QT Object Model

- Signals and Slots

- A signal is emitted when a particular event occurs, Qt's widgets have many predefined signals, but we can always subclass widgets to add our own signals to them.
- A slot is a function that is called in response to a particular signal. QT's widgets have many predefined slots, but it is common practise to subclass widgets and add your own slots so that you can handle the signals that you are interested in.
- All classes that inherit from QObject or one of its subclasses can contain signals and slots.
- Signals are emitted by an object when its internal state has changed in some way that might be interesting to the object's client or owner. Signals are public classes functions and can be emitted from anywhere
- When a signal is emitted, the slots connected to it are usually executed immediately, just like normal function call.
- A slot is called when a signal connected to it is emitted. Slots are normal C++ functions and can be called normally; their only special ~~function~~ feature is that signals can be connected to them.

- All classes that contain signals and slots must mention Q_OBJECT at the top of their declaration. They must also derive (directly or indirectly) from QObject

---

EXAMPLE

```
#include <QObject>
class Counter : public QObject
{
    Q_OBJECT
public:
    Counter() { m_value = 0; }

    int value() const {
        return m_value;
    }
public slots:
    void setValue(int newValue);
signals:
    void valueChanged(int newValue);
private:
    int m_value;
};

void Counter::setValue(int newValue)
{
    if (m_value != newValue)
    {
        m_value = newValue;
        emit valueChanged(newValue);
    }
}
```

```
Counter a, b;
QObject::connect(
    &a, &Counter::valueChanged,
    &b, &Counter::setValue);

a.setValue(12);   // a.value = 12
                  // b.value = 12

b.setValue(48);   // a.value = 12
                  // b.value = 48
```

QT Property System

QT provides a sophisticated property system similar to the ones supplied by some compiler vendors. However, as a compiler- and platform-independent library, Qt does not rely on non standard compiler features like __property or [property]. It works with every standard C++ compiler on every platform Qt supports.

To declare property use Q_PROPERTY() macro in a class that inherits QObject.

QT Event System

Events are objects, derived from the abstract QEvent class, that represent things that have happened either within an application or as a result of outside activity that the app needs to know about. They can be handled and received by any instance of QObject's subclass, but they are especially relevant to widgets.

When an event occurs, Qt creates an event object to represent it by constructing an instance of the appropriate QEvent subclass and delivers it to a particular instance of QObject by calling its event() function.

This function does not handle the event itself; based on the type of event delivered, it calls an event handler for that specific type of event and sends response based on whether the event was accepted or ignored.

QT Meta-Object Compiler (moc)

- MOC - it is a program that handles Qt's C++ extensions

The moc tool reads a C++ header file. If it finds one or more class declarations that contain the Q_OBJECT macro, it produces a C++ source file containing the meta-object code for those classes. Among other things, meta-object code is required for the signals and slots mechanism, the run-time type information and the dynamic property system.

QT Meta-Object System

It provides the signals and slots mechanism for inter-object communication, run-time type information and the dynamic property system. It is based on three things:

1. The QObject class provides a base class for objects that can take advantage of the meta-object system.
2. The Q_OBJECT macro inside the private section of the class declaration is used to enable meta-object features such as dynamic properties, signals and slots.
3. The Meta-object Compiler supplies each QObject subclass with the necessary code to implement meta-object features.

Internationalization with QT

The internationalization and localization of an application are the processes of adapting the application to different languages, regional differences and technical requirements of a target market. It means designing a software application so that it can be adapted to various languages and regions without engineering changes.

- QCollator
- QCollatorSortKey
- QLocale
- QTextCodec
- QTextDecoder
- QTextEncoder
- ~~QTextI~~
- QTranslator

Timers

QObject provides the basic timer support in Qt.
QObject::startTimer() → QObject::killTimer()
Application must run in an event loop. You start an event loop with QApplication::exec(). In multithreaded apps, you can use the timer mechanism in any thread that has an event loop. You can start an event loop on non-GUI thread by QThread::exec().
The main API for the timer functionality is QTimer. That class provides regular timers that emit a signal when the timer fires and inherits QObject so that it fits well into the ownership structure of most GUI programs.

Qt Pointers

QPointer - can only point to QObject instances. It will be automatically set to nullptr if the pointed object is destroyed. It is a weak pointer specialized for QPointer.

QSharedPointer - a reference-counted object. The actual object will only be deleted, when all shared pointers are destroyed. Equivalent to std::shared_ptr.

QWeakPointer - can hold a weak reference to shared pointer. It will not prevent the object from being destroyed and is simply reset. Equivalent to std::weak_ptr where lock is equivalent to toStrongRef.

Qt dynamic cast

The qobject_cast() functions behaves similarly to the standard C++ dynamic_cast(), with the advantages that it doesn't require RTTI support and it works across dynamic library boundaries

It attemps to cast its argument to the pointer type specified in angle-brackets, returning a non-zero pointer if the object is of the correct type, or 0 if the object's type is incompatible

Qt Important Classes

QMetaClassInfo - provides additional information about a class

QMetaEnum - provides metadata about an enumerator

QMetaMethod - provides meta-data about a member function

QMetaObject - contains meta-information about Qt objects

QMetaProperty - provides meta-data about a property

QMetaType - manages named types in the meta-object system

QObject - base class of all Qt objects

QObjectCleanupHandler - watches the lifetime of multiple QObjects

QPointer - template class that provides guarded pointers to QObject

QSignalBlocker - exception-safe wrapper around QObject::blockSignals()

QVariant - acts like a union for the most common Qt data types.

The State Machine Framework

It provides classes for creating and executing state graphs.

Statecharts provide a graphical way of modeling how a system reacts to stimuli. This is done by defining the possible states that the system can be in, and how the system can move from one state to another (transitions between states)

This framework provides an API and execution model that can be used to effectively embed the elements and semantics of statecharts in Qt apps.

Classes:

QAbstractState - base class of states of a QStateMachine

QAbstractTransition - base class of transitions between QAbstractState objects

QEventTransition - QObject-specific transition for Qt events

QFinalState

QHistoryState - means of returning to a previously active substate

QKeyEventTransition - transition for key events
QMouseEventTransition - -||- mouse events

QSignalTransition - Transition based on a Qt signal

QState - General purpose state for QStateMachine

QStateMachine - hierarchical finite state machine

Qt Container classes

- QList
- QLinkedList
- QVector
- QStack
- QQueue
- QSet
- QMap
- QMultiMap
- QHash
- QMultiHash

Qt Iterator Classes

Container	Read-only	Read-write
QList, QQueue	QListIterator	QMutableListIterator
QLinkedList	QLinkedListIterator	EVERY WITH "MUTABLE" KEY-WORD
QVector, QStack	QVectorIterator	
QSet	QSetIterator	
QMap, QMultiMap	QMapIterator	
QHash, QMultiHash	QHashIterator	QMutableHashIterator

STL-style Iterators

Container	Read-only	Read-write
Same as above	QList<T>::const_iterator	QList<T>::iterator
	⋮	⋮
	Same style but another name	Same style but another name

Other Container-like objects

* Cannot be used with foreach

- QVarLengthArray<T, Prealloc> - low-level variable-length array. It can be used instead of QVector in places where speed is particularly important
- QCache<Key, T> - cache to store objects of a certain type T associated with keys of type Key
- QContiguousCache<T> - efficient way of caching data that is typically accessed in a contiguous way
- QPair<T1, T2> - stores pair of elements.
- QBitArray - provides an array of bits
- QByteArray - -||- bytes
- QString - provides a Unicode character string
- QStringList - provides a list of strings

* For unicode you have to check Qt documentation*

Inter-Process Communication in Qt

- TCP / IP

  The cross-platform QtNetwork module provides classes that make network programming portable and easy. It offers high-level classes that communicate using specific application-level protocols and lower-level class for implementing protocols.

- SharedMemory

  The cross-platform shared memory class provide access to the operating system's shared memory implementation.

- D-Bus protocols

  This module is a Unix-only library you can use to implement IPC using D-Bus protocol. It extends Qt's signals and slots mechanism to the IPC level, allowing a signal emitted by one process to be connected to a slot in another process.

- QProcess class

  Can be used external programs as child processes and to communicate with them. It provides an API for monitoring and controlling the state of the child

- Session Management

  Only for Linux/X11 platforms