**Часто** **використовувані сутності Qt**

[**QString**](http://doc.qt.io/qt-5/qstring.html)

Клас QString представляє рядок символів Unicode. Він зберігає рядок як 16-бітна колекція QChars. Кожен QChar відповідає одному символу Unicode 4.0.

std::string QString::toStdString() const

[QString](http://doc.qt.io/qt-5/qstring.html#QString) QString::fromStdString(const std::string& *str*)

Методи порівняння:

bool QString::startsWith(const [QStringRef](http://doc.qt.io/qt-5/qstringref.html) &*s*, [Qt::CaseSensitivity](http://doc.qt.io/qt-5/qt.html#CaseSensitivity-enum) *cs* = Qt::CaseSensitive) const

bool QString::endsWith(const [QStringRef](http://doc.qt.io/qt-5/qstringref.html) &*s*, [Qt::CaseSensitivity](http://doc.qt.io/qt-5/qt.html#CaseSensitivity-enum) *cs* = Qt::CaseSensitive) const

Рядки можна конвертувати в інші типи та навпаки.

[QString](http://doc.qt.io/qt-5/qstring.html#QString) QString::number(T *n*, int *base* = 10)

double QString::toDouble(bool \**ok* = Q\_NULLPTR) const

int QString::toInt(bool \**ok* = Q\_NULLPTR, int *base* = 10) const

[**QDateTime**](http://doc.qt.io/qt-5/qdatetime.html#QDateTime)

QDateTime — це клас, який працює з календарною датою за григоріанським календарем і часом годинника.

- [fromString](http://doc.qt.io/qt-5/qdatetime.html#fromString)(const QString &*string*, Qt::DateFormat *format*) – зчитує дату з рядка;

- [QString](http://doc.qt.io/qt-5/qstring.html) toString(const [QString](http://doc.qt.io/qt-5/qstring.html) &*format*) const – повертає datetime у вигляді рядка;

- qint64 QdateTime::toSecsSinceEpoch() const – повертає datetime як кількість секунд що пройшла з 1970-01-01T00:00:00.000, Coordinated Universal Time;

- [QDateTime](http://doc.qt.io/qt-5/qdatetime.html#QDateTime) QDateTime::toUTC() const – повертає datetime що містить вказані дату та час в специвікації Qt::UTC

**Контейнери** [**QVector**](http://doc.qt.io/qt-5/qvector.html#QVector)**,** [**QList**](http://doc.qt.io/qt-5/qlist.html#QList)**,** [**QMap**](http://doc.qt.io/qt-5/qmap.html#QMap)

Контейнери — це класи загального призначення, які зберігають елементи певного типу в пам’яті. C++ має стандартну бібліотеку шаблонів (STL), яка має власні контейнери. У Qt ми можемо використовувати контейнери Qt або STL.

QVector — це шаблонний клас, який представляє динамічний масив.

QList — це контейнер для створення списку елементів.

QStringList — це зручний контейнер, який представляє список рядків. Він має швидкий доступ на основі індексу, а також швидке вставлення та видалення.

QSet представляє single-valued математичний набір із швидким пошуком. Значення зберігаються в невизначеному порядку.

QMap — це асоціативний масив (словник), який зберігає пари ключ-значення. Він забезпечує швидкий пошук значення, пов’язаного з ключем.

**GUI**

**Widgets**

Віджети є основними будівельними блоками програми GUI. Кожен елемент керування (кнопка, смуга прокрутки, діалогове вікно) називається віджетом, і кожен із класів елементів керування (QPushButton, QScrollBar, QDialog) успадковує клас QWidget. QWidget успадковує QObject і підтримує зв’язок «батько-нащадок»: QWidget приймає вказівник на інший QWidget у своєму конструкторі:

QWidget::QWidget([QWidget](http://doc.qt.io/qt-4.8/qwidget.html#QWidget) \**parent* = 0, Qt::WindowFlags*f* = 0)

Батьківський віджет, який буде передано дочірньому конструктору, видалить дочірній віджет у своєму деструкторі.

Такий підхід до побудови графічного інтерфейсу дозволяє створити ієрархію віджетів, у якій видалення, скажімо, діалогового вікна призведе до видалення всіх його дочірніх елементів. Тому в коді Qt операції з покажчиками на великі об'єкти не є небезпечними.

Існує багато вбудованих типів елементів керування: QLabel, QPushButton, QCheckBox, QLineEdit, QTextEdit, QSpinBox тощо. Кожен з них може служити базовим класом для власного елемента керування, поведінку якого можна налаштувати.

**Model/View**

Віджети таблиці, списку та дерева є компонентами, які часто використовуються в GUI. Є 2 різні способи, як ці віджети можуть отримати доступ до своїх даних. Традиційний спосіб включає віджети, які містять внутрішні контейнери для зберігання даних. Цей підхід дуже інтуїтивно зрозумілий, однак у багатьох нетривіальних програмах він призводить до проблем із синхронізацією даних. Другий підхід — це програмування моделі/подання, у якому віджети не підтримують внутрішні контейнери даних. Вони отримують доступ до зовнішніх даних через стандартизований інтерфейс і тому уникають дублювання даних.

Компонент, який надає ці служби представлення та редагування, називається делегатом. Делегат також потрібен для візуалізації користувацьких елементів і обробки подій, які відбуваються з елементами (наприклад, клацання певної області або виділення тексту, що відображається на елементі). Елементом View вважається 1 комірка таблиці. Модель списку використовуватиме таблицю з кількістю стовпців, рівним 1, а в моделі дерева елемент таблиці має вказівник на батьківський елемент. У будь-якому з цих випадків модель має справу з тим самим типом абстрактного елемента, який ідентифікується стовпцем, рядком і батьківським індексом.

Таким чином, одну модель даних можна використовувати в різних представленнях. Наприклад: модель містить дані деревовидної структури. Усі дані моделі можна відобразити в одному поданні типу QTreeView. Кожен із батьківських елементів можна відобразити в окремому QTableView. І, скажімо, властивості кожного елемента таблиці можна відобразити в QListView. Таким чином, маючи одну модель, ми можемо відображати різні частини її даних у різних Views, і в той же час, крім того, зміни моделі відображаються відразу в усіх Views.

**Multithreading**

[**QThread**](http://doc.qt.io/qt-4.8/qthread.html#QThread)

Об'єкт QThread керує одним потоком керування в програмі. QThreads починають виконуватися в run(). За замовчуванням run() запускає цикл подій шляхом виклику exec() і запускає цикл подій Qt всередині потоку.

Ви можете використовувати робочі об’єкти, перемістивши їх у потік за допомогою Qobject::moveToThread().

Тоді код у слоті Worker виконується в окремому потоці. Однак ви можете підключити слоти Worker до будь-якого сигналу, від будь-якого об’єкта, у будь-якому потоці.

Інший спосіб змусити код запускатися в окремому потоці – створити підклас QThread і повторно реалізувати run().

[**QtConcurrent**](http://doc.qt.io/qt-5/qtconcurrent-module.html)

Простір імен QtConcurrent надає API високого рівня, які дозволяють писати багатопотокові програми без використання примітивів низькорівневих потоків, таких як м’ютекси, блокування читання-запису, умови очікування або семафори. Програми, написані за допомогою QtConcurrent, автоматично регулюють кількість використовуваних потоків відповідно до кількості доступних ядер процесора. Це означає, що додатки, написані сьогодні, продовжуватимуть масштабуватися при розгортанні на багатоядерних системах у майбутньому.

QtConcurrent::map() застосовує функцію до кожного елемента в контейнері, змінюючи елементи на місці.

QtConcurrent::run() виконує функцію в іншому потоці.

QFuture представляє результат асинхронного обчислення.

**Events**

Events є важливою частиною будь-якої програми з графічним інтерфейсом користувача. Усі додатки GUI керуються подіями. Програма реагує на різні типи подій, які генеруються протягом її життя. Події генеруються в основному користувачем програми. Але вони можуть бути згенеровані й іншими засобами, напр. підключення до Інтернету, менеджер вікон або таймер.

Події в Qt є об’єктами спеціалізованих класів, успадкованих від QEvent. Обробники подій — це функції в класах, похідних від класу QObject, які приймають певні типи подій і призначені для виконання певних дій щодо свого об’єкта залежно від інформації, переданої в об’єкті події. Якщо подія не може бути оброблена об’єктом, який її отримав, подія передається об’єкту-власнику, доки він не досягне рівня QApplication.

Об’єкти QObject отримують події за допомогою виклику функції QObject::event(). Функція може бути повторно реалізована в підкласах, щоб налаштувати обробку подій і додати додаткові типи подій; QWidget::event() є яскравим прикладом. За замовчуванням події надсилаються до обробників подій, таких як QObject::timerEvent() і QWidget::mouseMoveEvent(). QObject::installEventFilter() дозволяє об’єкту перехоплювати події, призначені для іншого об’єкта.

**Signals & Slots**

Сигнали та слоти використовуються для зв'язку між об'єктами. Механізм сигналів і слотів є центральною особливістю Qt і, ймовірно, частиною, яка найбільше відрізняється від функцій, наданих іншими фреймворками.

Сигнал ініціюється, коли відбувається певна подія. Віджети Qt мають багато попередньо визначених сигналів, але ми завжди можемо створити підкласи віджетів, щоб додати до них власні сигнали. Слот — це функція, яка викликається у відповідь на певний сигнал. Віджети Qt мають багато попередньо визначених слотів, але звичайною практикою є створення підкласів віджетів і додавання власних слотів, щоб ви могли обробляти сигнали, які вас цікавлять.

emit SignalExample();

QObject::connect(&emitter, &Emitter::SignalExample, &receiver, &Receiver::SlotExample);

**Literature:**

Official docs: https://doc.qt.io/qt.html

Events: <http://wiki.qt.io/Threads_Events_QObjects>

Model/view: <http://doc.qt.io/qt-5/modelview.html>

**Books**:

Guillaume Lazar, «Mastering Qt 5: Create stunning cross-platform applications using C++ with Qt Widgets and QML with Qt Quick»

Jasmine Blanche, Mark Summerfield, «C ++ GUI Programming with Qt 4»