The Event System | Qt 4.8

In Qt, 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 application needs to know about. Events can be received and handled by any instance of a QObject subclass, but they are especially relevant to widgets. This document describes how events are delivered and handled in a typical application.

How Events are Delivered

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 (or one of its subclasses) 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 a response based on whether the event was accepted or ignored.

Some events, such as QMouseEvent and QKeyEvent, come from the window system; some, such as QTimerEvent, come from other sources; some come from the application itself.

Event Types

Most events types have special classes, notably QResizeEvent, QPaintEvent, QMouseEvent, QKeyEvent, and QCloseEvent. Each class subclasses QEvent and adds event-specific functions. For example, QResizeEvent adds size() and oldSize() to enable widgets to discover how their dimensions have been changed.

Some classes support more than one actual event type. QMouseEvent supports mouse button presses, double-clicks, moves, and other related operations.

Each event has an associated type, defined in QEvent::Type, and this can be used as a convenient source of run-time type information to quickly determine which subclass a given event object was constructed from.

Since programs need to react in varied and complex ways, Qt's event delivery mechanisms are flexible. The documentation for QCoreApplication::notify() concisely tells the whole story; the *Qt Quarterly* article Another Look at Events rehashes it less concisely. Here we will explain enough for 95% of applications.

Event Handlers

The normal way for an event to be delivered is by calling a virtual function. For example, QPaintEvent is delivered by calling QWidget::paintEvent(). This virtual function is responsible for reacting appropriately, normally by repainting the widget. If you do not perform all the necessary work in your implementation of the virtual function, you may need to call the base class's implementation.

For example, the following code handles left mouse button clicks on a custom checkbox widget while passing all other button clicks to the base QCheckBox class:

```
void MyCheckBox::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        // handle left mouse button here
    } else {
        // pass on other buttons to base class
        QCheckBox::mousePressEvent(event);
    }
}
```

If you want to replace the base class's function, you must implement everything yourself. However, if you only want to extend the base class's functionality, then you implement what you want and call the base class to obtain the default behavior for any cases you do not want to handle.

Occasionally, there isn't such an event-specific function, or the event-specific function isn't sufficient. The most common example involves **Tab** key presses. Normally, QWidget intercepts these to move the keyboard focus, but a few widgets need the **Tab** key for themselves.

These objects can reimplement QObject::event(), the general event handler, and either do their event handling before or after the usual handling, or they can replace the function completely. A very unusual widget that both interprets **Tab** and has an application-specific custom event might contain the following event() function:

```
bool MyWidget::event(QEvent *event)
{
    if (event->type() == QEvent::KeyPress) {
        QKeyEvent *ke = static_cast<QKeyEvent *>(event);
        if (ke->key() == Qt::Key_Tab) {
            // special tab handling here
            return true;
        }
    } else if (event->type() == MyCustomEventType) {
        MyCustomEvent *myEvent = static_cast<MyCustomEvent *>(event);
        // custom event handling here
        return true;
    }
```

```
    return QWidget::event(event);
}
```

Note that QWidget::event() is still called for all of the cases not handled, and that the return value indicates whether an event was dealt with; a `true` value prevents the event from being sent on to other objects.

Event Filters

Sometimes an object needs to look at, and possibly intercept, the events that are delivered to another object. For example, dialogs commonly want to filter key presses for some widgets; for example, to modify **Return**-key handling.

The QObject::installEventFilter() function enables this by setting up an *event filter*, causing a nominated filter object to receive the events for a target object in its QObject::eventFilter() function. An event filter gets to process events before the target object does, allowing it to inspect and discard the events as required. An existing event filter can be removed using the QObject::removeEventFilter() function.

When the filter object's eventFilter() implementation is called, it can accept or reject the event, and allow or deny further processing of the event. If all the event filters allow further processing of an event (by each returning `false`), the event is sent to the target object itself. If one of them stops processing (by returning `true`), the target and any later event filters do not get to see the event at all.

```
bool FilterObject::eventFilter(QObject *object, QEvent *event)
{
    if (object == target && event->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
        if (keyEvent->key() == Qt::Key_Tab) {
            // Special tab handling
            return true;
        } else
            return false;
    }
    return false;
}
```

The above code shows another way to intercept **Tab** key press events sent to a particular target widget. In this case, the filter handles the relevant events and returns `true` to stop them from being processed any further. All other events are ignored, and the filter returns `false` to allow them to be sent on to the target widget, via any other event filters that are installed on it.

It is also possible to filter *all* events for the entire application, by installing an event filter on the

QApplication or QCoreApplication object. Such global event filters are called before the object-specific filters. This is very powerful, but it also slows down event delivery of every single event in the entire application; the other techniques discussed should generally be used instead.

Sending Events

Many applications want to create and send their own events. You can send events in exactly the same ways as Qt's own event loop by constructing suitable event objects and sending them with QCoreApplication::sendEvent() and QCoreApplication::postEvent().

sendEvent() processes the event immediately. When it returns, the event filters and/or the object itself have already processed the event. For many event classes there is a function called isAccepted() that tells you whether the event was accepted or rejected by the last handler that was called.

postEvent() posts the event on a queue for later dispatch. The next time Qt's main event loop runs, it dispatches all posted events, with some optimization. For example, if there are several resize events, they are compressed into one. The same applies to paint events: QWidget::update() calls postEvent(), which eliminates flickering and increases speed by avoiding multiple repaints.

postEvent() is also used during object initialization, since the posted event will typically be dispatched very soon after the initialization of the object is complete. When implementing a widget, it is important to realise that events can be delivered very early in its lifetime so, in its constructor, be sure to initialize member variables early on, before there's any chance that it might receive an event.

To create events of a custom type, you need to define an event number, which must be greater than QEvent::User, and you may need to subclass QEvent in order to pass specific information about your custom event. See the QEvent documentation for further details.