События. Фильтры событий

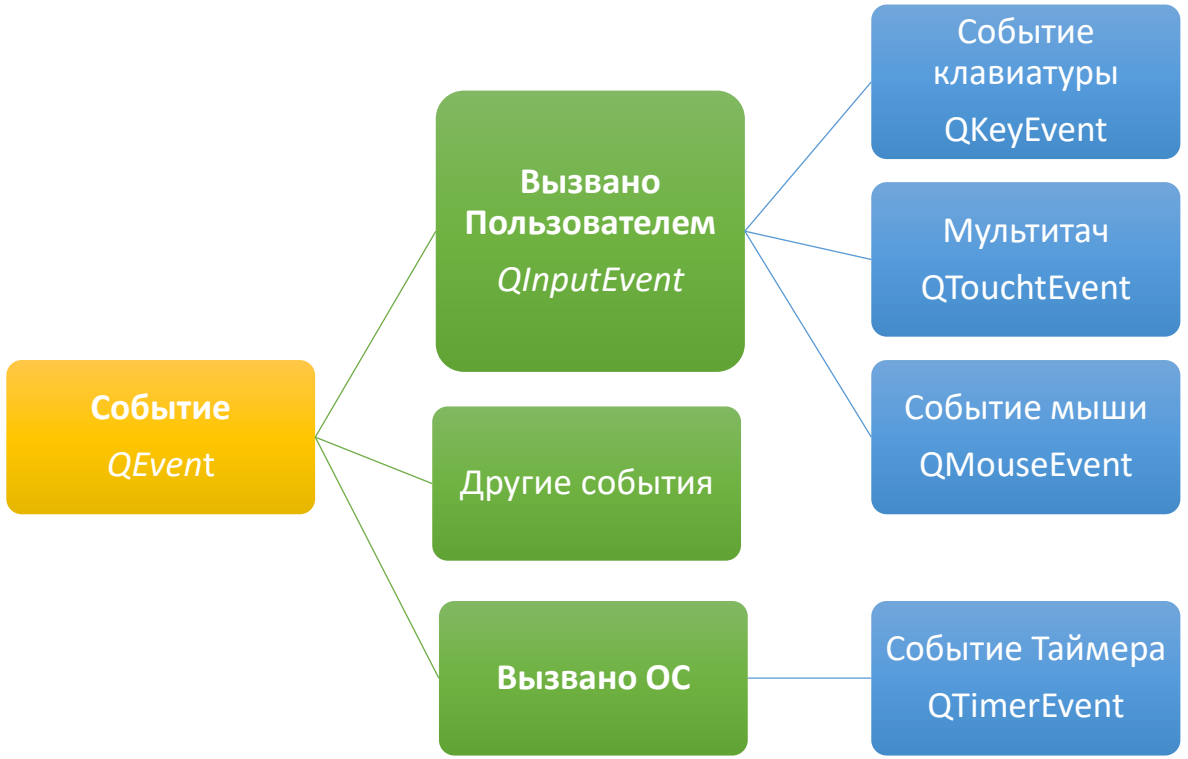
Лекция + практика

Обработка событий

Событие есть механизм оповещения приложения о каком-либо происшествии. Нажатие кнопки мыши или клавиатуры приведет к созданию события мыши или клавиатуры.

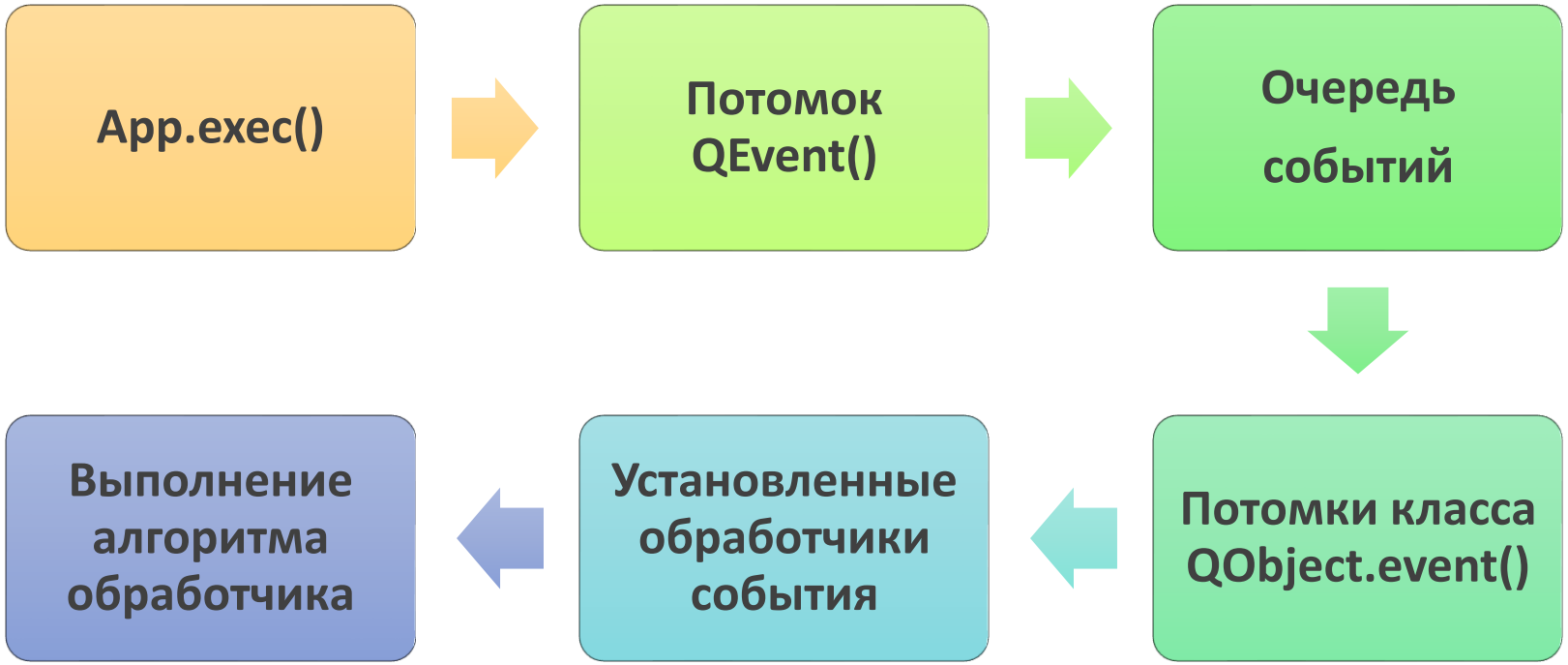
Событие и механизм сигналов-слотов выполняют в сущности одно и то же: оповещают ПО о к- л происшествиях, но с разных уровней. Сигналы предназначены для связи объектов внутри ПО. События системой и программными объектами. Часто **сигналы** высылаются из **методов обработки событий.**
Щелчок мыши на виджете приведет к обработке события *mousePressEvent()* и высылке сигнала *clicked()*.

Все созданные события помещаются в соответствующую очередь для обработки.



Событие обрабатывается только одним методом а сигналы могут обрабатываться неограниченным кол-вом соединённых с ним слотов.

Обработка событий



Обработка событий

Обработка событий начинается с момента вызова в по метода **QCoreApp::exec().** События доставляются всем наследникам *QObject,* сразу или через очередь в которой хранятся до обработки при возвращении управления циклу обработки событий ***exec()***.

Переопределение спец. Методов является распространенным способом обработки событий. т.е. Чтобы обработать определенное событие, нужно унаследовать определенный класс и переопределить нужный метод обработки события.
Например метод ***keyPressEvent()*** получает указатель на объект класса ***QKeyEvent***.

Когда происходит событие, то для его представления Qt создаёт объект события, - создавая экземпляр соответствующего подкласса **QEvent**, - и доставляет его отдельному экземпляру класса *QObject* (или одного из его подклассов), вызывая его функцию:

virtual bool event(QEvent *e)

Эта функция не обрабатывает событие сама; **основываясь на типе** доставленного события, она вызывает **обработчик событий для данного конкретного типа** события и отправляет ответ на основе того, будет ли событие принято или проигнорировано.

Обработка событий

Обычный способ доставки события - вызов виртуальной функции. Например, QPaintEvent доставляется вызовом **QWidget::paintEvent().** Эта виртуальная функция отвечает за соответствующее взаимодействие, обычно перерисовывая виджет. Если в своей реализации виртуальной функции вы не выполнили всю необходимую работу, вам может понадобиться вызывать реализацию из базового класса.

Если вы хотите заменить функцию базового класса, вы должны реализовать её самостоятельно. Однако если вы хотите лишь расширить функциональность базового класса, тогда реализуйте то что вы хотите и вызовите базовый класс, чтобы получить поведение по умолчанию для любых случаев, которые вы не хотите обрабатывать.

Код обрабатывает щелчки левой кнопкой мыши на пользовательском виджете флажка в то время, как щелчки остальными кнопками передаются в базовый класс QCheckBox:

```
void MyCheckBox::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton)
        {
      // здесь обрабатываем левую кнопку мыши
    } else {
      // передаём остальные кнопки в базовый класс
       QCheckBox::mousePressEvent(event);
    }
}
```

События поступают к объектам в функцию **event()**, унаследованную от ***QObject***. Реализация функции **event()** в **QWidget** передает наиболее употребим типы событий специализированным обработчикам, таким как ***mousePressEvent(), keyPressEvent() и paintEvent(),*** остальные события игнорируются.

События клавиатуры. QKeyEvent

Объект **QKyeEvent** – содержит данные о событиях клавиатуры. Он дает информацию о клавише, вызвавшей событие, а так же ASCII – код отображаемого клавишей символа. Объект события передается в методы:
QWidget::keyPressEvent() ,
QWidget::keyReleaseEvent().
Клавиша <Tab>+ <Shift> - используются методом обработки **QWidget::event()** для передачи фокуса след. виджету.

Метод ***keyPressEvent ( )*** вызывается каждый раз при нажатии одной из клавиш на клавиатуре, а метод ***keyReleaseEvent ()*** - при ее отпускании.

В методе обработки события с помощью метода **QKeyEvent::key ()** можно определить, какая из клавиш его инициировала. Этот метод возвращает значение целого типа, которое можно сравнить с константами клавиш, определенными в классе Qt (см. далее).

Если необходимо узнать, были ли в момент наступления события совместно с клавишей нажаты клавиши модификаторов, например <Shift>, <Ctrl> или <Alt>, то это можно проверить с помощью метода **modifiers** ().

С помощью метода **text ()** можно узнать Unicode-текст, полученный вследствие нажатия клавиши. Этот метод может оказаться полезным в том случае, если в виджете потребуется обеспечить ввод с клавиатуры. Для клавиш модификаторов метод вернет пустую строку. В таком случае нужно воспользоваться методом **key** (), который будет содержать код клавиши.

События клавиатуры. QKeyEvent

Метод для обработки событий клавиатуры класса, унаследованного от класса QWidget, может выглядеть следующим образом:

```
void MyWidget::keyPressEvent(QKeyEvent* pe)
{
  switch (pe->key()) {
      case Qt::Key_Z:
            if (pe->modifiers() & Qt::ShiftModifier) {
              // Выполнить какие-либо действия
             }
            else { // Выполнить какие-либо действия
             }
            break;
      default:
             QWidget::keyPressEvent(pe); //Передать событие
            дальше
   }
}
```

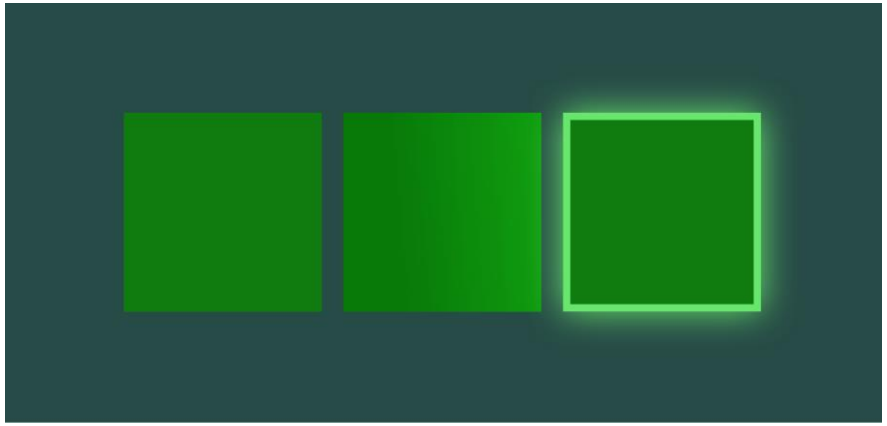
Константы клавиш:

Key_Space	20
Key_B	42
Key_Insert	1000006
Key_C	43
Key_F	46
Key_Delete	1000007
Key_H	48
Key_Right	1000014
Key_Up	1000013

В этом примере проверяется, не нажаты ли совместно клавиши **<Z> и <Shift>**

QFocusEvent

Когда пользователь набирает что-нибудь на клавиатуре, информацию о нажатых клавишах может принимать только один виджет. Если виджет в этот момент выбран для ввода с клавиатуры, то говорят, что он **находится в фокусе**. Объект события фокуса ***QFocusEvent*** передается в методы обработки сообщений ***focusInEvent ()*** и ***focusOutEvent ()***. Этот объект не содержит значимой информации. Основное назначение класса ***QFocusEvent*** – сообщить о получении или потере виджетом фокуса, для того чтобы можно было, например, изменить его внешний вид. Эти ·методы вызываются в том случае, когда виджет получает (***focusInEvent ()***) ИЛИ теряет (***focusOutEvent ())***



События отрисовки. QPaintEvent

В объекте класса **QPaintEvent** передается информация для перерисовки всего изображения или его части. Событие возникает тогда, когда виджет впервые отображается на экране явным или неявным вызовом метода ***show ()***, а также в результате вызова методов ***repaint ()*** и ***update ()*** . Объект события передается в метод ***paintEvent (),*** в котором реализуется отображение самого виджета. В большинстве случаев этот метод используется для полной перерисовки виджета. Для маленьких виджетов такой подход вполне приемлем, но для виджетов больших размеров рациональнее перерисовывать только отдельную область, действительно нуждающуюся в этом. Для получения координат и размеров такого участка вызывается метод ***region ().*** Вызовом метода ***contains ()*** можно проверить, находится ли объект в заданной области.

```
MyClass::paintEvent(QPaintEvent* pe)
{
        QPainter painter(this);
        QRect r(40, 40, 100, 100);
        if (pe->region().contains(r)) {
                painter.drawRect(r);
            }
}
```

События мыши. QMouseEvent

Мышь дает возможность пользователю указывать на объекты, находящиеся на экране компьютера, и с ее помощью можно проводить различные манипуляции над объектами, которые невозможно или неудобно выполнять с помощью клавиатуры.

Реализация событий мыши сложнее других, поскольку программа должна определять, какая кнопка мыши нажата, удерживается она или нет, был ли выполнен двойной щелчок, и какие клавиши клавиатуры были дополнительно нажаты в момент возникновения события.

Объект этого класса содержит информацию о событии, вызванном действием мыши, и хранит в себе информацию о позиции указателя мыши в момент вызова события, статус кнопок мыши и даже некоторых клавиш клавиатуры. Этот объект передается в методы: ***mousePressEvent (), mouseMoveEvent (), mouseReleaseEvent () и mouseDoubleClickEvent ().***

Для определения местоположения указателя мыши в момент возникновения события используют методы **globalx ()**, **globalY(), x()** и **y()**, которые возвращают целые значения, а также методами **pos ()** или **globalPos ()**.
Метод **pos ()** класса ***QMouseEvent*** возвращает позицию указателя мыши в момент наступления события относительно *левого верхнего угла виджета*. Если нужна *абсолютная позиция* (относительно левого верхнего угла экрана), то ее получают с помощью метода **globalPos ().**

Класс QWheelEvent.

Если используются мыши, оснащенные колесиком, рекомендуется реализовывать метод для обработки события прокрутки колеса - ***QWheelEvent***, который унаследован от **QInputEvent.**

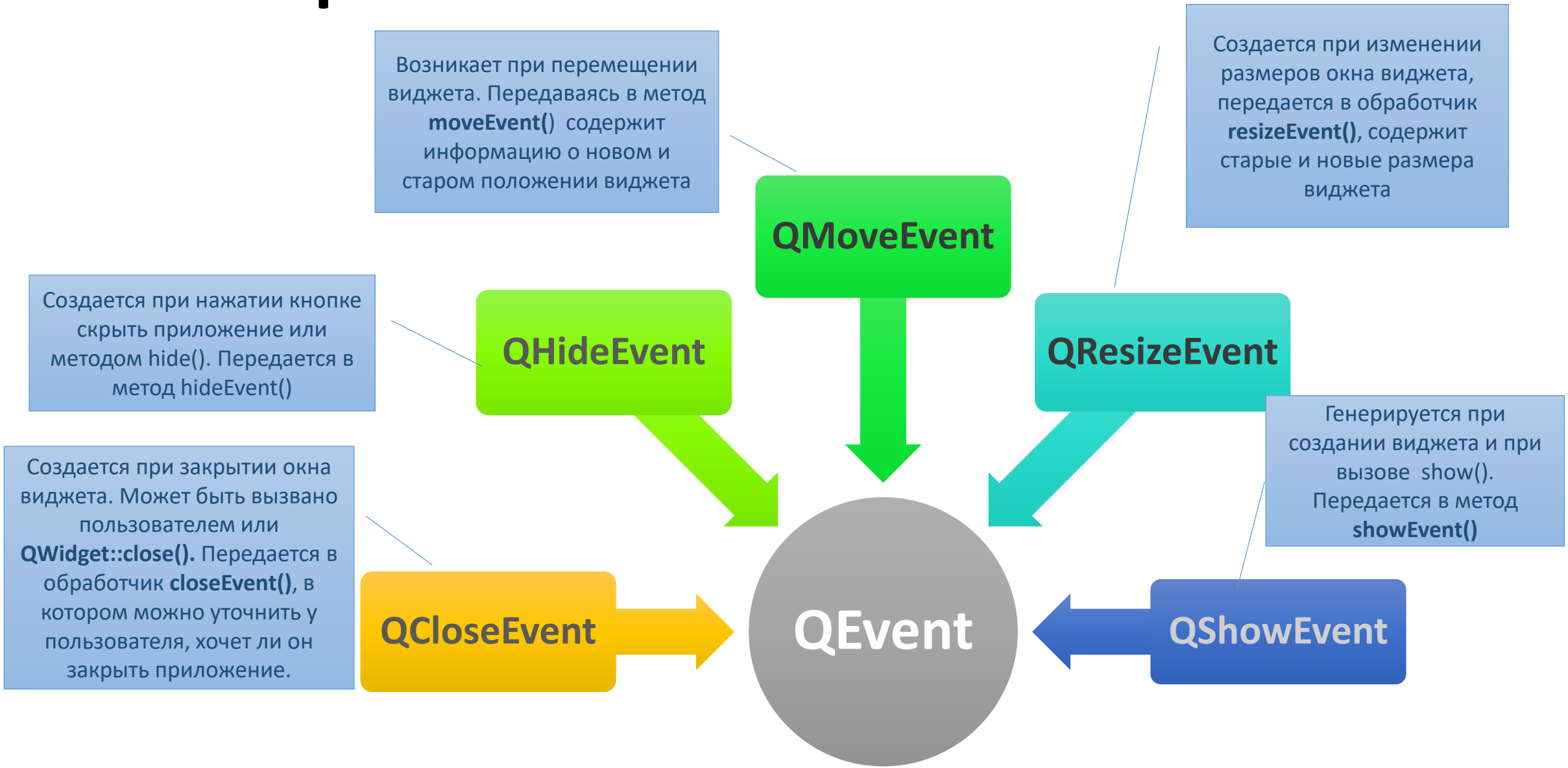
Объект класса ***QWheelEvent*** содержит информацию о событии, вызванном колесиком мыши.
Объект события передается в метод ***wheelEvent ()*** и содержит информацию об угле и направлении, в котором было повернуто колесико, а также о позиции указателя мыши, статусе кнопок мыши и некоторых клавиш клавиатуры. Наряду с методами ***buttons ()***, ***pos ()*** и ***globalPos ()***, которые полностью идентичны методам класса события ***QMouseEvent***, в классе ***QWheelEvent*** имеется метод ***angleDelta ()***, с помощью которого можно узнать угол поворота колесика мыши. Положительное значение говорит о том, что колесико было повернуто ОТ себя, а отрицательное значение - НА себя.

События таймера. QTimerEvent

Объект класса ***QTimerEvent*** содержит информацию о событии, инициированном таймером.
Этот объект передается в метод обработки события ***timerEvent ().*** Объект события содержит идентификационный номер таймера. Например, для класса, унаследованного от класса **QWidget**, метод обработки этого события может выглядеть следующим образом:

```
void MyClass::timerEvent(QTimerEvent* e)
{
        if (event->timerId() == myTimerId) {
            // Выполнить какие-либо действия
        }
        else {
                QWidget::timerEvent(e);
                // Передать событие дальше
                }
}
```

Некоторые классы событий



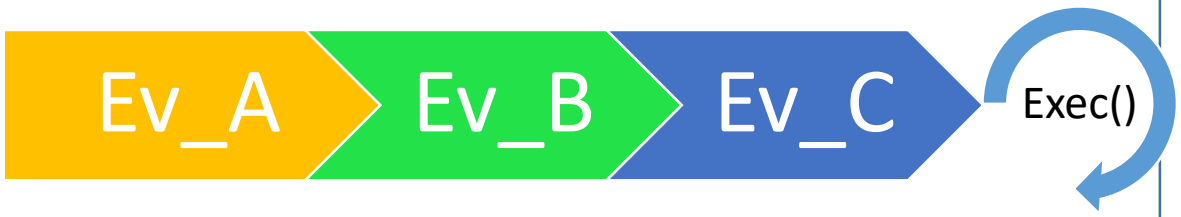
Пользовательские классы событий

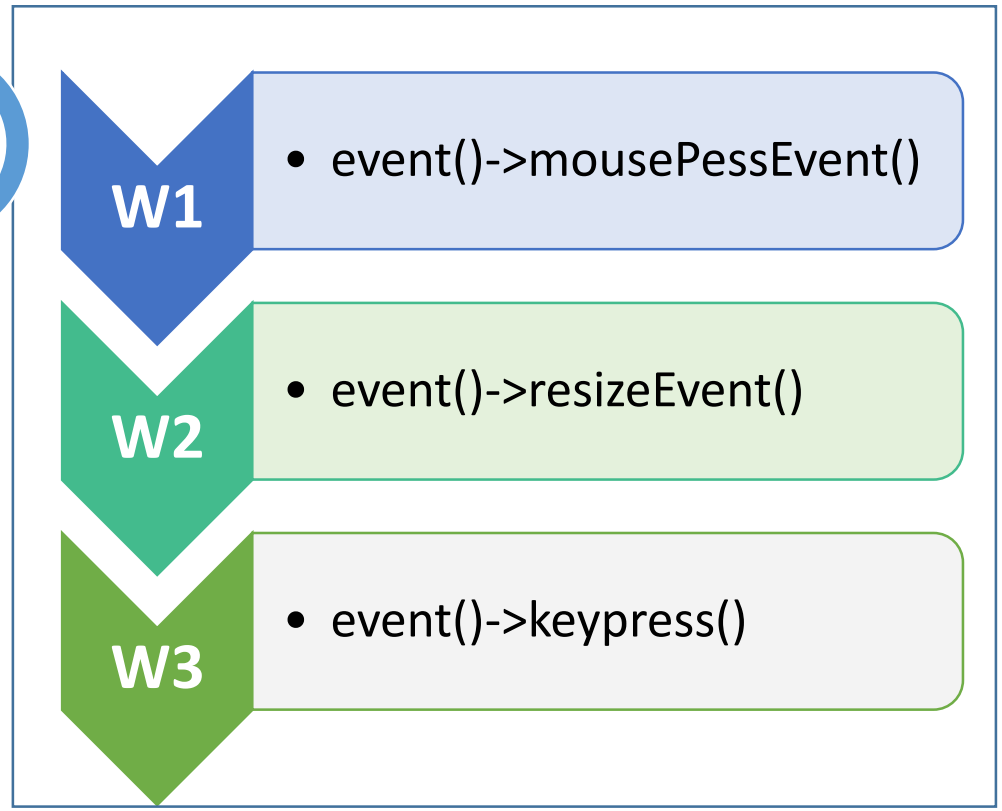
Если вам не будет хватать событий, предоставляемых Qt, и понадобится определить свое собственное, то необходимо - унаследовать базовый класс для всех событий ***QEvent***. В конструкторе ***QEvent*** нужно передать идентификационный номер для типа события, который должен быть больше, чем значение ***QEvent ::User*** (= 1000), - чтобы не создать конфликт с уже определенными типами. В созданном событии можно реализовать все необходимые вам методы для передачи доп. информации.

```
class MyEvent : public QEvent {
public:
        MyEvent () :QEvent ( (Type) ( QEvent :: User + 200)
        {
        }
        QString info ()
        {
                return “CustomEvent”;
        }
};
```

Свои собственные события можно высылать с помощью методов:
[static] bool QCoreApplication::sendEvent(QObject *receiver, QEvent *event)
или
[static] void QCoreApplication::postEvent(QObject *receiver, QEvent *event, int priority = Qt::NormalEventPriority),
а получать методами ***QObject::event ()*** ИЛИ ***QObject::customEvent ()*** .

Переопределение метода event()






```
bool MyClass::event(QEvent* pe)
{
        if (pe->type () == QEvent::KeyPress) {
                QKeyEvent* pKeyEvent = static_cast<QKeyEvent*>(pe);
                if (pKeyEvent->key() == Qt::Key_Tab) {
                        // Вьмолнить какие-либо действия
                        return true; } }
        if (pe->type () == QEvent:: Hide) {
                // Выполнить какие-либо действия
                return true;
        }
        return QWidget::event(pe);
}
```

Метод возвращает *true* в том случае, если событие было обработано и не требует передачи дальше. После этого событие удаляется из очереди событий. При возвращении *false* событие будет передано дальше - виджету-предку. Если ни один из предков не сможет обработано – оно игнорируется и удаляется из очереди.

Метод processEvents()

При интенсивных действиях в вашей программе может произойти «Замирание» графического интерфейса и он не сможет обрабатывать действия пользователя.

Один из лучших вариантов решения этой проблемы - исполнение подобного кода в отдельном потоке . Более простой способ- вызов метода ***QCoreApplication::processEvents (),*** который позаботится о том, чтобы все накопившиеся в очереди события были обработаны.

```
for (int i = 0; i < 1000; ++i)
{
    // Выполнить трудоемкие вычисления
    qApp->processEvents(); //Доставить накопившиеся события
}
```

Теперь в каждой итерации цикла после выполнения действий осуществляется обработка событий, что дает программе возможность перед выполнением очередных действий «вдохнуть воздух» и отреагировать на действия пользователя.

Фильтры событий

Событие передается тому объекту, над которым было осуществлено действие, но иногда требуется его обработка в другом объекте. В библиотеке Qt предусмотрен механизм перехвата событий, который позволяет объекту фильтра просматривать события раньше объекта, которому они предназначены, и принимать решение по своему усмотрению - обрабатывать их и/или передавать дальше. Такой мониторинг осуществляется с помощью метода
QObject:: installEventFilter (QObject *),
в который передается указатель на объект, осуществляющий фильтрацию событий.

Фильтры событий можно использовать, например, в тех случаях, когда нужно добавить функциональность к каким-либо уже реализованным классам, не наследуя при этом каждый из них. После реализации класса фильтра его объекты можно будет устанавливать в любых объектах, созданных от наследующих ***QObject*** классов. Это позволит сэкономить время на реализацию.

Важно то, что установка фильтров событий происходит не на уровне классов, а на уровне самих объектов. Это дает возможность вместо того, чтобы наследовать класс или изменять уже имеющийся (что не всегда возможно), просто воспользоваться объектом фильтра.
Для настройки на определенные события необходимо создать класс фильтра, установив его в нужном объекте. Все получаемые события и их обработка будут касаться только тех объектов, в которых установлены фильтры.

Реализация фильтра

Чтобы реализовать класс фильтра, нужно унаследовать класс ***QObject*** и переопределить метод ***eventFilter ().*** Этот метод будет вызываться при каждом событии, предназначенном для объекта, в котором установлен фильтр событий до его получения.

Метод имеет два параметра:
1) - это указатель на объект, для которого предназначено событие;
2) - указатель на сам объект события.

Если метод ***eventFilter ()*** возвращает значение true, то это означает, что событие не должно передаваться дальше, а возвращение false означает что событие должно быть передано объекту, для которого оно и предназначалось.

Генерация события

Иногда возникает необходимость в событиях, созданных искусственно. Например, это полезно при отладке своей программы, чтобы имитировать действия пользователя.

Для генерации события можно воспользоваться одним из двух статических методов класса ***QCoreApplication:***
sendEvent () или ***postEvent ().***
Оба метода получают в качестве параметров указатель на объект, которому посылается событие, и адрес объекта события.
Разница между ними состоит в том, что метод ***sendEvent ()*** отправляет событие без задержек, то есть его вызов приводит к немедленному вызову метода события, в то время как метод ***postEvent ()*** помещает его в очередь для дальнейшей обработки.

Практика.

- Проект обработка события мыши: «MouseObserver»
- Фильтр событий: «FilterEvent»
- Генерация событие: «ImitEvent»

На сегодня всё