



ГЛАВА 19

Обработка сигналов и событий

При взаимодействии пользователя с окном возникают *события* — своего рода извещения о том, что пользователь выполнил какое-либо действие или в самой системе возникло некоторое условие. В ответ на события система генерирует определенные *сигналы*, которые можно рассматривать как представления системных событий внутри библиотеки PyQt.

Сигналы являются важнейшей составляющей приложения с графическим интерфейсом, поэтому необходимо знать, как назначить обработчик сигнала, как удалить его, а также уметь правильно обработать событие. Сигналы, которые генерирует тот или иной компонент, мы будем рассматривать при изучении конкретного компонента.

19.1. Назначение обработчиков сигналов

Чтобы обработать какой-либо сигнал, необходимо сопоставить ему функцию или метод класса, который будет вызван при возникновении события и станет его обработчиком.

Каждый сигнал, поддерживаемый классом, соответствует одноименному атрибуту этого класса (отметим, что это именно атрибуты класса, а не атрибуты экземпляра). Так, сигнал `clicked`, генерируемый при щелчке мышью, соответствует атрибуту `clicked`. Каждый такой атрибут хранит экземпляр особого класса, представляющего соответствующий сигнал.

Чтобы назначить сигналу обработчик, следует использовать метод `connect()`, унаследованный от класса `QObject`. Форматы вызова этого метода таковы:

```
<Компонент>.«Сигнал» .connect (<Обработчик> [, <Тип соединения>])  
<Компонент>.«Сигнал» [<Тип>] .connect (<Обработчик> [, <Тип соединения>])
```

Здесь мы назначаем *«обработчику»* для параметра *«сигнал»*, генерируемого параметром *«Компонент»*. В качестве обработчика можно указать:

- ◆ ссылку на пользовательскую функцию;
- ◆ ссылку на метод класса;
- ◆ ссылку на экземпляр класса, в котором определен метод `__call__()`;
- ◆ анонимную функцию;
- ◆ ссылку на слот класса.

Вот пример назначения функции `on_clicked_button()` в качестве обработчика сигнала `clicked` кнопки `button`:

```
button.clicked.connect (on_clicked_button)
```

Сигналы могут принимать произвольное число параметров, каждый из которых может относиться к любому типу данных. При этом бывает и так, что в классе существуют два сигнала с одинаковыми наименованиями, но разными наборами параметров. Тогда следует дополнительно в квадратных скобках указать <Тип> данных, принимаемых сигналом, — либо просто написав его наименование, либо задав его в виде строки. Например, оба следующих выражения назначают обработчик сигнала, принимающего один параметр логического типа:

```
button.clicked[bool].connect(on_clicked_button)
button.clicked["bool"].connect(on_clicked_button)
```

Одному и тому же сигналу можно назначить произвольное количество обработчиков. Это иллюстрирует код из листинга 19.1, где сигналу `clicked` кнопки назначены сразу четыре обработчика.

Листинг 19.1. Назначение сигналу нескольких обработчиков

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys

def on_clicked():
    print("Кнопка нажата. Функция on_clicked()")

class MyClass():
    def __init__(self, x=0):
        self.x = x
    def __call__(self):
        print("Кнопка нажата. Метод MyClass.__call__()")
        print("x =", self.x)
    def on_clicked(self):
        print("Кнопка нажата. Метод MyClass.on_clicked()")

obj = MyClass()
app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Нажми меня")
# Назначаем обработчиком функцию
button.clicked.connect(on_clicked)
# Назначаем обработчиком метод класса
button.clicked.connect(obj.on_clicked)
# Назначаем обработчиком ссылку на экземпляр класса
button.clicked.connect(MyClass(10))
# Назначаем обработчиком анонимную функцию
button.clicked.connect(lambda: MyClass(5)())
button.show()
sys.exit(app.exec_())
```

В четвертом обработчике мы использовали анонимную функцию. В ней мы сначала создаем объект класса `MyClass`, передав ему в качестве параметра число 5, после чего сразу же вызываем его как функцию, указав после конструктора пустые скобки:

```
button.clicked.connect(lambda: MyClass(5)())
```

Результат выполнения в окне консоли при щелчке на кнопке:

```
Кнопка нажата. Функция on_clicked()
Кнопка нажата. Метод MyClass.on_clicked()
Кнопка нажата. Метод MyClass.__call__()
x = 10
Кнопка нажата. Метод MyClass.__call__()
x = 5
```

Классы PyQt 5 поддерживают ряд методов, специально предназначенных для использования в качестве обработчиков сигналов. Такие методы называются *слотами*. Например, класс QApplication поддерживает слот quit(), завершающий текущее приложение. В листинге 19.2 показан код, использующий этот слот.

Листинг 19.2. Использование слота

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys

app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Завершить работу")
button.clicked.connect(app.quit)
button.show()
sys.exit(app.exec_())
```

Любой пользовательский метод можно сделать слотом, для чего необходимо перед его определением вставить декоратор @pyqtSlot(). Формат декоратора:

```
@QtCore.pyqtSlot(*<Типы данных>, name=None, result=None)
```

В параметре <Типы данных> через запятую указываются типы данных параметров, принимаемых слотом, — например: bool или int. При задании типа данных C++ его название необходимо указать в виде строки. Если метод не принимает параметров, параметр <Типы данных> не указывается. В именованном параметре name можно передать название слота в виде строки — это название станет использоваться вместо названия метода, а если параметр name не задан, название слота будет совпадать с названием метода. Именованный параметр result предназначен для указания типа данных, возвращаемых методом, — если параметр не задан, то метод ничего не возвращает. Чтобы создать перегруженную версию слота, декоратор указывается последовательно несколько раз с разными типами данных. Пример использования декоратора @pyqtSlot() приведен в листинге 19.3.

Листинг 19.3. Использование декоратора @pyqtSlot()

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
import sys

class MyClass(QtCore.QObject):
    def __init__(self):
        QtCore.QObject.__init__(self)
```

```
@QtCore.pyqtSlot()
def on_clicked(self):
    print("Кнопка нажата. Слот on_clicked()")
@QtCore.pyqtSlot(bool, name="myslot")
def on_clicked2(self, status):
    print("Кнопка нажата. Слот myslot(bool)", status)

obj = MyClass()
app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Нажми меня")
button.clicked.connect(obj.on_clicked)
button.clicked.connect(obj.myslot)
button.show()
sys.exit(app.exec_())
```

PyQt не требует обязательного превращения в слот метода, который будет использоваться как обработчик сигнала. Однако это рекомендуется сделать, т. к. вызов слота в этом случае выполняется быстрее, чем вызов обычного метода.

Необязательный параметр <Тип соединения> метода `connect()` определяет тип соединения между сигналом и обработчиком. На этот параметр следует обратить особое внимание при использовании в приложении нескольких потоков, т. к. изменять GUI-поток из другого потока нельзя. В параметре можно указать один из следующих атрибутов класса `QtCore.Qt`:

- ◆ `AutoConnection` — 0 — значение по умолчанию. Если источник сигнала и обработчик находятся в одном потоке, то оно эквивалентно значению `DirectConnection`, а если в разных потоках, то — `QueuedConnection`;
- ◆ `DirectConnection` — 1 — обработчик вызывается сразу после генерации сигнала и выполняется в потоке его источника;
- ◆ `QueuedConnection` — 2 — сигнал помещается в очередь обработки событий, а его обработчик выполняется в потоке приемника сигнала;
- ◆ `BlockingQueuedConnection` — 4 — аналогично значению `QueuedConnection` за тем исключением, что поток блокируется на время обработки сигнала. Обратите внимание, что источник и обработчик сигнала обязательно должны быть расположены в разных потоках;
- ◆ `UniqueConnection` — 0x80 — указывает, что обработчик можно назначить только один раз. Этот атрибут с помощью оператора `|` может быть объединен с любым из представленных ранее флагов:

```
# Эти два обработчика будут успешно назначены и выполнены
button.clicked.connect(on_clicked)
button.clicked.connect(on_clicked)
# А эти два обработчика назначены не будут
button.clicked.connect(on_clicked, QtCore.Qt.AutoConnection | QtCore.Qt.UniqueConnection)
button.clicked.connect(on_clicked, QtCore.Qt.AutoConnection | QtCore.Qt.UniqueConnection)
# Тем не менее, эти два обработчика будут назначены, поскольку они разные
button.clicked.connect(on_clicked, QtCore.Qt.AutoConnection | QtCore.Qt.UniqueConnection)
button.clicked.connect(obj.on_clicked, QtCore.Qt.AutoConnection | QtCore.Qt.UniqueConnection)
```

19.2. Блокировка и удаление обработчика

Для блокировки и удаления обработчиков предназначены следующие методы класса `QObject`:

- ◆ `blockSignals(<Флаг>)` — временно блокирует прием сигналов, если параметр имеет значение `True`, и снимает блокировку, если параметр имеет значение `False`. Метод возвращает логическое представление предыдущего состояния соединения;
- ◆ `signalsBlocked()` — возвращает значение `True`, если блокировка сигналов установлена, и `False` — в противном случае;
- ◆ `disconnect()` — удаляет обработчик. Форматы метода:

```
<Компонент>.«Сигнал».disconnect ([<Обработчик>])  
<Компонент>.«Сигнал»[<Тип>].disconnect ([<Обработчик>])
```

Если параметр `<Обработчик>` не указан, удаляются все обработчики, назначенные ранее, в противном случае удаляется только указанный обработчик. Параметр `<Тип>` указывается лишь в том случае, если существуют сигналы с одинаковыми именами, но принимающие разные параметры:

```
button.clicked.disconnect()  
button.clicked[bool].disconnect(on_clicked_button)  
button.clicked["bool"].disconnect(on_clicked_button)
```

Создадим окно с четырьмя кнопками (листинг 19.4). Для кнопки **Нажми меня** назначим обработчик сигнала `clicked`. Чтобы информировать о нажатии кнопки, выведем сообщение в окно консоли. Для кнопок **Блокировать**, **Разблокировать** и **Удалить обработчик** создадим обработчики, которые будут изменять статус обработчика для кнопки **Нажми меня**.

Листинг 19.4. Блокировка и удаление обработчика

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle("Блокировка и удаление обработчика")
        self.resize(300, 150)
        self.button1 = QtWidgets.QPushButton("Нажми меня")
        self.button2 = QtWidgets.QPushButton("Блокировать")
        self.button3 = QtWidgets.QPushButton("Разблокировать")
        self.button4 = QtWidgets.QPushButton("Удалить обработчик")
        self.button3.setEnabled(False)
        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        vbox.addWidget(self.button3)
        vbox.addWidget(self.button4)
        self.setLayout(vbox)
        self.button1.clicked.connect(self.on_clicked_button1)
```

```
self.button2.clicked.connect(self.on_clicked_button2)
self.button3.clicked.connect(self.on_clicked_button3)
self.button4.clicked.connect(self.on_clicked_button4)

@QtCore.pyqtSlot()
def on_clicked_button1(self):
    print("Нажата кнопка button1")
@QtCore.pyqtSlot()
def on_clicked_button2(self):
    self.button1.blockSignals(True)
    self.button2.setEnabled(False)
    self.button3.setEnabled(True)
@QtCore.pyqtSlot()
def on_clicked_button3(self):
    self.button1.blockSignals(False)
    self.button2.setEnabled(True)
    self.button3.setEnabled(False)
@QtCore.pyqtSlot()
def on_clicked_button4(self):
    self.button1.clicked.disconnect(self.on_clicked_button1)
    self.button2.setEnabled(False)
    self.button3.setEnabled(False)
    self.button4.setEnabled(False)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Если нажать кнопку **Нажми меня**, в окно консоли будет выведена строка Нажата кнопка button1. Нажатие кнопки **Блокировать** производит блокировку обработчика — теперь при нажатии кнопки **Нажми меня** никаких сообщений в окно консоли не выводится. Отменить блокировку можно с помощью кнопки **Разблокировать**. Нажатие кнопки **Удалить обработчик** производит полное удаление обработчика — в этом случае, чтобы обрабатывать нажатие кнопки **Нажми меня**, необходимо заново назначить обработчик.

Также можно отключить генерацию сигнала, сделав компонент недоступным с помощью следующих методов из класса `QWidget`:

- ◆ `setEnabled(<Флаг>)` — если в параметре указано значение `False`, компонент станет недоступным. Чтобы сделать компонент опять доступным, следует передать значение `True`;
- ◆ `setDisabled(<Флаг>)` — если в параметре указано значение `True`, компонент станет недоступным. Чтобы сделать компонент опять доступным, следует передать значение `False`.

Проверить, доступен компонент или нет, позволяет метод `isEnabled()`. Он возвращает значение `True`, если компонент доступен, и `False` — в противном случае.

19.3. Генерация сигналов

В некоторых случаях необходимо сгенерировать сигнал программно. Например, при заполнении последнего текстового поля и нажатии клавиши <Enter> можно имитировать нажатие кнопки **OK** и тем самым выполнить подтверждение ввода пользователя. Осуществить генерацию сигнала из программы позволяет метод `emit()` класса `QObject`. Форматы этого метода:

```
<Компонент>.<Сигнал>.emit ([<Данные>])
<Компонент>.<Сигнал> [<Тип>].emit ([<Данные>])
```

Метод `emit()` всегда вызывается у объекта, которому посыпается сигнал:

```
button.clicked.emit()
```

Сигналу и, соответственно, его обработчику можно передать данные, указав их в вызове метода `emit()`:

```
button.clicked[bool].emit(False)
button.clicked["bool"].emit(False)
```

Также мы можем создавать свои собственные сигналы. Для этого следует определить в классе атрибут, чье имя совпадет с наименованием сигнала. Отметим, что это должен быть атрибут класса, а не экземпляра. Далее мы присвоим вновь созданному атрибуту результат, возвращенный функцией `pyqtSignal()` из модуля `QtCore`. Формат функции:

```
<Объект сигнала> = QtCore.pyqtSignal(*<Типы данных>[, name=<Имя сигнала>])
```

В параметре `<Типы данных>` через запятую указываются названия типов данных, передаваемых сигналу, — например: `bool` или `int`:

```
mysignal1 = QtCore.pyqtSignal(int)
mysignal2 = QtCore.pyqtSignal(int, str)
```

При использовании типа данных C++ его название необходимо указать в виде строки:

```
mysignal3 = QtCore.pyqtSignal("QDate")
```

Если сигнал не принимает параметров, параметр `<Типы данных>` не указывается.

Сигнал может иметь несколько перегруженных версий, различающихся количеством и типом принимаемых параметров. В этом случае типы параметров указываются внутри квадратных скобок. Вот пример сигнала, передающего данные типа `int` или `str`:

```
mysignal4 = QtCore.pyqtSignal([int], [str])
```

По умолчанию название создаваемого сигнала будет совпадать с названием атрибута класса. Однако мы можем указать для сигнала другое название, после чего он будет доступен под двумя названиями: совпадающим с именем атрибута класса и заданным нами. Для указания названия сигнала применяется параметр `name`:

```
mysignal = QtCore.pyqtSignal(int, name="mySignal")
```

В качестве примера создадим окно с двумя кнопками (листинг 19.5), которым назначим обработчики сигнала `clicked` (нажатие кнопки). Внутри обработчика щелчка на первой кнопке сгенерируем два сигнала: первый будет имитировать нажатие второй кнопки, а второй станет пользовательским, привязанным к окну. Внутри обработчиков выведем сообщения в окно консоли.

Листинг 19.5. Генерация сигнала из программы

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets

class MyWindow(QtWidgets.QWidget):
    mysignal = QtCore.pyqtSignal(int, int)
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle("Генерация сигнала из программы")
        self.resize(300, 100)
        self.button1 = QtWidgets.QPushButton("Нажми меня")
        self.button2 = QtWidgets.QPushButton("Кнопка 2")
        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.button1.clicked.connect(self.on_clicked_button1)
        self.button2.clicked.connect(self.on_clicked_button2)
        self.mysignal.connect(self.on_mysignal)
    def on_clicked_button1(self):
        print("Нажата кнопка button1")
        # Генерируем сигналы
        self.button2.clicked[bool].emit(False)
        self.mysignal.emit(10, 20)
    def on_clicked_button2(self):
        print("Нажата кнопка button2")
    def on_mysignal(self, x, y):
        print("Обработан пользовательский сигнал mysignal()")
        print("x =", x, "y =", y)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Результат выполнения после нажатия первой кнопки:

```
Нажата кнопка button1
Нажата кнопка button2
Обработан пользовательский сигнал mysignal()
x = 10 y = 20
```

Вместо конкретного типа принимаемого сигналом параметра можно указать тип QVariant из модуля QtCore. В этом случае при генерации сигнала допускается передавать ему данные любого типа. Вот пример создания и генерирования сигнала, принимающего параметр любого типа:

```
mysignal = QtCore.pyqtSignal(QtCore.QVariant)
...
self.mysignal.emit(20)
```

```
selfmysignal.emit("Привет!")
selfmysignal.emit([1, "2"])
```

Сгенерировать сигнал можно не только с помощью метода `emit()`. Некоторые компоненты предоставляют методы, которые посылают сигнал. Например, у кнопок существует метод `click()`. Используя этот метод, инструкцию:

```
button.clicked.emit()
```

можно записать следующим образом:

```
button.click()
```

Более подробно такие методы мы будем рассматривать при изучении конкретных компонентов.

19.4. Передача данных в обработчик

При назначении обработчика в метод `connect()` передается ссылка на функцию или метод. Если после названия функции (метода) указать внутри круглых скобок какой-либо параметр, то это приведет к вызову функции (метода) и вместо ссылки будет передан результат ее выполнения, что вызовет ошибку. Передать данные в обработчик можно следующими способами:

- ◆ создать анонимную функцию и внутри ее выполнить вызов обработчика с параметрами.

Вот пример передачи обработчику числа 10:

```
self.button1.clicked.connect(lambda : self.on_clicked_button1(10))
```

Если передаваемое обработчику значение вычисляется в процессе выполнения кода, переменную, хранящую это значение, следует указывать в анонимной функции как значение по умолчанию, иначе функции будет передана ссылка на это значение, а не оно само:

```
y = 10
self.button1.clicked.connect(lambda x=y: self.on_clicked_button1(x))
```

- ◆ передать ссылку на экземпляр класса, внутри которого определен метод `__call__()`.

Передаваемое значение указывается в качестве параметра конструктора этого класса:

```
class MyClass():
    def __init__(self, x=0):
        self.x = x
    def __call__(self):
        print("x =", self.x)

...
self.button1.clicked.connect(MyClass(10))
```

- ◆ передать ссылку на обработчик и данные в функцию `partial()` из модуля `functools`.

Формат функции:

```
partial(<Функция>[, *<Неименованные параметры>] [, **<Именованные параметры>])
```

Пример передачи параметра в обработчик:

```
from functools import partial
self.button1.clicked.connect(partial(self.on_clicked_button1, 10))
```

Если при генерации сигнала передается предопределено значение, то оно будет доступно в обработчике после остальных параметров. Назначим обработчик сигнала `clicked`, принимающего логический параметр, и дополнительно передадим число:

```
self.button1.clicked.connect(partial(self.on_clicked_button1, 10))
```

Обработчик будет иметь следующий вид:

```
def on_clicked_button1(self, x, status):
    print("Нажата кнопка button1", x, status)
```

Результат выполнения:

```
Нажата кнопка button1 10 False
```

19.5. Использование таймеров

Таймеры позволяют через заданный интервал времени выполнять метод с предопределенным названием `timerEvent()`. Для назначения таймера используется метод `startTimer()` класса `QObject`. Формат метода:

```
<Id> = <Объект>.startTimer(<Интервал>[, timerType=<Тип таймера>])
```

Параметр `<Интервал>` задает промежуток времени в миллисекундах, по истечении которого выполняется метод `timerEvent()`. Минимальное значение интервала зависит от операционной системы. Если в параметре `<Интервал>` указать значение 0, таймер будет срабатывать много раз при отсутствии других необработанных событий.

Необязательный параметр `timerType` позволяет указать тип таймера в виде одного из атрибутов класса `QtCore.Qt`:

- ◆ `PreciseTimer` — точный таймер, обеспечивающий точность до миллисекунд;
- ◆ `CoarseTimer` — таймер, обеспечивающий точность в пределах 5% от заданного интервала (значение по умолчанию);
- ◆ `VeryCoarseTimer` — «приблизительный» таймер, обеспечивающий точность до секунд.

Метод `startTimer()` возвращает идентификатор таймера, с помощью которого впоследствии можно остановить таймер.

Формат метода `timerEvent()`:

```
timerEvent(self, <Объект класса QTimerEvent>)
```

Внутри него можно получить идентификатор таймера с помощью метода `timerId()` объекта класса `QTimerEvent`.

Чтобы остановить таймер, необходимо воспользоваться методом `killTimer()` класса `QObject`. Формат метода:

```
<Объект>.killTimer(<Id>)
```

В качестве параметра указывается идентификатор, возвращаемый методом `startTimer()`.

Создадим в окне часы, которые будут отображать текущее системное время с точностью до секунды, и добавим возможность запуска и остановки часов с помощью соответствующих кнопок (листинг 19.6).

Листинг 19.6. Вывод времени в окне с точностью до секунды

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
import time

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle("Часы в окне")
        self.resize(200, 100)
        self.timer_id = 0
        self.label = QtWidgets.QLabel("")
        self.label.setAlignment(QtCore.Qt.AlignHCenter)
        self.button1 = QtWidgets.QPushButton("Запустить")
        self.button2 = QtWidgets.QPushButton("Остановить")
        self.button2.setEnabled(False)
        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(self.label)
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.button1.clicked.connect(self.on_clicked_button1)
        self.button2.clicked.connect(self.on_clicked_button2)
    def on_clicked_button1(self):
        # Задаем интервал в 1 секунду и "приближенный" таймер
        self.timer_id = self.startTimer(1000, timerType = QtCore.Qt.VeryCoarseTimer)
        self.button1.setEnabled(False)
        self.button2.setEnabled(True)
    def on_clicked_button2(self):
        if self.timer_id:
            self.killTimer(self.timer_id)
            self.timer_id = 0
        self.button1.setEnabled(True)
        self.button2.setEnabled(False)
    def timerEvent(self, event):
        self.label.setText(time.strftime("%H:%M:%S"))

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Вместо методов startTimer() и killTimer() класса QObject можно воспользоваться классом QTimer из модуля QtCore. Конструктор класса имеет следующий формат:

<Объект> = QTimer([parent=None])

Методы класса:

- ◆ `setInterval(<Интервал>)` — задает промежуток времени в миллисекундах, по истечении которого генерируется сигнал `timeout`. Минимальное значение интервала зависит от операционной системы. Если в параметре `<Интервал>` указать значение 0, таймер будет срабатывать много раз при отсутствии других необработанных сигналов;
- ◆ `start([<Интервал>])` — запускает таймер. В необязательном параметре можно указать промежуток времени в миллисекундах. Если параметр не указан, используется значение, заданное в вызове метода `setInterval()`;
- ◆ `stop()` — останавливает таймер;
- ◆ `isActive()` — возвращает значение `True`, если таймер запущен, и `False` — в противном случае;
- ◆ `timerId()` — возвращает идентификатор таймера, если он запущен, и значение `-1` — в противном случае;
- ◆ `remainingTime()` — возвращает время, оставшееся до очередного срабатывания таймера, в миллисекундах;
- ◆ `interval()` — возвращает установленный интервал;
- ◆ `setSingleShot(<Флаг>)` — если в параметре указано значение `True`, таймер сработает только один раз, в противном случае — будет срабатывать многократно;
- ◆ `isSingleShot()` — возвращает значение `True`, если таймер будет срабатывать только один раз, и `False` — в противном случае;
- ◆ `setTimerType(<Тип таймера>)` — задает тип таймера, который указывается в том же виде, что и в случае вызова метода `startTimer()`;
- ◆ `timerType()` — возвращает тип таймера.

Переделаем предыдущий пример и используем класс `QTimer` вместо методов `startTimer()` и `killTimer()` (листинг 19.7).

Листинг 19.7. Использование класса `QTimer`

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
import time

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setWindowTitle("Использование класса QTimer")
        self.resize(200, 100)
        self.label = QtWidgets.QLabel("")
        self.label.setAlignment(QtCore.Qt.AlignHCenter)
        self.button1 = QtWidgets.QPushButton("Запустить")
        self.button2 = QtWidgets.QPushButton("Остановить")
        self.button2.setEnabled(False)
        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(self.label)
        vbox.addWidget(self.button1)
```

```
vbox.addWidget(self.button2)
self.setLayout(vbox)
self.button1.clicked.connect(self.on_clicked_button1)
self.button2.clicked.connect(self.on_clicked_button2)
self.timer = QtCore.QTimer()
self.timer.timeout.connect(self.on_timeout);
def on_clicked_button1(self):
    self.timer.start(1000) # 1 секунда
    self.button1.setEnabled(False)
    self.button2.setEnabled(True)
def on_clicked_button2(self):
    self.timer.stop()
    self.button1.setEnabled(True)
    self.button2.setEnabled(False)
def on_timeout(self):
    self.label.setText(time.strftime("%H:%M:%S"))

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Статический метод `singleShot()` класса `QTimer` запускает таймер, настраивает его для однократного срабатывания и указывает функцию или метод, который будет вызван по истечении заданного интервала. Формат вызова этого метода следующий:

```
singleShot(<Интервал>[, <Тип таймера>], <Функция или метод>)
```

Примеры использования этого статического метода:

```
QtCore.QTimer.singleShot(1000, self.on_timeout)
QtCore.QTimer.singleShot(1000, QtWidgets.qApp.quit)
```

19.6. Перехват всех событий

В предыдущих разделах мы рассмотрели обработку сигналов, которые позволяют обмениваться сообщениями между компонентами. Обработка внешних событий — например, нажатий клавиш, — осуществляется несколько иначе. Чтобы обработать событие, необходимо наследовать класс и переопределить в нем метод со специальным названием, — так, чтобы обработать нажатие клавиши, следует переопределить метод `keyPressEvent()`. Специальные методы принимают объект, содержащий детальную информацию о событии, — например, код нажатой клавиши. Все эти объекты являются наследниками класса `QEvent` и наследуют следующие методы:

- ◆ `accept()` — устанавливает флаг, разрешающий дальнейшую обработку события. Скажем, если в методе `closeEvent()` вызвать метод `accept()` через объект события, окно будет закрыто. Этот флаг обычно установлен по умолчанию;
- ◆ `ignore()` — сбрасывает флаг, разрешающий дальнейшую обработку события. Так, если в методе `closeEvent()` вызвать метод `ignore()` через объект события, окно закрыто не будет;

- ◆ `setAccepted(<Флаг>)` — если в качестве параметра указано значение `True`, флаг, разрешающий дальнейшую обработку события, будет установлен (аналогично вызову метода `accept()`), а если `False` — сброшен (аналогично вызову метода `ignore()`);
- ◆ `isAccepted()` — возвращает текущее состояние флага, разрешающего дальнейшую обработку события;
- ◆ `spontaneous()` — возвращает `True`, если событие сгенерировано системой, и `False` — если внутри программы;
- ◆ `type()` — возвращает тип события. Приведем основные типы событий (полный их список содержится в документации по классу `QEvent` на странице <https://doc.qt.io/qt-5/qevent.html>):
 - 0 — нет события;
 - 1 — `Timer` — событие таймера;
 - 2 — `MouseButtonPress` — нажата кнопка мыши;
 - 3 — `MouseButtonRelease` — отпущена кнопка мыши;
 - 4 — `MouseButtonDoubleClick` — двойной щелчок мышью;
 - 5 — `MouseMove` — перемещение мыши;
 - 6 — `KeyPress` — клавиша на клавиатуре нажата;
 - 7 — `KeyRelease` — клавиша на клавиатуре отпущена;
 - 8 — `FocusIn` — получен фокус ввода с клавиатуры;
 - 9 — `FocusOut` — потерян фокус ввода с клавиатуры;
 - 10 — `Enter` — указатель мыши входит в область компонента;
 - 11 — `Leave` — указатель мыши покидает область компонента;
 - 12 — `Paint` — перерисовка компонента;
 - 13 — `Move` — позиция компонента изменилась;
 - 14 — `Resize` — изменился размер компонента;
 - 17 — `Show` — компонент отображен;
 - 18 — `Hide` — компонент скрыт;
 - 19 — `Close` — окно закрыто;
 - 24 — `WindowActivate` — окно стало активным;
 - 25 — `WindowDeactivate` — окно стало неактивным;
 - 26 — `ShowToParent` — дочерний компонент отображен;
 - 27 — `HideToParent` — дочерний компонент скрыт;
 - 31 — `Wheel` — прокрученено колесико мыши;
 - 40 — `Clipboard` — содержимое буфера обмена изменено;
 - 60 — `DragEnter` — указатель мыши входит в область компонента при операции перетаскивания;
 - 61 — `DragMove` — производится операция перетаскивания;
 - 62 — `DragLeave` — указатель мыши покидает область компонента при операции перетаскивания;

- 63 — Drop — операция перетаскивания завершена;
- 68 — ChildAdded — добавлен дочерний компонент;
- 69 — ChildPolished — производится настройка дочернего компонента;
- 71 — ChildRemoved — удален дочерний компонент;
- 74 — PolishRequest — компонент настроен;
- 75 — Polish — производится настройка компонента;
- 82 — ContextMenu — событие контекстного меню;
- 99 — ActivationChange — изменился статус активности окна верхнего уровня;
- 103 — WindowBlocked — окно блокировано модальным окном;
- 104 — WindowUnblocked — текущее окно разблокировано после закрытия модального окна;
- 105 — WindowStateChange — статус окна изменился;
- 121 — ApplicationActivate — приложение стало доступно пользователю;
- 122 — ApplicationDeactivate — приложение стало недоступно пользователю;
- 1000 — User — пользовательское событие;
- 65535 — MaxUser — максимальный идентификатор пользовательского события.

Статический метод `registerEventType(<Число>)` позволяет зарегистрировать пользовательский тип события, возвращая идентификатор зарегистрированного события. В качестве параметра можно указать значение в пределах от `QEvent.User` (1000) до `QEvent.MaxUser` (65535).

Перехват всех событий осуществляется с помощью метода с предопределенным названием `event(self, <event>)`. Через параметр `<event>` доступен объект с дополнительной информацией о событии. Этот объект различен для разных типов событий — например, для события `MouseButtonPress` объект будет экземпляром класса `QMouseEvent`, а для события `KeyPress` — экземпляром класса `QKeyEvent`. Методы, поддерживаемые всеми этими классами, мы рассмотрим в следующих разделах.

Из метода `event()` следует вернуть в качестве результата значение `True`, если событие было обработано, и `False` — в противном случае. Если возвращается значение `True`, то родительский компонент не получит событие. Чтобы продолжить распространение события, необходимо вызвать метод `event()` базового класса и передать ему текущий объект события. Обычно это делается так:

```
return QtWidgets.QWidget.event(self, e)
```

В этом случае пользовательский класс является наследником класса `QWidget` и переопределяет метод `event()`. Если вы наследуете другой класс, следует вызывать метод именно этого класса. Например, при наследовании класса `QLabel` инструкция будет выглядеть так:

```
return QtWidgets.QLabel.event(self, e)
```

Пример перехвата нажатия клавиши, щелчка мышью и закрытия окна показан в листинге 19.8.

Листинг 19.8. Перехват всех событий

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
```

```
class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def event(self, e):
        if e.type() == QtCore.QEvent.KeyPress:
            print("Нажата клавиша на клавиатуре")
            print("Код:", e.key(), ", текст:", e.text())
        elif e.type() == QtCore.QEvent.Close:
            print("Окно закрыто")
        elif e.type() == QtCore.QEvent.MouseButtonPress:
            print("Щелчок мышью. Координаты:", e.x(), e.y())
        return QtWidgets.QWidget.event(self, e) # Отправляем дальше

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

19.7. События окна

Перехватывать все события следует только в самом крайнем случае. В обычных ситуациях нужно использовать методы, предназначенные для обработки определенного события, — например, чтобы обработать закрытие окна, достаточно переопределить метод `closeEvent()`. Методы, которые требуется переопределять для обработки событий окна, мы сейчас и рассмотрим.

19.7.1. Изменение состояния окна

Отследить изменение состояния окна (сворачивание, разворачивание, скрытие и отображение) позволяют следующие методы:

- ◆ `changeEvent(self, <event>)` — вызывается при изменении состояния окна, приложения или компонента, заголовка окна, его палитры, статуса активности окна верхнего уровня, языка, локали и др. (полный список смотрите в документации). При обработке события `WindowStateChange` через параметр `<event>` доступен экземпляр класса `QWindowStateChangeEvent`. Этот класс поддерживает только метод `oldState()`, с помощью которого можно получить предыдущее состояние окна;
- ◆ `showEvent(self, <event>)` — вызывается при отображении компонента. Через параметр `<event>` доступен экземпляр класса `QShowEvent`;
- ◆ `hideEvent(self, <event>)` — вызывается при скрытии компонента. Через параметр `<event>` доступен экземпляр класса `QHideEvent`.

Для примера выведем в консоль текущее состояние окна при его сворачивании, разворачивании, скрытии и отображении (листинг 19.9).

Листинг 19.9. Отслеживание состояния окна

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def changeEvent(self, e):
        if e.type() == QtCore.QEvent.WindowStateChange:
            if self.isMinimized():
                print("Окно свернуто")
            elif self.isMaximized():
                print("Окно раскрыто до максимальных размеров")
            elif self.isFullScreen():
                print("Полноэкранный режим")
            elif self.isActiveWindow():
                print("Окно находится в фокусе ввода")
        QtWidgets.QWidget.changeEvent(self, e) # Отправляем дальше
    def showEvent(self, e):
        print("Окно отображено")
        QtWidgets.QWidget.showEvent(self, e) # Отправляем дальше
    def hideEvent(self, e):
        print("Окно скрыто")
        QtWidgets.QWidget.hideEvent(self, e) # Отправляем дальше

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

19.7.2. Изменение положения и размеров окна

При перемещении и изменении размеров окна вызываются следующие методы:

- ◆ moveEvent(self, <event>) — непрерывно вызывается при перемещении окна. Через параметр <event> доступен экземпляр класса QMoveEvent. Получить координаты окна позволяют следующие методы этого класса:
 - pos() — возвращает экземпляр класса QPoint с текущими координатами;
 - oldPos() — возвращает экземпляр класса QPoint с предыдущими координатами;
- ◆ resizeEvent(self, <event>) — непрерывно вызывается при изменении размеров окна. Через параметр <event> доступен экземпляр класса QResizeEvent. Получить размеры окна позволяют следующие методы этого класса:
 - size() — возвращает экземпляр класса QSize с текущими размерами;
 - oldSize() — возвращает экземпляр класса QSize с предыдущими размерами.

Пример обработки изменения положения окна и его размера показан в листинге 19.10.

Листинг 19.10. Отслеживание смены положения и размеров окна

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def moveEvent(self, e):
        print("x = {0}; y = {1}".format(e.pos().x(), e.pos().y()))
        QtWidgets.QWidget.moveEvent(self, e)    # Отправляем дальше
    def resizeEvent(self, e):
        print("w = {0}; h = {1}".format(e.size().width(),
                                       e.size().height()))
        QtWidgets.QWidget.resizeEvent(self, e) # Отправляем дальше

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

19.7.3. Перерисовка окна или его части

Когда компонент (или часть компонента) становится видимым, требуется выполнить его перерисовку. В этом случае вызывается метод с названием `paintEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QPaintEvent`, который поддерживает следующие методы:

- ◆ `rect()` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, которую требуется перерисовать;
- ◆ `region()` — возвращает экземпляр класса `QRegion` с регионом, требующим перерисовки.

С помощью этих методов можно получить координаты области, которая, например, была ранее перекрыта другим окном и теперь вновь оказалась в зоне видимости. Перерисовывая только область, а не весь компонент, можно заметно повысить быстродействие приложения. Следует также заметить, что в целях эффективности последовательность событий перерисовки может быть объединена в одно событие с общей областью перерисовки.

В некоторых случаях перерисовку окна необходимо выполнить вне зависимости от внешних действий системы или пользователя — например, при изменении каких-либо значений требуется обновить график. Вызвать событие перерисовки компонента позволяют следующие методы класса `QWidget`:

- ◆ `repaint()` — немедленно вызывает метод `paintEvent()` для перерисовки компонента при условии, что таковой не скрыт, и обновление не было запрещено вызовом метода `setUpdatesEnabled()`. Форматы метода:

```
repaint()  
repaint(<X>, <Y>, <Ширина>, <Высота>)  
repaint(<QRect>)  
repaint(<QRegion>)
```

Первый формат вызова выполняет перерисовку всего компонента, а остальные — только области с указанными координатами;

- ◆ `update()` — посылает сообщение о необходимости перерисовки компонента при условии, что компонент не скрыт и обновление не запрещено. Событие будет обработано на следующей итерации основного цикла приложения. Если посылаются сразу несколько сообщений, они объединяются в одно, благодаря чему можно избежать неприятного мерцания. Рекомендуется использовать этот метод вместо метода `repaint()`. Форматы вызова:

```
update()  
update(<X>, <Y>, <Ширина>, <Высота>)  
update(<QRect>)  
update(<QRegion>)
```

19.7.4. Предотвращение закрытия окна

При закрытии окна нажатием кнопки **Закрыть** в его заголовке или вызовом метода `close()` в коде выполняется метод `closeEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QCloseEvent`. Чтобы предотвратить закрытие окна, у объекта события следует вызвать метод `ignore()`, в противном случае — метод `accept()`.

В качестве примера по нажатию кнопки **Закрыть** выведем стандартное диалоговое окно с запросом подтверждения закрытия окна (листинг 19.11). Если пользователь нажмет кнопку **Да**, закроем окно, а если щелкнет кнопку **Нет** или просто закроет диалоговое окно, не будем его закрывать.

Листинг 19.11. Обработка закрытия окна

```
# -*- coding: utf-8 -*-  
from PyQt5 import QtWidgets  
  
class MyWindow(QtWidgets.QWidget):  
    def __init__(self, parent=None):  
        QtWidgets.QWidget.__init__(self, parent)  
        self.resize(300, 100)  
    def closeEvent(self, e):  
        result = QtWidgets.QMessageBox.question(self,  
                                                "Подтверждение закрытия окна",  
                                                "Вы действительно хотите закрыть окно?",  
                                                QtWidgets.QMessageBox.Yes | QtWidgets.QMessageBox.No,  
                                                QtWidgets.QMessageBox.No)  
        if result == QtWidgets.QMessageBox.Yes:  
            e.accept()  
            QtWidgets.QWidget.closeEvent(self, e)  
        else:  
            e.ignore()
```

```
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

19.8. События клавиатуры

События клавиатуры обрабатываются очень часто. Например, при нажатии клавиши <F1> выводится справочная информация, при нажатии клавиши <Enter> в одностороннем текстовом поле фокус ввода переносится на другой компонент, и т. д. Рассмотрим события клавиатуры подробно.

19.8.1. Установка фокуса ввода

В текущий момент времени только один компонент (или вообще ни одного) может иметь фокус ввода. Для управления фокусом ввода предназначены следующие методы класса `QWidget`:

- ◆ `setFocus ([<Причина>])` — устанавливает фокус ввода, если компонент находится в активном окне. В параметре <Причина> можно указать причину изменения фокуса ввода в виде одного из следующих атрибутов класса `QtCore.Qt`:
 - `MouseFocusReason` — 0 — фокус изменен с помощью мыши;
 - `TabFocusReason` — 1 — нажата клавиша <Tab>;
 - `BacktabFocusReason` — 2 — нажата комбинация клавиш <Shift>+<Tab>;
 - `ActiveWindowFocusReason` — 3 — окно стало активным или неактивным;
 - `PopupFocusReason` — 4 — открыто или закрыто всплывающее окно;
 - `ShortcutFocusReason` — 5 — нажата комбинация клавиш быстрого доступа;
 - `MenuBarFocusReason` — 6 — фокус изменился из-за меню;
 - `OtherFocusReason` — 7 — другая причина;
- ◆ `clearFocus()` — убирает фокус ввода с компонента;
- ◆ `hasFocus ()` — возвращает значение `True`, если компонент имеет фокус ввода, и `False` — в противном случае;
- ◆ `focusWidget ()` — возвращает ссылку на последний компонент, для которого вызывался метод `setFocus ()`. Для компонентов верхнего уровня возвращается ссылка на компонент, который получит фокус после того, как окно станет активным;
- ◆ `setFocusProxy(<QWidget>)` — позволяет указать ссылку на компонент, который будет получать фокус ввода вместо текущего компонента;
- ◆ `focusProxy ()` — возвращает ссылку на компонент, который обрабатывает фокус ввода вместо текущего компонента. Если такого компонента нет, метод возвращает значение `None`;
- ◆ `focusNextChild ()` — находит следующий компонент, которому можно передать фокус, и передает фокус ему. Фактически работает аналогично нажатию клавиши <Tab>. Возвращает значение `True`, если компонент найден, и `False` — в противном случае;

- ◆ `focusPreviousChild()` — находит предыдущий компонент, которому можно передать фокус, и передает фокус ему. Работает аналогично нажатию комбинации клавиш `<Shift>+<Tab>`. Возвращает значение `True`, если компонент найден, и `False` — в противном случае;
- ◆ `focusNextPrevChild(<Флаг>)` — если в параметре указано значение `True`, работает аналогично методу `focusNextChild()`, если указано `False` — аналогично методу `focusPreviousChild()`. Возвращает значение `True`, если компонент найден, и `False` — в противном случае;
- ◆ `setTabOrder(<Компонент1>, <Компонент2>)` — позволяет задать последовательность смены фокуса при нажатии клавиши `<Tab>`. Метод является статическим. В параметре `<Компонент2>` указывается ссылка на компонент, на который переместится фокус с компонента `<Компонент1>`. Если компонентов много, метод вызывается несколько раз. Вот пример указания цепочки перехода `widget1 -> widget2 -> widget3 -> widget4`:

```
QtWidgets.QWidget.setTabOrder(widget1, widget2)
QtWidgets.QWidget.setTabOrder(widget2, widget3)
QtWidgets.QWidget.setTabOrder(widget3, widget4)
```

- ◆ `setFocusPolicy(<Способ>)` — задает способ получения фокуса компонентом в виде одного из следующих атрибутов класса `QtCore.Qt`:
 - `NoFocus` — 0 — компонент не может получать фокус;
 - `TabFocus` — 1 — получает фокус с помощью клавиши `<Tab>`;
 - `ClickFocus` — 2 — получает фокус с помощью щелчка мышью;
 - `StrongFocus` — 11 — получает фокус с помощью клавиши `<Tab>` и щелчка мышью;
 - `WheelFocus` — 15 — получает фокус с помощью клавиши `<Tab>`, щелчка мышью и колесика мыши;
- ◆ `focusPolicy()` — возвращает текущий способ получения фокуса;
- ◆ `grabKeyboard()` — захватывает ввод с клавиатуры. Другие компоненты не будут получать события клавиатуры, пока не будет вызван метод `releaseKeyboard()`;
- ◆ `releaseKeyboard()` — освобождает захваченный ранее ввод с клавиатуры.

Получить ссылку на компонент, находящийся в фокусе ввода, позволяет статический метод `focusWidget()` класса `QApplication`. Если ни один компонент не имеет фокуса ввода, метод возвращает значение `None`. Не путайте этот метод с одноименным методом из класса `QWidget`.

Обработать получение и потерю фокуса ввода позволяют следующие методы класса `QWidget`:

- ◆ `focusInEvent(self, <event>)` — вызывается при получении фокуса ввода;
- ◆ `focusOutEvent(self, <event>)` — вызывается при потере фокуса ввода.

Через параметр `<event>` доступен экземпляр класса `QFocusEvent`, который поддерживает следующие методы:

- ◆ `gotFocus()` — возвращает значение `True`, если тип события `QEvent.FocusIn` (получение фокуса ввода), и `False` — в противном случае;
- ◆ `lostFocus()` — возвращает значение `True`, если тип события `QEvent.FocusOut` (потеря фокуса ввода), и `False` — в противном случае;

- ◆ `reason()` — возвращает причину установки фокуса. Значение аналогично значению параметра `<Причина>` в методе `setFocus()`.

Создадим окно с кнопкой и двумя однострочными полями ввода (листинг 19.12). Для полей ввода обработаем получение и потерю фокуса ввода, а по нажатию кнопки установим фокус ввода на второе поле. Кроме того, зададим последовательность перехода при нажатии клавиши `<Tab>`.

Листинг 19.12. Установка фокуса ввода

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets

class MyLineEdit(QtWidgets.QLineEdit):
    def __init__(self, id, parent=None):
        QtWidgets.QLineEdit.__init__(self, parent)
        self.id = id
    def focusInEvent(self, e):
        print("Получен фокус полем", self.id)
        QtWidgets.QLineEdit.focusInEvent(self, e)
    def focusOutEvent(self, e):
        print("Потерян фокус полем", self.id)
        QtWidgets.QLineEdit.focusOutEvent(self, e)

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
        self.button = QtWidgets.QPushButton("Установить фокус на поле 2")
        self.line1 = MyLineEdit(1)
        self.line2 = MyLineEdit(2)
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.button)
        self.vbox.addWidget(self.line1)
        self.vbox.addWidget(self.line2)
        self.setLayout(self.vbox)
        self.button.clicked.connect(self.on_clicked)
        # Задаем порядок обхода с помощью клавиши <Tab>
        QtWidgets.QWidget.setTabOrder(self.line1, self.line2)
        QtWidgets.QWidget.setTabOrder(self.line2, self.button)
    def on_clicked(self):
        self.line2.setFocus()

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

19.8.2. Назначение клавиш быстрого доступа

Клавиши быстрого доступа (иногда их также называют «горячими» клавишами) позволяют установить фокус ввода с помощью нажатия специальной (например, `<Alt>` или `<Ctrl>`) и какой-либо дополнительной клавиши. Если после нажатия клавиш быстрого доступа в фокусе окажется кнопка (или пункт меню), она будет нажата.

Чтобы задать клавиши быстрого доступа, следует в тексте надписи указать символ `&` перед буквой. В этом случае буква, перед которой указан символ `&`, будет — в качестве подсказки пользователю — подчеркнута. При одновременном нажатии клавиши `<Alt>` и подчеркнутой буквы компонент окажется в фокусе ввода. Некоторые компоненты, например текстовое поле, не имеют надписи. Чтобы задать клавиши быстрого доступа для таких компонентов, необходимо отдельно создать надпись и связать ее с компонентом с помощью метода `setBuddy(<Компонент>)` класса `QLabel`. Если же создание надписи не представляется возможным, можно воспользоваться следующими методами класса `QWidget`:

- ◆ `grabShortcut(<Клавиши>[, <Контекст>])` — регистрирует клавиши быстрого доступа и возвращает идентификатор, с помощью которого можно управлять ими в дальнейшем. В параметре `<Клавиши>` указывается экземпляр класса `QKeySequence` из модуля `QtGui`. Создать экземпляр этого класса для комбинации клавиш `<Alt>+<E>` можно, например, так:

```
QtGui.QKeySequence.mnemonic("&e")
QtGui.QKeySequence("Alt+e")
QtGui.QKeySequence(QtCore.Qt.ALT + QtCore.Qt.Key_E)
```

В параметре `<Контекст>` можно указать атрибуты `WidgetShortcut`, `WidgetWithChildrenShortcut`, `WindowShortcut` (значение по умолчанию) и `ApplicationShortcut` класса `QtCore.Qt`;

- ◆ `releaseShortcut(<ID>)` — удаляет комбинацию с идентификатором `<ID>`;
- ◆ `setShortcutEnabled(<ID>[, <Флаг>])` — если в качестве параметра `<Флаг>` указано `True` (значение по умолчанию), клавиша быстрого доступа с идентификатором `<ID>` разрешена. Значение `False` запрещает использование клавиши быстрого доступа.

При нажатии клавиш быстрого доступа генерируется событие `QEEvent.Shortcut`, которое можно обработать в методе `event(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QShortcutEvent`, поддерживающий следующие методы:

- ◆ `shortcutId()` — возвращает идентификатор комбинации клавиш;
- ◆ `isAmbiguous()` — возвращает значение `True`, если событие отправлено сразу нескольким компонентам, и `False` — в противном случае;
- ◆ `key()` — возвращает экземпляр класса `QKeySequence`, представляющий нажатую клавишу быстрого доступа.

Создадим окно с надписью, двумя однострочными текстовыми полями и кнопкой (листинг 19.13). Для первого текстового поля назначим комбинацию клавиш `<Alt>+` через надпись, а для второго поля — комбинацию `<Alt>+<E>` с помощью метода `grabShortcut()`. Для кнопки назначим комбинацию клавиш `<Alt>+<Y>` обычным образом — через надпись на кнопке.

Листинг 19.13. Назначение клавиш быстрого доступа разными способами

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtGui, QtWidgets
```

```

class MyLineEdit(QtWidgets.QLineEdit):
    def __init__(self, parent=None):
        QtWidgets.QLineEdit.__init__(self, parent)
        self.id = None
    def event(self, e):
        if e.type() == QtCore.QEvent.Shortcut:
            if self.id == e.shortcutId():
                self.setFocus(QtCore.Qt.ShortcutFocusReason)
                return True
        return QtWidgets.QLineEdit.event(self, e)

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(300, 100)
        self.label = QtWidgets.QLabel("Установить фокус на поле 1")
        self.lineEdit1 = QtWidgets.QLineEdit()
        self.label.setBuddy(self.lineEdit1)
        self.lineEdit2 = MyLineEdit()
        self.lineEdit2.id = self.lineEdit2.grabShortcut(
            QtGui.QKeySequence.mnemonic("&e"))
        self.button = QtWidgets.QPushButton("&Убрать фокус с поля 1")
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.lineEdit1)
        self.vbox.addWidget(self.lineEdit2)
        self.vbox.addWidget(self.button)
        self.setLayout(self.vbox)
        self.button.clicked.connect(self.on_clicked)
    def on_clicked(self):
        self.lineEdit1.clearFocus()

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())

```

Помимо рассмотренных способов, для назначения клавиш быстрого доступа можно воспользоваться классом `QShortcut` из модуля `QtWidgets`. В этом случае назначение клавиш для второго текстового поля будет выглядеть так:

```

self.lineEdit2 = QtWidgets.QLineEdit()
self.shc = QtWidgets.QShortcut(QtGui.QKeySequence.mnemonic("&e"), self)
self.shc.setContext(QtCore.Qt.WindowShortcut)
self.shc.activated.connect(self.lineEdit2.setFocus)

```

Назначить комбинацию быстрых клавиш также позволяет класс `QAction` из модуля `QtWidgets`. Назначение клавиш для второго текстового поля выглядит следующим образом:

```
self.lineEdit2 = QtWidgets.QLineEdit()
self.act = QtWidgets.QAction(self)
self.act.setShortcut(QtGui.QKeySequence.mnemonic("&e"))
self.act.triggered.connect(self.lineEdit2.setFocus)
self.addAction(self.act)
```

19.8.3. Нажатие и отпускание клавиши на клавиатуре

При нажатии и отпускании клавиши вызываются следующие методы:

- ◆ `keyPressEvent(self, <event>)` — вызывается при нажатии клавиши на клавиатуре. Если клавишу удерживать нажатой, метод будет вызываться многократно, пока клавишу не отпустят;
- ◆ `keyReleaseEvent(self, <event>)` — вызывается при отпускании нажатой ранее клавиши.

Через параметр `<event>` доступен экземпляр класса `QKeyEvent`, хранящий дополнительную информацию о событии. Он поддерживает следующие полезные для нас методы (полный их список приведен в документации по классу `QKeyEvent` на странице <https://doc.qt.io/qt-5/qkeyevent.html>):

- ◆ `key()` — возвращает код нажатой клавиши. Пример определения клавиши:

```
if e.key() == QtCore.Qt.Key_B:
    print("Нажата клавиша <B>")
```

- ◆ `text()` — возвращает текстовое представление введенного символа в кодировке Unicode или пустую строку, если была нажата специальная клавиша;
- ◆ `modifiers()` — позволяет определить, какие клавиши-модификаторы (`<Shift>`, `<Ctrl>`, `<Alt>` и др.) были нажаты вместе с клавишей. Может содержать значения следующих атрибутов из класса `QtCore.Qt` или их комбинацию:

- `NoModifier` — модификаторы не были нажаты;
- `ShiftModifier` — была нажата клавиша `<Shift>`;
- `ControlModifier` — была нажата клавиша `<Ctrl>`;
- `AltModifier` — была нажата клавиша `<Alt>`;
- `MetaModifier` — была нажата клавиша `<Meta>`;
- `KeypadModifier` — была нажата любая клавиша на дополнительной клавиатуре;
- `GroupSwitchModifier` — была нажата клавиша `<Mode_switch>` (только в X11).

Вот пример определения, была ли нажата клавиша-модификатор `<Shift>`:

```
if e.modifiers() & QtCore.Qt.ShiftModifier:
    print("Нажата клавиша-модификатор <Shift>")
```

- ◆ `isAutoRepeat()` — возвращает `True`, если событие было вызвано удержанием клавиши нажатой, и `False` — в противном случае;
- ◆ `matches(<QKeySequence.StandardKey>)` — возвращает значение `True`, если была нажата специальная комбинация клавиш, соответствующая указанному значению, и `False` — в противном случае. В качестве значения указываются атрибуты из класса `QKeySequence` — например, `QKeySequence.Copy` для комбинации клавиш `<Ctrl>+<C>` (Копировать):

```
if e.matches(QtGui.QKeySequence.Copy) :  
    print ("Нажата комбинация <Ctrl>+<C>")
```

Полный список атрибутов содержится в документации по классу QKeySequence (см. <https://doc.qt.io/qt-5/qkeysequence.html#StandardKey-enum>).

При обработке нажатия клавиш следует учитывать, что:

- ◆ компонент должен иметь возможность принимать фокус ввода. Некоторые компоненты по умолчанию не могут принимать фокус ввода — например, надпись. Чтобы изменить способ получения фокуса, следует воспользоваться методом `setFocusPolicy(<Способ>)`, который мы рассматривали в разд. 19.8.1;
- ◆ чтобы захватить эксклюзивный ввод с клавиатуры, следует воспользоваться методом `grabKeyboard()`, а чтобы освободить ввод — методом `releaseKeyboard()`;
- ◆ можно перехватить нажатие любых клавиш, кроме клавиши `<Tab>` и комбинации `<Shift>+<Tab>`. Эти клавиши используются для передачи фокуса следующему и предыдущему компоненту соответственно. Перехватить нажатие этих клавиш можно только в методе `event(self, <event>)`;
- ◆ если событие обработано, следует вызвать метод `accept()` объекта события. Чтобы родительский компонент смог получить событие, вместо метода `accept()` необходимо вызвать метод `ignore()`.

19.9. События мыши

События мыши обрабатываются не реже, чем события клавиатуры. С помощью специальных методов можно обработать нажатие и отпускание кнопки мыши, перемещение указателя, входжение указателя в область компонента и выхода из этой области. В зависимости от ситуации можно изменить вид указателя — например, при выполнении длительной операции отобразить указатель в виде песочных часов. В этом разделе мы рассмотрим изменение вида указателя мыши как для отдельного компонента, так и для всего приложения.

19.9.1. Нажатие и отпускание кнопки мыши

При нажатии и отпускании кнопки мыши вызываются следующие методы:

- ◆ `mousePressEvent(self, <event>)` — вызывается при нажатии кнопки мыши;
- ◆ `mouseReleaseEvent(self, <event>)` — вызывается при отпускании ранее нажатой кнопки мыши;
- ◆ `mouseDoubleClickEvent(self, <event>)` — вызывается при двойном щелчке мышью в области компонента. Следует учитывать, что двойному щелчку предшествуют другие события. Последовательность событий при двойном щелчке выглядит так:

```
MouseButtonPress  
MouseButtonRelease  
MouseButtonDblClick  
MouseButtonPress  
MouseButtonRelease
```

Задать интервал двойного щелчка позволяет метод `setDoubleClickInterval()` класса `QApplication`, а получить его текущее значение — метод `doubleClickInterval()` того же класса.

Через параметр `<event>` доступен экземпляр класса `QMouseEvent`, хранящий дополнительную информацию о событии. Он поддерживает такие методы:

- ◆ `x()` и `y()` — возвращают координаты по осям X и Y соответственно в пределах области компонента;
- ◆ `pos()` — возвращает экземпляр класса `QPoint` с целочисленными координатами в пределах области компонента;
- ◆ `localPos()` — возвращает экземпляр класса `QPointF` с вещественными координатами в пределах области компонента;
- ◆ `globalX()` и `globalY()` — возвращают координаты по осям X и Y соответственно в пределах экрана;
- ◆ `globalPos()` — возвращает экземпляр класса `QPoint` с координатами в пределах экрана;
- ◆ `windowPos()` — возвращает экземпляр класса `QPointF` с вещественными координатами в пределах окна;
- ◆ `screenPos()` — возвращает экземпляр класса `QPointF` с вещественными координатами в пределах экрана;
- ◆ `button()` — позволяет определить, щелчок какой кнопкой мыши вызвал событие. Возвращает значение одного из следующих атрибутов класса `QtCore.Qt` (здесь указаны не все атрибуты, полный их список приведен на странице <https://doc.qt.io/qt-5/qt.html#MouseButton-enum>):
 - `NoButton` — 0 — кнопки не нажаты. Это значение возвращается методом `button()` при перемещении указателя мыши;
 - `LeftButton` — 1 — нажата левая кнопка мыши;
 - `RightButton` — 2 — нажата правая кнопка мыши;
 - `MidButton` и `MiddleButton` — 4 — нажата средняя кнопка мыши;
 - `XButton1`, `ExtraButton1` и `BackButton` — 8 — нажата первая из дополнительных кнопок мыши;
 - `XButton2`, `ExtraButton2` и `ForwardButton` — 16 — нажата вторая из дополнительных кнопок мыши;
- ◆ `buttons()` — позволяет определить все кнопки, которые нажаты одновременно. Возвращает комбинацию упомянутых ранее атрибутов. Вот пример определения нажатой кнопки мыши:

```
if e.buttons() & QtCore.Qt.LeftButton:  
    print("Нажата левая кнопка мыши")
```
- ◆ `modifiers()` — позволяет определить, какие клавиши-модификаторы (`<Shift>`, `<Ctrl>`, `<Alt>` и др.) были нажаты вместе с кнопкой мыши. Возможные значения мы уже рассматривали в разд. 19.8.3;
- ◆ `timestamp()` — возвращает в виде числа отметку системного времени, в которое возникло событие.

Если событие было успешно обработано, следует вызывать метод `accept()` объекта события. Чтобы родительский компонент мог получить событие, вместо метода `accept()` нужно вызвать метод `ignore()`.

Если у компонента атрибут `WA_NoMousePropagation` класса `QtCore.Qt` установлен в `True`, событие мыши не будет передаваться родительскому компоненту. Значение атрибута можно изменить с помощью метода `setAttribute()`, вызванного у этого компонента:

```
button.setAttribute(QtCore.Qt.WA_NoMousePropagation, True)
```

По умолчанию событие мыши перехватывает компонент, над которым был произведен щелчок мышью. Чтобы перехватывать нажатие и отпускание мыши вне компонента, следует захватить мышь вызовом метода `grabMouse()`. Освободить захваченную ранее мышь позволяет метод `releaseMouse()`.

19.9.2. Перемещение указателя мыши

Чтобы обработать перемещение указателя мыши, необходимо переопределить метод `mouseMoveEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QMouseEvent`, содержащий дополнительную информацию о событии. Методы этого класса мы уже рассматривали в предыдущем разделе. Следует учитывать, что метод `button()` при перемещении мыши возвращает значение `QtCore.Qt.NoButton`.

По умолчанию метод `mouseMoveEvent()` вызывается только в том случае, если при перемещении удерживается нажатой какая-либо кнопка мыши. Это сделано специально, чтобы не создавать лишних событий при обычном перемещении указателя мыши. Если необходимо обрабатывать любые перемещения указателя в пределах компонента, следует вызвать у этого компонента метод `setMouseTracking()`, которому передать значение `True`. Чтобы обработать все перемещения внутри окна, нужно дополнительно захватить мышь вызовом метода `grabMouse()`.

Метод `pos()` объекта события возвращает позицию точки в системе координат текущего компонента. Чтобы преобразовать координаты точки в систему координат родительского компонента или в глобальную систему координат, нужно воспользоваться следующими методами класса `QWidget`:

- ◆ `mapToGlobal(<QPoint>)` — преобразует координаты точки из системы координат компонента в глобальную систему координат. Возвращает экземпляр класса `QPoint`;
- ◆ `mapFromGlobal(<QPoint>)` — преобразует координаты точки из глобальной в систему координат компонента. Возвращает экземпляр класса `QPoint`;
- ◆ `mapToParent(<QPoint>)` — преобразует координаты точки из системы координат компонента в систему координат родительского компонента. Если компонент не имеет родителя, действует как метод `mapToGlobal()`. Возвращает экземпляр класса `QPoint`;
- ◆ `mapFromParent(<QPoint>)` — преобразует координаты точки из системы координат родительского компонента в систему координат текущего компонента. Если компонент не имеет родителя, работает подобно методу `mapFromGlobal()`. Возвращает экземпляр класса `QPoint`;
- ◆ `mapTo(<QWidget>, <QPoint>)` — преобразует координаты точки из системы координат текущего компонента в систему координат родительского компонента `<QWidget>`. Возвращает экземпляр класса `QPoint`;
- ◆ `mapFrom(<QWidget>, <QPoint>)` — преобразует координаты точки из системы координат родительского компонента `<QWidget>` в систему координат текущего компонента. Возвращает экземпляр класса `QPoint`.

19.9.3. Наведение и увод указателя

Обработать наведение указателя мыши на компонент и увод его с компонента позволяют следующие методы:

- ◆ `enterEvent(self, <event>)` — вызывается при наведении указателя мыши на область компонента;
- ◆ `leaveEvent(self, <event>)` — вызывается, когда указатель мыши покидает область компонента.

Через параметр `<event>` доступен экземпляр класса `QEvent`, не несущий никакой дополнительной информации. Вполне достаточно знать, что указатель попал в область компонента или покинул ее.

19.9.4. Прокрутка колесика мыши

Все современные мыши комплектуются колесиком, обычно используемым для прокрутки содержимого компонента. Обработать поворот колесика позволяет метод `wheelEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QWheelEvent`, который позволяет получить дополнительную информацию о событии.

Класс `QWheelEvent` поддерживает методы:

- ◆ `angleDelta()` — возвращает угол поворота колесика в градусах, умноженный на 8, в виде экземпляра класса `QPoint`. Это значение может быть положительным или отрицательным — в зависимости от направления поворота. Вот пример определения угла поворота колесика:

```
angle = e.angleDelta() / 8
angleX = angle.x()
angleY = angle.y()
```

- ◆ `pixelDelta()` — возвращает величину поворота колесика в пикселях в виде экземпляра класса `QPoint`. Это значение может быть положительным или отрицательным — в зависимости от направления поворота;
- ◆ `x()` и `y()` — возвращают координаты указателя в момент события по осям X и Y соответственно в пределах области компонента;
- ◆ `pos()` — возвращает экземпляр класса `QPoint` с целочисленными координатами указателя в момент события в пределах области компонента;
- ◆ `posF()` — возвращает экземпляр класса `QPointF` с вещественными координатами указателя в момент события в пределах области компонента;
- ◆ `globalX()` и `globalY()` — возвращают координаты указателя в момент события по осям X и Y соответственно в пределах экрана;
- ◆ `globalPos()` — возвращает экземпляр класса `QPoint` с целочисленными координатами указателя в момент события в пределах экрана;
- ◆ `globalPosF()` — возвращает экземпляр класса `QPointF` с вещественными координатами указателя в момент события в пределах экрана;
- ◆ `buttons()` — позволяет определить кнопки, которые нажаты одновременно с поворотом колесика. Возвращает комбинацию значений атрибутов, указанных в описании метода `buttons()` (см. разд. 19.9.1). Вот пример определения нажатой кнопки мыши:

```
if e.buttons() & QtCore.Qt.LeftButton:
    print("Нажата левая кнопка мыши")
```

- ◆ `modifiers()` — позволяет определить, какие клавиши-модификаторы (`<Shift>`, `<Ctrl>`, `<Alt>` и др.) были нажаты одновременно с поворотом колесика. Возможные значения мы уже рассматривали в разд. 19.8.3;
- ◆ `timestamp()` — возвращает в виде числа отметку системного времени, в которое возникло событие.

Если событие было успешно обработано, необходимо вызвать метод `accept()` объекта события. Чтобы родительский компонент мог получить событие, вместо метода `accept()` необходимо вызвать метод `ignore()`.

19.9.5. Изменение внешнего вида указателя мыши

Для изменения внешнего вида указателя мыши при вхождении его в область компонента предназначены следующие методы класса `QWidget`:

- ◆ `setCursor(<Курсор>)` — задает внешний вид указателя мыши для компонента. В качестве параметра указывается экземпляр класса `QCursor` или следующие атрибуты из класса `QtCore.Qt`: `ArrowCursor` (стандартная стрелка), `UpArrowCursor` (стрелка, направленная вверх), `CrossCursor` (крестообразный указатель), `WaitCursor` (песочные часы), `IBeamCursor` (I-образный указатель), `SizeVerCursor` (стрелки, направленные вверх и вниз), `SizeHorCursor` (стрелки, направленные влево и вправо), `SizeBDiagCursor` (стрелки, направленные в правый верхний угол и левый нижний угол), `SizeFDiagCursor` (стрелки, направленные в левый верхний угол и правый нижний угол), `SizeAllCursor` (стрелки, направленные вверх, вниз, влево и вправо), `SplitVCursor` (указатель изменения высоты), `SplitHCursor` (указатель изменения ширины), `PointingHandCursor` (указатель в виде руки), `ForbiddenCursor` (перечеркнутый круг), `OpenHandCursor` (разжатая рука), `ClosedHandCursor` (сжатая рука), `WhatsThisCursor` (стрелка с вопросительным знаком) и `BusyCursor` (стрелка с песочными часами):

```
self.setCursor(QtCore.Qt.WaitCursor)
```

- ◆ `unsetCursor()` — отменяет установку указателя для компонента. В результате внешний вид указателя мыши будет наследоваться от родительского компонента;
- ◆ `cursor()` — возвращает экземпляр класса `QCursor`, представляющий текущий указатель.

Управлять видом указателя для всего приложения сразу можно с помощью следующих статических методов из класса `QApplication`:

- ◆ `setOverrideCursor(<Курсор>)` — задает внешний вид указателя мыши для всего приложения. В качестве параметра указывается экземпляр класса `QCursor` или один из ранее упомянутых специальных атрибутов класса `QtCore.Qt`. Для отмены установки необходимо вызвать метод `restoreOverrideCursor()`;

- ◆ `restoreOverrideCursor()` — отменяет изменение внешнего вида указателя мыши для всего приложения:

```
QtWidgets.QApplication.setOverrideCursor(QtCore.Qt.WaitCursor)
# Выполняем длительную операцию
QtWidgets.QApplication.restoreOverrideCursor()
```

- ◆ `changeOverrideCursor(<Курсор>)` — изменяет внешний вид указателя мыши для всего приложения. Если до вызова этого метода не вызывался метод `setOverrideCursor()`,

значение будет проигнорировано. В качестве параметра указывается экземпляр класса `QCursor` или один из специальных атрибутов класса `QtCore.Qt`;

- ◆ `overrideCursor()` — возвращает экземпляр класса `QCursor`, представляющий текущий указатель, или значение `None`, если таковой не был изменен.

Изменять внешний вид указателя мыши для всего приложения принято на небольшой промежуток времени — обычно на время выполнения какой-либо операции, в процессе которой приложение не может нормально реагировать на действия пользователя. Чтобы информировать об этом пользователя, указатель принятого выводить в виде песочных часов (атрибут `WaitCursor`) или стрелки с песочными часами (атрибут `BusyCursor`).

Метод `setOverrideCursor()` может быть вызван несколько раз. В этом случае курсоры помещаются в стек. Каждый вызов метода `restoreOverrideCursor()` удаляет последний курсор, добавленный в стек. Для нормальной работы приложения необходимо вызывать методы `setOverrideCursor()` и `restoreOverrideCursor()` одинаковое количество раз.

Класс `QCursor` позволяет создать объект курсора с изображением любой формы. Чтобы загрузить изображение, следует передать конструктору класса `QPixmap` путь к файлу изображения. Для создания объекта курсора необходимо передать конструктору класса `QCursor` в первом параметре экземпляр класса `QPixmap`, а во втором и третьем параметрах — координаты «горячей» точки будущего курсора. Вот пример создания и установки пользовательского курсора:

```
self.setCursor(QtGui.QCursor(QtGui.QPixmap("cursor.png"), 0, 0))
```

Класс `QCursor` также поддерживает два статических метода:

- ◆ `pos()` — возвращает экземпляр класса `QPoint` с координатами указателя мыши относительно экрана:

```
p = QtGui.QCursor.pos()  
print(p.x(), p.y())
```

- ◆ `setPos()` — позволяет задать позицию указателя мыши. Метод имеет два формата: `setPos(<X>, <Y>)` и `setPos(<QPoint>)`.

19.10. Технология drag & drop

Технология `drag & drop` позволяет обмениваться данными различных типов между компонентами как одного приложения, так и разных приложений, путем перетаскивания и сбрасывания объектов с помощью мыши. Типичным примером использования технологии служит перемещение файлов в Проводнике Windows. Чтобы переместить файл в другой каталог, достаточно нажать левую кнопку мыши над значком файла и, не отпуская кнопки, перетащить файл на значок каталога, а затем отпустить кнопку мыши. Если необходимо скопировать файл, следует дополнительно удерживать нажатой клавишу `<Ctrl>`.

19.10.1. Запуск перетаскивания

Операция перетаскивания состоит из двух частей: первая часть запускает процесс, а вторая обрабатывает момент сброса объекта. Обе части могут обрабатываться как одним, так и двумя разными приложениями. Запуск перетаскивания осуществляется следующим образом:

1. Внутри метода `mousePressEvent()` запоминаются координаты указателя мыши в момент щелчка ее левой кнопкой.

2. Внутри метода `mouseMoveEvent()` вычисляется пройденное расстояние или измеряется время операции. Это необходимо для того, чтобы предотвратить случайное перетаскивание. Управлять задержкой позволяют следующие статические методы класса `QApplication`:

- `setStartDragDistance(<Дистанция>)` — задает минимальное расстояние, после прохождения которого будет запущена операция перетаскивания;
- `startDragDistance()` — возвращает это расстояние;
- `setStartDragTime(<Время>)` — задает время задержки в миллисекундах перед запуском операции перетаскивания;
- `startDragTime()` — возвращает это время.

3. Если пройдено минимальное расстояние или истек минимальный промежуток времени, создается экземпляр класса `QDrag`, и у него вызывается метод `exec()`, который после завершения операции возвращает действие, выполненное с данными (например, их копирование или перемещение).

Создать экземпляр класса `QDrag` можно так:

```
<Объект> = QtGui.QDrag(<Ссылка на компонент>)
```

Класс `QDrag` поддерживает следующие методы:

◆ `exec()` — запускает процесс перетаскивания и возвращает действие, которое было выполнено по завершении операции. Метод имеет два формата:

- ```
exec([<Действия>=MoveAction])
exec(<Действия>, <Действие по умолчанию>)
```

В параметре `<Действия>` указывается комбинация допустимых действий, а в параметре `<Действие по умолчанию>` — действие, которое осуществляется, если в процессе выполнения операции не были нажаты клавиши-модификаторы. Возможные действия могут быть заданы следующими атрибутами класса `QtCore.Qt`: `CopyAction` (1, копирование), `MoveAction` (2, перемещение), `LinkAction` (4, создание ссылки), `IgnoreAction` (0, действие игнорировано), `TargetMoveAction` (32770):

```
act = drag.exec(QtCore.Qt.MoveAction | QtCore.Qt.CopyAction,
 QtCore.Qt.MoveAction)
```

Вместо метода `exec()` можно использовать аналогичный метод `exec_()`, сохраненный в PyQt 5 для совместимости с кодом, написанным под библиотеку PyQt 4;

◆ `setMimeData(<QMimeType>)` — позволяет задать перемещаемые данные. В качестве значения указывается экземпляр класса `QMimeType`. Вот пример передачи текста:

```
data = QtCore.QMimeData()
data.setText("Перетаскиваемый текст")
drag = QtGui.QDrag(self)
drag.setMimeData(data)
```

- ◆ `mimeData()` — возвращает экземпляр класса `QMimeType` с перемещаемыми данными;
  - ◆ `setPixmap(<QPixmap>)` — задает изображение, которое будет перемещаться вместе с указателем мыши. В качестве параметра указывается экземпляр класса `QPixmap`:
- ```
drag.setPixmap(QtGui.QPixmap("dd_representer.png"))
```
- ◆ `pixmap()` — возвращает экземпляр класса `QPixmap` с изображением, которое перемещается вместе с указателем;

- ◆ `setHotSpot(<QPoint>)` — задает координаты «горячей» точки на перемещаемом изображении. В качестве параметра указывается экземпляр класса `QPoint`:

```
drag.setHotSpot(QtCore.QPoint(20, 20))
```
- ◆ `hotSpot()` — возвращает экземпляр класса `QPoint` с координатами «горячей» точки на перемещаемом изображении;
- ◆ `setDragCursor(<QPixmap>, <Действие>)` — позволяет изменить внешний вид указателя мыши для действия, указанного во втором параметре. Первым параметром передается экземпляр класса `QPixmap`, который, собственно, станет указателем мыши. Если в первом параметре указан пустой объект класса `QPixmap`, ранее установленный указатель для действия будет отменен. Вот пример изменения указателя для перемещения:

```
drag.setDragCursor(QtGui.QPixmap("move_cursor.png"),  
                   QtCore.Qt.MoveAction)
```
- ◆ `dragCursor(<Действие>)` — возвращает экземпляр класса `QPixmap`, представляющий указатель мыши для заданного действия;
- ◆ `source()` — возвращает ссылку на компонент-источник;
- ◆ `target()` — возвращает ссылку на компонент-приемник или значение `None`, если компонент находится в другом приложении;
- ◆ `supportedActions()` — возвращает значение, представляющее комбинацию допустимых в текущей операции действий. Возможные действия обозначаются упомянутыми ранее атрибутами класса `QtCore.Qt`;
- ◆ `defaultAction()` — возвращает действие по умолчанию в виде одного из перечисленных ранее атрибутов класса `QtCore.Qt`.

Класс `QDrag` поддерживает два сигнала:

- ◆ `actionChanged(<Действие>)` — генерируется при изменении действия. Новое действие представляется одним из упомянутых ранее атрибутов класса `QtCore.Qt`;
- ◆ `targetChanged(<Компонент>)` — генерируется при изменении принимающего компонента, который представляется экземпляром соответствующего класса.

Вот пример назначения обработчиков сигналов:

```
drag.actionChanged.connect(self.on_action_changed)  
drag.targetChanged.connect(self.on_target_changed)
```

19.10.2. Класс `QMimeTypeData`

Перемещаемые данные и сведения о MIME-типе должны быть представлены экземпляром класса `QMimeTypeData`. Его следует передать в метод `setMimeData()` класса `QDrag`. Выражение, создающее экземпляр класса `QMimeTypeData`, выглядит так:

```
data = QtCore.QMimeTypeData()
```

Класс `QMimeTypeData` поддерживает следующие полезные для нас методы (полный их список приведен на странице <https://doc.qt.io/qt-5/qmimedata.html>):

- ◆ `setText(<Текст>)` — устанавливает текстовые данные (MIME-тип `text/plain`):

```
data.setText("Перетаскиваемый текст")
```
- ◆ `text()` — возвращает текстовые данные;

- ◆ `hasText()` — возвращает значение `True`, если объект содержит текстовые данные, и `False` — в противном случае;
- ◆ `setHtml(<HTML-текст>)` — устанавливает текстовые данные в формате HTML (MIME-тип `text/html`):

```
data.setHtml("<b>Перетаскиваемый HTML-текст</b>")
```
- ◆ `html()` — возвращает текстовые данные в формате HTML;
- ◆ `hasHtml()` — возвращает значение `True`, если объект содержит текстовые данные в формате HTML, и `False` — в противном случае;
- ◆ `setUrls(<Список URI-адресов>)` — устанавливает список URI-адресов (MIME-тип `text/uri-list`). В качестве значения указывается список с экземплярами класса `QUrl`. С помощью этого MIME-типа можно обработать перетаскивание файлов:

```
data.setUrls([QtCore.QUrl("https://www.google.ru/")])
```
- ◆ `urls()` — возвращает список URI-адресов:

```
uri = e.mimeData().urls()[0].toString()
```
- ◆ `hasUrls()` — возвращает значение `True`, если объект содержит список URI-адресов, и `False` — в противном случае;
- ◆ `setImageData(<Объект изображения>)` — устанавливает изображение (MIME-тип `application/x-qt-image`). В качестве значения можно указать, например, экземпляр класса `QImage` или `QPixmap`:

```
data.setImageData(QtGui.QImage("pixmap.png"))
data.setImageData(QtGui.QPixmap("pixmap.png"))
```
- ◆ `imageData()` — возвращает объект изображения (тип возвращаемого объекта зависит от типа объекта, указанного в методе `setImageData()`);
- ◆ `hasImage()` — возвращает значение `True`, если объект содержит изображение, и `False` — в противном случае;
- ◆ `setData(<MIME-тип>, <Данные>)` — позволяет установить данные произвольного MIME-типа. В первом параметре указывается MIME-тип в виде строки, а во втором параметре — экземпляр класса `QByteArray` с данными. Метод можно вызвать несколько раз с различными MIME-тиปами. Вот пример передачи текстовых данных:

```
data.setData("text/plain",
            QtCore.QByteArray(bytes("Данные", "utf-8")))
data.setData("text/html", QtCore.QByteArray(bytes("Данные", "utf-8")))
```
- ◆ `data(<MIME-тип>)` — возвращает экземпляр класса `QByteArray` с данными, соответствующими указанному MIME-типу;
- ◆ `hasFormat(<MIME-тип>)` — возвращает значение `True`, если объект содержит данные указанного MIME-типа, и `False` — в противном случае;
- ◆ `formats()` — возвращает список с поддерживаемыми объектом MIME-типами;
- ◆ `removeFormat(<MIME-тип>)` — удаляет данные, соответствующие указанному MIME-типу;
- ◆ `clear()` — удаляет все данные.

Если необходимо перетаскивать данные какого-либо специфического типа, нужно наследовать класс `QMimeType` и переопределить в нем методы `retrieveData()` и `formats()`. За подробной информацией по этому вопросу обращайтесь к документации.

19.10.3. Обработка сброса

Прежде чем обрабатывать перетаскивание и сбрасывание объекта, необходимо сообщить системе, что компонент может обрабатывать эти события. Для этого внутри конструктора компонента следует вызвать метод `setAcceptDrops()`, унаследованный от класса `QWidget`, и передать этому методу `True`:

```
self.setAcceptDrops(True)
```

Обработка перетаскивания и сброса объекта выполняется следующим образом:

1. Внутри метода `dragEnterEvent()` компонента проверяется MIME-тип перетаскиваемых данных и действие. Если компонент способен обработать сброс этих данных и соглашается с предложенным действием, необходимо вызвать метод `acceptProposedAction()` объекта события. Если нужно изменить действие, методу `setDropAction()` объекта события передается новое действие, а затем у того же объекта вызывается метод `accept()` вместо `acceptProposedAction()`.
2. Если необходимо ограничить область сброса некоторым участком компонента, следует дополнительно определить в нем метод `dragMoveEvent()`. Этот метод будет постоянно вызываться при перетаскивании внутри области компонента. При достижении указателем мыши нужного участка компонента следует вызвать метод `accept()` и передать ему экземпляр класса `QRect` с координатами и размером этого участка. В этом случае при перетаскивании внутри участка метод `dragMoveEvent()` повторно вызываться не будет.
3. Внутри метода `dropEvent()` компонента производится обработка сброса.

Обработать события, возникающие в процессе перетаскивания и сбрасывания, позволяют следующие методы класса `QWidget`:

- ◆ `dragEnterEvent(self, <event>)` — вызывается, когда перетаскиваемый объект входит в область компонента. Через параметр `<event>` доступен экземпляр класса `QDragEnterEvent`;
- ◆ `dragLeaveEvent(self, <event>)` — вызывается, когда перетаскиваемый объект покидает область компонента. Через параметр `<event>` доступен экземпляр класса `QDragLeaveEvent`;
- ◆ `dragMoveEvent(self, <event>)` — вызывается при перетаскивании объекта внутри области компонента. Через параметр `<event>` доступен экземпляр класса `QDragMoveEvent`;
- ◆ `dropEvent(self, <event>)` — вызывается при сбрасывании объекта в области компонента. Через параметр `<event>` доступен экземпляр класса `QDropEvent`.

Класс `QDragLeaveEvent` наследует класс `QEvent` и не несет никакой дополнительной информации. Достаточно просто знать, что перетаскиваемый объект покинул область компонента.

Цепочка наследования остальных классов выглядит так:

`QEvent` — `QDropEvent` — `QDragMoveEvent` — `QDragEnterEvent`

Класс `QDragEnterEvent` не содержит собственных методов, но наследует все методы классов `QDropEvent` и `QDragMoveEvent`.

Класс `QDropEvent` поддерживает следующие методы:

- ◆ `mimeData()` — возвращает экземпляр класса `QMimeData` с перемещаемыми данными и информацией о MIME-типе;
- ◆ `pos()` — возвращает экземпляр класса `QPoint` с целочисленными координатами сбрасывания объекта;

- ◆ `posF()` — возвращает экземпляр класса `QPointF` с вещественными координатами сбрасывания объекта;
- ◆ `possibleActions()` — возвращает комбинацию возможных действий при сбрасывании. Вот пример определения значений:

```
if e.possibleActions() & QtCore.Qt.MoveAction:  
    print("MoveAction")  
if e.possibleActions() & QtCore.Qt.CopyAction:  
    print("CopyAction")
```
- ◆ `proposedAction()` — возвращает действие по умолчанию при сбрасывании;
- ◆ `acceptProposedAction()` — сообщает о готовности принять переносимые данные и согласия с действием, возвращаемым методом `proposedAction()`. Метод `acceptProposedAction()` (или метод `accept()`, поддерживаемый классом `QDragMoveEvent`) необходимо вызвать внутри метода `dragEnterEvent()`, иначе метод `dropEvent()` вызван не будет;
- ◆ `setDropAction(<Действие>)` — позволяет указать другое действие при сбрасывании. После изменения действия следует вызвать метод `accept()`, а не `acceptProposedAction()`;
- ◆ `dropAction()` — возвращает действие, которое должно быть выполнено при сбрасывании. Оно может не совпадать со значением, возвращаемым методом `proposedAction()`, если действие было изменено с помощью метода `setDropAction()`;
- ◆ `keyboardModifiers()` — позволяет определить, какие клавиши-модификаторы (`<Shift>`, `<Ctrl>`, `<Alt>` и др.) были нажаты вместе с кнопкой мыши. Возможные значения мы уже рассматривали в разд. 19.8.3;
- ◆ `mouseButtons()` — позволяет определить кнопки мыши, которые были нажаты в процессе переноса данных;
- ◆ `source()` — возвращает ссылку на компонент внутри приложения, являющийся источником события, или значение `None`, если данные переносятся из другого приложения.

Теперь рассмотрим методы класса `QDragMoveEvent`:

- ◆ `accept([<QRect>])` — сообщает о согласии с дальнейшей обработкой события. В качестве параметра можно указать экземпляр класса `QRect` с координатами и размерами прямоугольной области, в которой будет доступно сбрасывание;
- ◆ `ignore([<QRect>])` — отменяет операцию переноса данных. В качестве параметра можно указать экземпляр класса `QRect` с координатами и размерами прямоугольной области, в которой сбрасывание запрещено;
- ◆ `answerRect()` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, в которой произойдет сбрасывание, если событие будет принято.

Некоторые компоненты в PyQt уже поддерживают технологию `drag & drop` — так, в одностороннее текстовое поле можно перетащить текст из другого приложения. Поэтому, прежде чем изобретать свой «велосипед», убедитесь, что поддержка технологии в компоненте не реализована.

19.11. Работа с буфером обмена

Помимо технологии `drag & drop`, для обмена данными между приложениями используется буфер обмена — одно приложение помещает данные в буфер обмена, а второе приложение

(или то же самое) может их извлечь. Получить ссылку на глобальный объект буфера обмена позволяет статический метод `clipboard()` класса `QApplication`:

```
clipboard = QtWidgets.QApplication.clipboard()
```

Класс `QClipboard` поддерживает следующие методы:

- ◆ `setText(<Текст>)` — заносит текст в буфер обмена:
`clipboard.setText("Текст")`
- ◆ `text()` — возвращает из буфера обмена текст или пустую строку;
- ◆ `text(<Тип>)` — возвращает кортеж из двух строк: первая хранит текст из буфера обмена, вторая — название типа. В параметре `<Тип>` могут быть указаны значения "plain" (простой текст), "html" (HTML-код) или пустая строка (любой тип);
- ◆ `setImage(<QImage>)` — заносит в буфер обмена изображение, представленное экземпляром класса `QImage`:
`clipboard.setImage(QtGui.QImage("image.jpg"))`
- ◆ `image()` — возвращает из буфера обмена изображение, представленное экземпляром класса `QImage`, или пустой экземпляр этого класса;
- ◆ `setPixmap(<QPixmap>)` — заносит в буфер обмена изображение, представленное экземпляром класса `QPixmap`:
`clipboard.setPixmap(QtGui.QPixmap("image.jpg"))`
- ◆ `pixmap()` — возвращает из буфера обмена изображение, представленное экземпляром класса `QPixmap`, или пустой экземпляр этого класса;
- ◆ `setData(<QMimeData>)` — позволяет сохранить в буфере данные любого типа, представленные экземпляром класса `QMimeData` (см. разд. 19.10.2);
- ◆ `mimeData([<Режим>])` — возвращает данные, представленные экземпляром класса `QMimeData`;
- ◆ `clear()` — очищает буфер обмена.

Отследить изменение состояния буфера обмена позволяет сигнал `dataChanged`. Назначить обработчик этого сигнала можно так:

```
QtWidgets.qApp.clipboard().dataChanged.connect(on_change_clipboard)
```

19.12. Фильтрация событий

События можно перехватывать еще до того, как они будут переданы компоненту. Для этого необходимо создать класс, который является наследником класса `QObject`, и переопределить в нем метод `eventFilter(self, <Объект>, <event>)`. Через параметр `<Объект>` доступна ссылка на компонент, а через параметр `<event>` — на объект с дополнительной информацией о событии. Этот объект различен для разных типов событий — так, для события `MouseButtonPress` объект будет экземпляром класса `QMouseEvent`, а для события `KeyPress` — экземпляром класса `QKeyEvent`. Из метода `eventFilter()` следует вернуть значение `True`, если событие не должно быть передано дальше, и `False` — в противном случае. Вот пример такого класса-фильтра, перехватывающего нажатие клавиши ``:

```
class MyFilter(QtCore.QObject):
    def __init__(self, parent=None):
        QtCore.QObject.__init__(self, parent)
```

```
def eventFilter(self, obj, e):
    if e.type() == QtCore.QEvent.KeyPress:
        if e.key() == QtCore.Qt.Key_B:
            print("Событие от клавиши <B> не дойдет до компонента")
            return True
    return QtCore.QObject.eventFilter(self, obj, e)
```

Далее следует создать экземпляр этого класса, передав в конструктор ссылку на компонент, а затем вызвать у того же компонента метод `installEventFilter()`, передав в качестве единственного параметра ссылку на объект фильтра. Вот пример установки фильтра для надписи:

```
self.label.installEventFilter(MyFilter(self.label))
```

Метод `installEventFilter()` можно вызвать несколько раз, передавая ссылку на разные объекты фильтров. В этом случае первым будет вызван фильтр, который был добавлен последним. Кроме того, один фильтр можно установить сразу в нескольких компонентах. Ссылка на компонент, который является источником события, доступна через второй параметр метода `eventFilter()`.

Удалить фильтр позволяет метод `removeEventFilter(<фильтр>)`, вызываемый у компонента, для которого был назначен этот фильтр. Если таковой не был установлен, при вызове метода ничего не произойдет.

19.13. Искусственные события

Для создания искусственных событий применяются следующие статические методы из класса `QCoreApplication`:

- ◆ `sendEvent(<QObject>, <QEvent>)` — немедленно посыпает событие компоненту и возвращает результат выполнения обработчика;
- ◆ `postEvent(<QObject>, <QEvent>[, priority=NormalEventPriority])` — добавляет событие в очередь. Параметром `priority` можно передать приоритет события, использовав один из следующих атрибутов класса `QtCore.Qt`: `HighEventPriority` (1, высокий приоритет), `NormalEventPriority` (0, обычный приоритет — значение по умолчанию) и `LowEventPriority` (-1, низкий приоритет). Этот метод является потокобезопасным, следовательно, его можно использовать в многопоточных приложениях для обмена событиями между потоками.

В параметре `<QObject>` указывается ссылка на объект, которому посыпается событие, а в параметре `<QEvent>` — объект события. Последний может быть экземпляром как стандартного (например, `QMouseEvent`), так и пользовательского класса, являющегося наследником класса `QEvent`. Вот пример отправки события `QEvent.MouseButtonPress` компоненту `label`:

```
e = QtGui.QMouseEvent(QtCore.QEvent.MouseButtonPress,
                      QtCore.QPointF(5, 5), QtCore.Qt.LeftButton,
                      QtCore.Qt.LeftButton, QtCore.Qt.NoModifier)
QtCore.QCoreApplication.sendEvent(self.label, e)
```

Для отправки пользовательского события необходимо создать класс, наследующий `QEvent`. В этом классе следует зарегистрировать пользовательское событие с помощью статического метода `registerEventType()` и сохранить идентификатор события в атрибуте класса:

```
class MyEvent (QtCore.QEvent) :  
    idType = QtCore.QEvent.registerEventType()  
    def __init__(self, data):  
        QtCore.QEvent.__init__(self, MyEvent.idType)  
        self.data = data  
    def get_data(self):  
        return self.data
```

Вот пример отправки события класса MyEvent компоненту label:

```
QtCore.QCoreApplication.sendEvent(self.label, MyEvent("512"))
```

Обработать пользовательское событие можно с помощью методов `event(self, <event>)` или `customEvent(self, <event>)`:

```
def customEvent(self, e):  
    if e.type() == MyEvent.idType:  
        self.setText("Получены данные: {}".format(e.get_data()))
```