає фокус введення з клавіатури.

Детальніше про QWidget: http://doc.crossplatform.ru/qt/4.6.x/qwidget.html

СИГНАЛИ І СЛОТИ

Механізм сигналів і слотів є розширенням мови програмування C ++ , який використовується для встановлення зв'язку між об'єктами. Якщо відбувається якась певна подія, то при цьому може генеруватися сигнал. Даний сигнал потрапляє в пов'язаний з ним слот. У свою чергу, слот - це звичайний метод в мові C++, який приєднується до сигналу; він викликається тоді, коли генерується пов'язаний з ним сигнал. Як бачите, нічого складного тут немає.

Всі графічні додатки управляються подіями: все, що відбувається в додатку є результатом обробки тих чи інших подій. Вони є важливою частиною будь-якої графічної програми. У більшості випадків події генеруються користувачем додатку, але вони також можуть бути згенеровані і іншими засобами, наприклад, підключенням до Інтернету, віконним менеджером або таймером.

У моделі подій є 3 учасники :

- джерело події - це об'єкт, стан якого змінюється;
- об'єкт події - це відстежується параметр джерела події (наприклад, натискання клавіші на клавіатурі або зміна розмірів віджету);
- мета події - це об'єкт, який повинен бути повідомлений про подію, що відбулася.

Не потрібно плутати сигнали з подіями. Сигнали необхідні для організації взаємодії між віджетами, тоді як події необхідні для організації взаємодії між віджетом і системою.

Клік миші

У даному прикладі ми розглянемо простий спосіб обробки подій. У нас є одна кнопка, одним помахом мишкою на яку ми завершуємо роботу програми.

Заголовки - click.h:

```
#pragma once
#include <QWidget>
```

```
class Click : public QWidget {
  public:
    Click(QWidget *parent = 0);
};
```

Файл реалізації - click.cpp:

```
#include <QPushButton>
#include <QApplication>
#include <QHBoxLayout>
#include "click.h"

Click::Click(QWidget *parent)
    : QWidget(parent) {

  QHBoxLayout *hbox = new QHBoxLayout(this);
  hbox->setSpacing(5);

  QPushButton *quitBtn = new QPushButton("Quit", this);
  hbox->addWidget(quitBtn, 0, Qt::AlignLeft | Qt::AlignTop);

  connect(quitBtn, &QPushButton::clicked, qApp, &QApplication::quit);
}
```

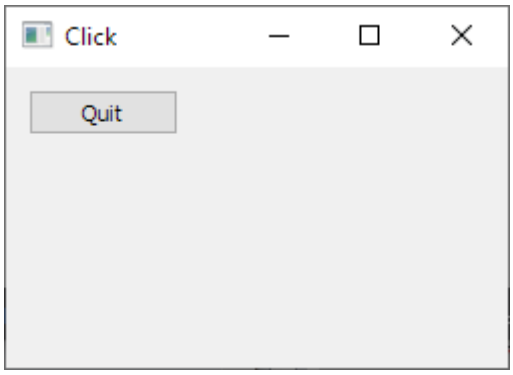
Метод connect() з'єднує сигнал зі слотом. Коли ми натискаємо на кнопку Quit, генерується сигнал клацання кнопки миші. qApp - це глобальний покажчик на об'єкт нашого застосування. Він визначається в заголовки QApplication. Метод quit() викликається при появі сигналу клацання мишкою:

```
connect(quitBtn, &QPushButton::clicked, qApp, &QApplication::quit);
```

```
connect(quitBtn, &QPushButton::clicked, qApp, &QApplication::quit);
```


Основний файл програми - main.cpp:

```
#include <QApplication>
#include "click.h"

int main(int argc, char *argv[]) {

  QApplication    app(argc,    argv);

  Click window;
  window.resize(250, 150);
  window.setWindowTitle("Click");
  window.show();

  return app.exec();
}
```



Натискання кнопки клавіатури

У наступному прикладі ми розглянемо спосіб реагування на натискання кнопки клавіатури. Додаток завершить своє виконання, якщо ми натиснемо на клавішу Esc.

Заголовки - keypress.h:

```
#pragma once
#include <QWidget>

class KeyPress : public QWidget {

  public:
    KeyPress(QWidget *parent = 0);

  protected:
    void keyPressEvent(QKeyEvent * e);
};
```

Файл реалізації - keypress.cpp:

```
#include <QApplication>
#include <QKeyEvent>
#include "keypress.h"

KeyPress::KeyPress(QWidget *parent)
    : QWidget(parent)
{ }

void KeyPress::keyPressEvent(QKeyEvent *event) {

   if (event->key() == Qt::Key_Escape) {
       qApp->quit();
   }
}
```

Одним із способів роботи з подіями в Qt5 є перевизначення обробника подій. QKeyEvent - це клас, який містить інформацію про подію, що відбулася. У нашому випадку ми використовуємо об'єкт даного класу, щоб визначити, яка саме клавіша була натиснута:

```
void KeyPress::keyPressEvent(QKeyEvent *e) {

   if (e->key() == Qt::Key_Escape) {
       qApp->quit();
   }
}
```

Основний файл програми - main.cpp:

```
#include <QApplication>
#include "keypress.h"

int main(int argc, char *argv[]) {

  QApplication app(argc, argv);

  KeyPress window;

  window.resize(250, 150);
  window.setWindowTitle("Key press");
  window.show();

  return app.exec();
}
```

Клас QMoveEvent

Клас QMoveEvent містить параметри подій, що виникають при переміщенні віджета. У наступному прикладі ми реагуємо на подію навігації, після чого визначаємо поточні координати хі уверхнього лівого кута клієнтської області вікна і встановлюємо ці значення в заголовок вікна.

Заголовки - move.h:

```
#pragma once
#include <QMainWindow>

class Move : public QWidget {

  Q_OBJECT

  public:
    Move(QWidget *parent = 0);

  protected:
    void moveEvent(QMoveEvent *e);
};
```

Файл реалізації - move.cpp:

```
#include <QMoveEvent>
#include "move.h"

Move::Move(QWidget *parent)
    : QWidget(parent)
```

```
{ }

void Move::moveEvent(QMoveEvent *e) {

  int x = e->pos().x();
  int y = e->pos().y();

  QString text = QString::number(x) + "," + QString::number(y);

  setWindowTitle(text);
}
```

Ми використовуємо об'єкт класу QMoveEvent для визначення значень xi y:

```
int x = e->pos().x();
int y = e->pos().y();
```

Потім ми конвертуємо цілочисельні значення в рядки :

```
QString text = QString::number(x) + "," + QString::number(y);
```

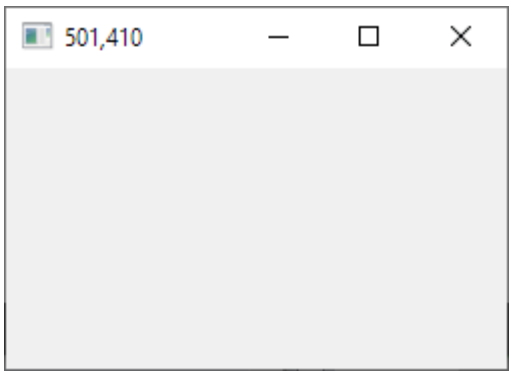
І за допомогою методу setWindowTitle () встановлюємо текст в заголовок вікна:

```
setWindowTitle(text);
```

Основний файл програми - main.cpp:

```
#include <QApplication>
#include "move.h"

int main(int argc, char *argv[]) {

  QApplication app(argc, argv);

  Move window;

  window.resize(250, 150);
  window.setWindowTitle("Move");
  window.show();

  return app.exec();
}
```

Результат:



Відключення сигналів

Сигнал може бути відключений від слота. Наступний приклад показує, як це можна зробити.

В заголовки ми оголосили два слота. Слід зазначити, що slotне є ключовим словом в мові C ++, а лише розширенням Qt5. Подібні розширення обробляються препроцесором фреймворка до виконання компіляції коду. Коли в наших класах ми використовуємо сигнали і слоти, то обов'язково повинні надати макрос Q_OBJECTна початку визначення класу. В іншому випадку препроцесор видаватиме повідомлення про помилки.

У наступному прикладі у нас є кнопка і прапорець. Прапорець підключає і відключає слот від сигналу натискання кнопок.

Заголовки - disconnect.h:

```
#pragma once

#include <QWidget>
#include <QPushButton>

class Disconnect : public QWidget {

  Q_OBJECT

  public:
    Disconnect(QWidget *parent = 0);

  private slots:
    void onClick();
    void onCheck(int);

  private:
    QPushButton *clickBtn;
};
```

Файл реалізації - disconnect.cpp:

```
#include <QTextStream>
#include <QCheckBox>
#include <QHBoxLayout>
#include "disconnect.h"

Disconnect::Disconnect(QWidget *parent)
    : QWidget(parent) {

  QHBoxLayout *hbox = new QHBoxLayout(this);
  hbox->setSpacing(5);
```

```
        clickBtn = new QPushButton("Click", this);
        hbox->addWidget(clickBtn, 0, Qt::AlignLeft | Qt::AlignTop);

        QCheckBox *cb = new QCheckBox("Connect", this);
        cb->setCheckState(Qt::Checked);
        hbox->addWidget(cb, 0, Qt::AlignLeft | Qt::AlignTop);
        connect(clickBtn,            &QPushButton::clicked,            this,
&Disconnect::onClick);
        connect(cb, &QCheckBox::stateChanged, this, &Disconnect::onCheck);
      }

      void Disconnect::onClick() {

        QTextStream out(stdout);
        out << "Button clicked" << endl;
      }

      void Disconnect::onCheck(int state) {

        if (state == Qt::Checked) {
          connect(clickBtn,           &QPushButton::clicked,           this,
&Disconnect::onClick);
        } else {
          disconnect(clickBtn, &QPushButton::clicked, this,
              &Disconnect::onClick);
        }
      }
```

Підключаємо сигнали до визначених слотів:

```
connect(clickBtn, &QPushButton::clicked, this, &Disconnect::onClick);
connect(cb, &QCheckBox::stateChanged, this, &Disconnect::onCheck);
```

Якщо ми робимо клацання мишкою, то в вікно терміналу відправлятиметься текст Button clicked:

```
      void Disconnect::onClick() {

        QTextStream out(stdout);
        out << "Button clicked" << endl;
      }
```

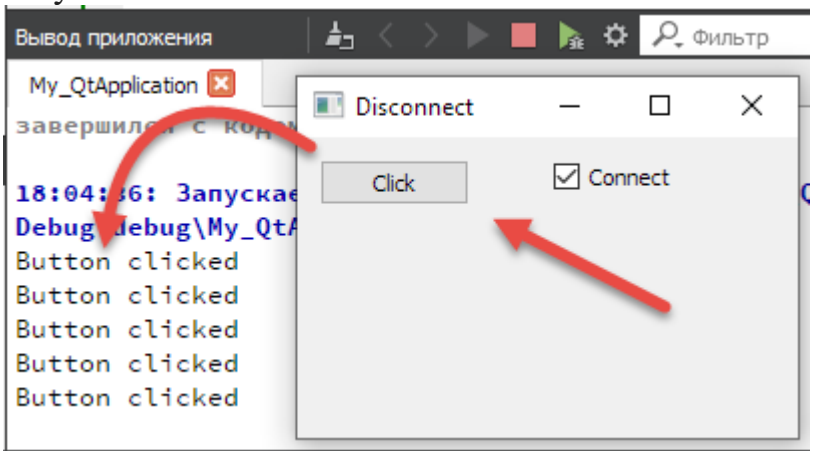
Усередині слота onCheck () ми підключаємо або відключаємо слот onClick () від кнопки, в залежності від отриманого параметра стану:

```
void Disconnect::onCheck(int state) {

  if (state == Qt::Checked) {
    connect(clickBtn,          &QPushButton::clicked,          this,
&Disconnect::onClick);
  } else {
    disconnect(clickBtn,          &QPushButton::clicked,          this,
&Disconnect::onClick);
  }
}
```

Основний файл програми - main.cpp:

```
#include <QApplication>
#include "disconnect.h"

int main(int argc, char *argv[]) {

  QApplication app(argc, argv);

  Disconnect window;

  window.resize(250, 150);
  window.setWindowTitle("Disconnect");
  window.show();

  return app.exec();
}
```

Результат:



Таймер

Таймер використовується для реалізації одиночного дії або ж повторюються. Хорошим прикладом, де ми можемо задіяти таймер, є годинник; щосекунди ми повинні оновлювати нашу мітку, яка буде показувати поточний час.

У наступному прикладі ми спробуємо відобразити у вікні поточний місцевий час.

Заголовки - timer.h:

```
#pragma once

#include <QWidget>
#include <QLabel>

class Timer : public QWidget {

  public:
    Timer(QWidget *parent = 0);

  protected:
    void timerEvent(QTimerEvent *e);

  private:
    QLabel *label;
};
```

Файл реалізації - timer.cpp:

```
#include "timer.h"
#include <QHBoxLayout>
#include <QTime>

Timer::Timer(QWidget *parent)
    : QWidget(parent) {

  QHBoxLayout *hbox = new QHBoxLayout(this);
  hbox->setSpacing(5);

  label = new QLabel("", this);
  hbox->addWidget(label, 0, Qt::AlignLeft | Qt::AlignTop);

  QTime qtime = QTime::currentTime();
  QString stime = qtime.toString();
  label->setText(stime);

  startTimer(1000);
}

void Timer::timerEvent(QTimerEvent *e) {
  Q_UNUSED(e);

  QTime qtime = QTime::currentTime();
```

```
  QString stime = qtime.toString();
  label->setText(stime);
}
```

Для відображення часу ми використовуємо віджет-мітку:

```
label = new QLabel("", this);
```

Потім ми визначаємо поточний місцевий час і встановлюємо його в віджет-мітку:

```
QTime qtime = QTime::currentTime();
QString stime = qtime.toString();
label->setText(stime);
```

Запускаємо таймер (при цьому кожні 1000 мс генерується подія таймера):

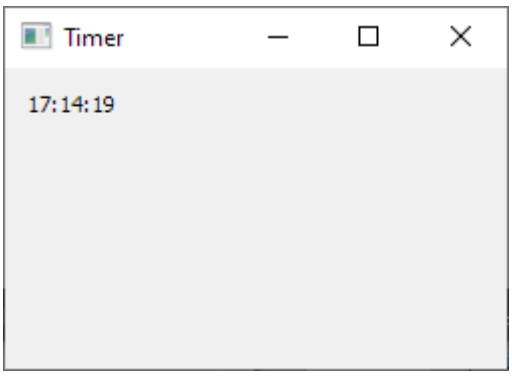
```
startTimer(1000);
```

Для роботи з подіями таймера необхідно перевизначити метод timerEvent ():

```
void Timer::timerEvent(QTimerEvent *e) {

  Q_UNUSED(e);

  QTime qtime = QTime::currentTime();
  QString stime = qtime.toString();
  label->setText(stime);
}
```

Основний файл програми - main.cpp:

```
#include <QApplication>
#include "timer.h"

int main(int argc, char *argv[]) {

  QApplication app(argc, argv);

  Timer window;

  window.resize(250, 150);
  window.setWindowTitle("Timer");
  window.show();

  return app.exec();
}
```



Клас QDialog є базовим класом діалогових вікон.

Діалогове вікно є вікном верхнього рівня, використовуваних головним чином для коротких завдань і лаконічного взаємодії з користувачем. QDialogs may be modal or modeless. QDialogs can provide a return value , and they can have default buttons . QDialogs can also have a QSizeGrip in their lower-right corner, using setSizeGripEnabled ().

Зауважте, що QDialog (будь-який інший віджет, що має тип Qt :: Dialog) використовує батьківський віджет дещо інакше, ніж інші класи в Qt. Діалогове вікно завжди є віджетом верхнього рівня, але якщо у нього є батько, його місцезнаходження за замовчуванням розташоване по центру зверху віджета верхнього рівня батьківського вікна (якщо це не сам рівень верхнього рівня). Він також матиме спільний запис на батьківській панелі завдань.

Використовуйте перевантаження функції QWidget :: setParent (), щоб змінити право власності на віджет QDialog. Ця функція дозволяє чітко встановити прапори вікон перенаправленого віджета; використання перевантаженої функції очистить прапори вікна із зазначенням властивостей віконної системи для віджета (зокрема, скине прапор Qt :: Dialog).

Модальні діалоги

Модальне діалогове вікно являє собою діалог , який блокує введення в інших видимих вікон в одному додатку. Діалоги, які використовуються для запиту імені файлу у користувача або використовуються для встановлення параметрів програми, як правило, модальні. Діалоги можуть бути модальними (за замовчуванням) або віконними .

Коли відкривається модальний діалог програми, користувач повинен закінчити взаємодію з діалогом і закрити його, перш ніж мати доступ до будь-якого іншого вікна програми. Модальні діалогові вікна лише блокують доступ до вікна, пов’язаного з діалоговим вікном, дозволяючи користувачеві продовжувати використовувати інші вікна у програмі.

Найпоширенішим способом відображення модального діалогового вікна є виклик його функції exec(). Коли користувач закриває діалогове вікно, exec () надасть корисне значення повернення . Як правило, щоб закрити діалогове вікно та повернути відповідне значення, ми підключаємо кнопку за замовчуванням, наприклад, OK , до слота accept () та кнопку Скасувати до слота reject (). Крім того, ви можете викликати слот done() за допомогою Accepted або Rejected.

Альтернативою є виклик setModal (true) або setWindowModality (), а потім show (). На відміну від exec (), show () негайно повертає управління абоненту. Виклик setModal (true) особливо корисний для діалогів прогресу, де користувач повинен мати можливість взаємодіяти з діалогом, наприклад, скасувати тривалу операцію. Якщо ви використовуєте show () і setModal (true) разом, щоб виконати тривалу операцію, ви повинні періодично викликати QApplication :: processEvents() під час обробки, щоб дозволити користувачеві взаємодіяти з діалоговим вікном. (Див. QProgressDialog .)

Немодальні діалоги

Немодальним діалог являє собою діалог, який працює незалежно від інших вікон в одному додатку. Знайти та замінити діалоги в текстових процесорах часто немодельні, щоб дозволити користувачеві взаємодіяти як з головним вікном програми, так і з діалоговим вікном.

Немодельні діалоги відображаються за допомогою show (), який негайно повертає контроль абоненту.

Якщо ви викликаєте функцію show () після приховування діалогового вікна, діалогове вікно відобразиться у вихідному положенні. Це пов'язано з тим, що менеджер вікон вирішує положення для вікон, які програміст не розмістив явно. Щоб зберегти позицію діалогового вікна, яке перемістив користувач, збережіть його місце у своєму обробнику closeEvent (), а потім перемістіть діалогове вікно у це положення, перш ніж показувати його знову.

Детальніше про QDialog: http://doc.crossplatform.ru/qt/4.6.x/qdialog.html

ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Створити новий графічний Qt-проєкт.
2. В класі MainWindow оголосити два приватні слоти: OpenModalDialog() та OpenNotModalDialog(), які відображатимуть відповідно модальне і не модальне вікно:

- за допомогою слота OpenModalDialog() відображати довільне повідомлення про критичну помилку (за допомогою класу QMessageBox());
- за допомогою слота OpenNotModalDialog() відображати не модальне вікно для отримання назви файлу, який потрібно відкрити. Для цього використати метод QFileDialog::getOpenFileName().

3. Розмістити на формі вікна класу MainWindow дві кнопки pbOpenModalDialog і pbOpenNotModalDialog. За допомогою методу connect() підключити сигнал clicked() даних кнопок до визначених в другому пункті слотів.
4. Добавити ще одну кнопку «Відключити», при кліку по якій відключити раніше підключені сигнали.
5. Переробити логіку роботи кнопки «Відключити», таким чином, щоб при повторному кліку по ній відбувалось підключення сигналів, якщо вони відключені, і відключення, якщо вони підключені. При цьому змінювати назву кнопки із «Відключити» на «Підключити» і наоборот.
6. Перевірити роботу програми.