Event Filters in Qt

This article demonstrate how to use Qt Event Filters.

Basic Concept

QObject-based classes have an additional event handlers with which they react to *their own* events.

Event Filters allow an object **A** to receive the events of another object **B**. For each event of **B** that **A** receives, **A** can then either forward it to **B** or remove it from **B**s event stream. Before using the filter events the Event Filter must be installed. To do this we call `installEventFilter()` in the constructor of the object **A** that is to monitor the events of object **B**.

```
b->installEventFilter(this);
```

Here **b** is a pointer to **B**. Now **B** gives up all its events to **A** and leaves **A** with the decision whether it should filter out the event or let it through to **B**.

For this purpose an `eventFilter()` method is used, which has the following signature:

```
bool QObject::eventFilter(QObject *watched, QEvent *e);
```

This must be reimplemented by **A**. The *watched* parameter allows events from several monitored objects to be distinguished from one another, and *e* is the event to be processed.

The return value tells the event system how it should proceed with the event. If *false* is returned, it is forwarded to the monitored object, whereas *true* causes it to be filtered out. This means that the event does not arrive at the object for which it was originally intended.

Implementation

For this example we will create a chat dialog that sends the entered message when the **Enter** key is pressed. This text is entered into a QTextEdit - we use an event filters to change the default behaviour when **Enter** key is pressed, which was to start a line.

Class Declaration

```
#ifndef CHATWINDOW_H
#define CHATWINDOW_H
#include <QMainWindow>
#include <QWidget>
#include "QVBoxLayout"
#include "QSplitter"
#include "QTextBrowser"
#include "QTextEdit"
#include "QKeyEvent"
class QTextBrowser;
class QTextEdit;
class QEvent;
namespace Ui {
    class ChatWindow;
}
class ChatWindow : public QWidget
{
    Q_OBJECT
public:
    explicit ChatWindow(QWidget *parent = 0);
    ~ChatWindow();
private:
    Ui::ChatWindow *ui;
public:
    bool eventFilter(QObject *watched, QEvent *e);
    void submitChatText();
    private:
    QTextBrowser *conversationView;
    QTextEdit *chatEdit;
};
#endif // CHATWINDOW_H
```

The `submitChatText()` method is responsible for sending the text. In this example its only task consists of including the written text from the QTextEdit instance *chatEdit* into the *conversationView*. Pointers to each of these widgets are saved in member variables.

Class Implementation

Here we first define the constructor, We place a vertical splitter into the widget with a QVBoxLayout. The **conversation view** comes into the splitter at the top, followed by the actual input widget, **chatEdit**

```
ChatWindow::ChatWindow(QWidget *parent) :
   QWidget(parent)
{
    QVBoxLayout *lay = new QVBoxLayout(this);
    QSplitter *splitter = new QSplitter(Qt::Vertical, this);
    lay->addWidget(splitter);
    conversationView = new QTextBrowser;
    chatEdit = new QTextEdit;
    splitter->addWidget(conversationView);
    splitter->addWidget(chatEdit);
    chatEdit->installEventFilter(this);
    setWindowTitle(tr("Chat Window"));
    setTabOrder(chatEdit, conversationView);
}
```

Then we install the event filter in the input widget using `installEventFilter()`. The target is the `ChatWindow` object itself. The `ChatWindow` will filter the key press events of the `chatEdit` object and respond to them, so that we do not need to implement a specialized subclass of QTextEdit for this application.

Finally, we set the window title and use `setTabOrder()` to specify the order in which the widgets will be given focus. The call in this case has the effect that `chatEdit` obtains the focus before `conversationView`, so that the user can begin typing immediately after the program starts. At the same time `chatEdit` obtains the focus as soon as the `show()` method of a **ChatWindow** instance is called.

Now lets turn to the core item of the example, the `eventFilter()` method:

```
bool ChatWindow::eventFilter(QObject *watched, QEvent* e)
{
if (watched == chatEdit && e->type() == QEvent::KeyPress) {
QKeyEvent *ke = static_cast<QKeyEvent*>(e);
if (ke->key() == Qt::Key_Enter ||
ke->key() == Qt::Key_Return) {
submitChatText();
return true;
}
}
return QWidget::eventFilter(watched, e);
}
```

We first use a pointer comparison to check whether the filter is currently handling `chatEdit` at all and whether the pressing of a key (`QEvent::KeyPress`) is involved.

Once we are sure of this, we cast the generic event e into its actual event type, `QKeyEvent`, with a `static_cast`. This is necessary to access the *keypress* event's `key()` method, which we now use to check whether the key pressed is either the **Enter** key. If this is the case, we call `submitChatText()` and request Qt to filter the event with return true, that is, not to forward it to the `chatWindow` object. If the event is not a *keypress* event, we forward it to the parent class's event filter.

The `submitChatText()` method, which would also be responsible for forwarding text in a real chat client, in our example only attaches the typed text to the conversation view and empties the text window.

```
void ChatWindow::submitChatText()
{
// append text as new paragraph
conversationView->append(chatEdit->toPlainText());
// clear chat window
chatEdit->setPlainText("");
}
```

We also check this class again for its functionality with a short test program, by starting an event loop via `QApplication::exec()`, after we have instantiated and displayed `ChatWindow`.

```
#include <QtGui/QApplication>
#include "chatwindow.h"
#include <QtGui>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    ChatWindow w;
    w.showMaximized();
    return a.exec();
}
```

Source Code

The full source code presented in this article is available here File:ChatWindow.zip