Qt: Signals & Slots

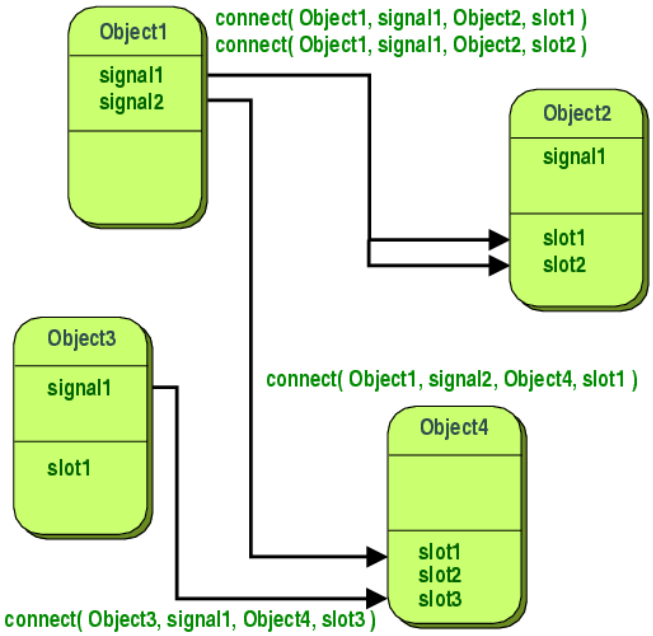
(Source: http://qt-project.org/doc/qt-4.8/signalsandslots.html)

Introduction

Signals and slots are used for communication between objects. The signals and slots mechanism is a central feature of Qt and probably the part that differs most from the features provided by other frameworks.

A callback is a pointer to a function, so if you want a processing function to notify you about some event you pass a pointer to another function (the callback) to the processing function. The processing function then calls the callback when appropriate.

In Qt, there is an alternative to the callback technique: signals and slots. A signal is emitted when a particular event occurs. Qt's widgets have many predefined signals, but a client programmer may always subclass widgets to add other signals to them. A slot is a function that is called in response to a particular signal. Qt's widgets have many pre-defined slots, but it is common practice to subclass widgets and add other slots so that a client may handle the signals that you are interested in.



The signals and slots mechanism is type safe: The signature of a signal must match the signature of the receiving slot. (In fact a slot may have a shorter signature than the signal it receives because it can ignore extra arguments.) Since the signatures are compatible, the compiler can help us detect type mismatches. Signals and slots are loosely coupled: A class which emits a signal neither knows nor cares which slots receive the signal. Qt's signals and slots mechanism ensures that if you connect a signal to a slot, the slot will be called with the signal's parameters at the right time. Signals and slots can take any number of arguments of any type. They are completely type safe.

All classes that inherit from `QObject` or one of its subclasses (e.g., `QWidget`) can contain signals and slots. Signals are emitted by objects when they change their state in a way that may be interesting to other objects. This is all the object does to communicate. It does not know or care whether anything is receiving the signals it emits. This is true information encapsulation, and ensures that the object can be used as a software component.

Slots can be used for receiving signals, but they are also normal member functions. Just as an object does not know if anything receives its signals, a slot does not know if it has any signals connected to it. This ensures that truly independent components can be created with Qt.

A client programmer can connect as many signals as necessary to a single slot, and a signal can be connected to as many slots as needed. It is even possible to connect a signal directly to another signal. (This will emit the second signal immediately whenever the first is emitted.)

Together, signals and slots make up a powerful component programming mechanism.

Simple example (http://www.tecgraf.puc-rio.br/ftp_pub/lfm/civ2802-qt-signalslot-example.zip)

File "counter.h"

```
#include <QObject>

class Counter : public QObject
{
    Q_OBJECT
public:
    Counter() { m_value = 0; }
    int value() const { return m_value; }
public slots:
    void setValue(int value);
signals:
    void valueChanged(int newValue);
private:
    int m_value;
};
```

File "counter.cpp"

```
#include "counter.h"

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

File "main.cpp"

```
#include "counter.h"

int main( void )
{
  Counter a, b;
  QObject::connect(&a, SIGNAL(valueChanged(int)),
                   &b, SLOT(setValue(int)));

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

  return 0;
}
```

A `QObject`-based class provides public methods to access its objects internal state, but in addition it has support for component programming using signals and slots. This class can tell the outside world that its state has changed by emitting a signal, `valueChanged()`, and it has a slot which other objects can send signals to.

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

Slots are implemented by the application programmer. In the file `counter.cpp` there is a possible implementation of the `Counter::setValue()`.

The `emit` line emits the signal `valueChanged()` from the object, with the new value as argument.

In the code in file `main.cpp`, two `Counter` objects are created and the first object's `valueChanged()` signal is connected to the second object's `setValue()` slot using `QObject::connect()`.

Calling `a.setValue(12)` makes object `a` emit a `valueChanged(12)` signal, which object `b` will receive in its `setValue()` slot, i.e. `b.setValue(12)` is called. Then `b` emits the same `valueChanged()` signal, but since no slot has been connected to `b`'s `valueChanged()` signal, the signal is ignored.

Note that the `setValue()` function sets the value and emits the signal only `if value != m_value`. This prevents infinite looping in the case of cyclic connections (e.g., if `b.valueChanged()` were connected to `a.setValue()`).

This example illustrates that objects can work together without needing to know any information about each other. To enable this, the objects only need to be connected together, and this can be achieved with some simple `QObject::connect()` function calls.