



ГЛАВА 17

Знакомство с PyQt 5

Итак, изучение основ языка Python закончено, и мы можем перейти к рассмотрению библиотеки PyQt, позволяющей разрабатывать приложения с графическим интерфейсом. Первые три главы второй части книги можно считать основными, поскольку в них описываются базовые возможности библиотеки и методы, которые наследуют все компоненты, так что материал этих глав нужно знать обязательно. Остальные главы содержат дополнительный справочный материал. В сопровождающий книгу электронный архив (см. *приложение*) включен файл PyQt.doc, который содержит более 750 дополнительных листингов, поясняющих материал второй части книги, — что позволило уменьшить ее объем, поскольку с этими листингами страниц книги было бы вдвое больше, чем всех страниц ее второй части.

17.1. Установка PyQt 5

Библиотека PyQt 5 не входит в комплект поставки Python, и прежде чем начать изучение ее основ, необходимо установить эту библиотеку на компьютер.

В настоящее время установка библиотеки PyQt 5 выполняется исключительно просто. Для этого достаточно запустить командную строку и отдать в ней команду:

```
pip3 install PyQt5
```

Утилита pip3, поставляемая в составе Python и предназначенная для установки дополнительных библиотек, самостоятельно загрузит последнюю версию PyQt 5 и установит ее по пути <каталог, в котором установлен Python>\lib\site-packages\PyQt5.

Внимание!

При установке PyQt таким способом устанавливаются только компоненты библиотеки, необходимые для запуска программ. Средства разработчика (такие как программа Designer) и дополнительные компоненты, в частности клиентские части серверных СУБД, должны быть установлены отдельно.

Чтобы проверить правильность установки, выведем версии PyQt и Qt:

```
>>> from PyQt5 import QtCore
>>> QtCore.PYQT_VERSION_STR
'5.9.2'
>>> QtCore.QT_VERSION_STR
'5.9.3'
```

17.2. Первая программа

При изучении языков и технологий принято начинать с программы, выводящей надпись «Привет, мир!». Не станем нарушать традицию и напишем программу (листинг 17.1), создающую окно с приветствием и кнопкой для закрытия окна (рис. 17.1).

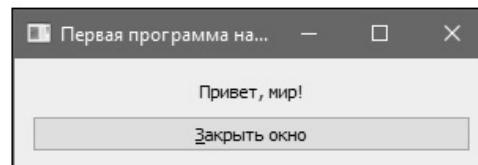


Рис. 17.1. Результат выполнения листинга 17.1

Листинг 17.1. Первая программа на PyQt

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

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Первая программа на PyQt")
window.resize(300, 70)
label = QtWidgets.QLabel("<center>Привет, мир!</center>")
btnQuit = QtWidgets.QPushButton("&Закрыть окно")
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(label)
vbox.addWidget(btnQuit)
window.setLayout(vbox)
btnQuit.clicked.connect(app.quit)
window.show()
sys.exit(app.exec_())
```

Для создания файла с программой можно по-прежнему пользоваться редактором IDLE. Однако запуск оконного приложения из IDLE нажатием клавиши <F5> приводит к очень неприятным артефактам (в частности, при завершении программы ее главное окно остается на экране) и даже аварийному завершению работы редактора. Поэтому запускать оконные приложения следует двойным щелчком на значке файла.

До сих пор мы создавали файлы с расширением `py` и все результаты выполнения программы выводили в консоль. Оконное приложение также можно сохранить с расширением `py`, но тогда при его запуске, помимо основного окна, также будет выведено окно консоли, что, впрочем, на этапе разработки дает возможность вывести отладочную информацию (таким способом мы будем пользоваться в дальнейших примерах). Чтобы избавиться от окна консоли, следует сохранять файл с расширением `pyw`.

Попробуйте создать два файла с различными расширениями и запустить двойным щелчком каждый из них.

17.3. Структура PyQt-программы

Запускать программу мы научились. Теперь рассмотрим код из листинга 17.1 построчно.

В первой строке указывается кодировка файла. Поскольку в Python 3 по умолчанию для сохранения исходного кода используется кодировка UTF-8, эту строку можно и не указывать. Во второй строке импортируется модуль `QtWidgets` — он содержит классы, реализующие компоненты графического интерфейса: окна, надписи, кнопки, текстовые поля и др. В третьей строке импортируется модуль `sys`, из которого нам потребуется список параметров, переданных в командной строке (`argv`), а также функция `exit()`, позволяющая завершить выполнение программы.

Выражение:

```
app = QtWidgets.QApplication(sys.argv)
```

создает объект приложения в виде экземпляра класса `QApplication`. Конструктор этого класса принимает список параметров, переданных в командной строке. Следует помнить, что в программе всегда должен быть объект приложения, причем обязательно только один. Может показаться, что после создания объекта он в программе больше нигде не используется, однако надо понимать, что с его помощью осуществляется управление приложением незаметно для нас. Получить доступ к этому объекту из любого места в программе можно через атрибут `qApp` из модуля `QtWidgets`. Например, вывести список параметров, переданных в командной строке, можно так:

```
print(QtWidgets.qApp.argv())
```

Следующее выражение:

```
window = QtWidgets.QWidget()
```

создает объект окна в виде экземпляра класса `QWidget`. Этот класс наследуют практически все классы, реализующие компоненты графического интерфейса. И любой компонент, не имеющий родителя, обладает своим собственным окном.

Выражение:

```
window.setWindowTitle("Первая программа на PyQt")
```

задает текст, который будет выводиться в заголовке окна, для чего используется метод `setWindowTitle()`.

Очередное выражение:

```
window.resize(300, 70)
```

задает минимальные размеры окна. В первом параметре метода `resize()` указывается ширина окна, а во втором параметре — его высота. При этом надо учитывать, что метод `resize()` устанавливает размеры не самого окна, а его клиентской области, при этом размеры заголовка и ширина границ окна не учитываются. Также следует помнить, что эти размеры являются рекомендацией, — т. е., если компоненты не помещаются в окне, оно будет увеличено.

Выражение:

```
label = QtWidgets.QLabel("<center>Привёт, мир!</center>")
```

создает объект надписи. Текст надписи задается в качестве параметра в конструкторе класса `QLabel`. Обратите внимание, что внутри строки мы указали HTML-теги, — а именно: с помощью тега `<center>` произвели выравнивание текста по центру компонента. Возможность

использования HTML-тегов и CSS-атрибутов является отличительной чертой библиотеки PyQt — например, внутри надписи можно вывести таблицу или отобразить изображение. Это очень удобно.

Следующее выражение:

```
btnQuit = QtWidgets.QPushButton("&Закрыть окно")
```

создает объект кнопки. Текст, который будет отображен на кнопке, задается в качестве параметра в конструкторе класса QPushButton. Обратите внимание на символ & перед буквой з — таким образом задаются клавиши быстрого доступа. Если нажать одновременно клавишу <Alt> и клавишу с буквой, перед которой в строке указан символ &, то кнопка срабатывает.

Выражение:

```
vbox = QtWidgets.QVBoxLayout()
```

создает вертикальный контейнер. Все компоненты, добавляемые в этот контейнер, будут располагаться по вертикали сверху вниз в порядке добавления, при этом размеры добавленных компонентов будут подогнаны под размеры контейнера. При изменении размеров контейнера будет произведено изменение размеров всех компонентов.

В следующих двух выражениях:

```
vbox.addWidget(label)  
vbox.addWidget(btnQuit)
```

с помощью метода addWidget() производится добавление созданных ранее объектов надписи и кнопки в вертикальный контейнер. Так как объект надписи добавляется первым, он будет расположен над кнопкой. При добавлении компонентов в контейнер они автоматически становятся потомками контейнера.

Новое выражение:

```
window.setLayout(vbox)
```

добавляет контейнер в основное окно с помощью метода setLayout(). Таким образом, контейнер становится потомком основного окна.

Выражение:

```
btnQuit.clicked.connect(app.quit)
```

назначает обработчик сигнала clicked() кнопки, который генерируется при ее нажатии. Этот сигнал доступен через одноименный атрибут класса кнопки и поддерживает метод connect(), который и назначает для него обработчик, передаваемый первым параметром. Обработчик представляет собой метод quit() объекта приложения, выполняющий немедленное завершение его работы. Такой метод принято называть *слотом*.

Пояснение

Сигналом в PyQt называется особое уведомление, генерируемое при наступлении какого-либо события в приложении: нажатия кнопки, ввода символа в текстовое поле, закрытия окна и пр.

Очередное выражение:

```
window.show()
```

выводит на экран окно и все компоненты, которые мы ранее в него добавили.

И, наконец, последнее выражение:

```
sys.exit(app.exec_())
```

запускает бесконечный цикл обработки событий в приложении.

Код, расположенный после вызова метода `exec_()`, будет выполнен только после завершения работы приложения, — поскольку результат выполнения метода `exec_()` мы передаем функции `exit()`, дальнейшее выполнение программы будет прекращено, а код возврата передан операционной системе.

17.4. ООП-стиль создания окна

Библиотека PyQt написана в объектно-ориентированном стиле (ООП-стиле) и содержит несколько сотен классов. Иерархия наследования всех классов имеет слишком большой размер, и приводить ее в книге возможности нет. Тем не менее, чтобы показать зависимости, при описании компонентов иерархия наследования конкретного класса будет показываться. В качестве примера выведем базовые классы класса `QWidget`:

```
>>> from PyQt5 import QtWidgets
>>> QtWidgets.QWidget.__bases__
(<class 'PyQt5.QtCore.QObject'>, <class 'PyQt5.QtGui.QPaintDevice'>)
```

Как видно из примера, класс `QWidget` наследует два класса: `QObject` и `QPaintDevice`. Класс `QObject` является классом верхнего уровня, и его в PyQt наследуют большинство классов. В свою очередь, класс `QWidget` является базовым классом для всех визуальных компонентов.

Внимание!

В описании каждого класса PyQt приводятся лишь атрибуты, методы, сигналы и слоты, определенные непосредственно в описываемом классе. Атрибуты, методы, сигналы и слоты базовых классов там не описываются — присутствуют лишь ссылки на соответствующие страницы документации.

В своих программах вы можете наследовать стандартные классы и добавлять новую функциональность. В качестве примера переделаем соответствующим образом код из листинга 17.1 и создадим окно в ООП-стиле (листинг 17.2).

Листинг 17.2. ООП-стиль создания окна

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

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.label = QtWidgets.QLabel("Привет, мир!")
        self.label.setAlignment(QtCore.Qt.AlignHCenter)
        self.btnExit = QtWidgets.QPushButton("&Закрыть окно")
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.btnExit)
        self.setLayout(self.vbox)
        self.btnExit.clicked.connect(QtWidgets.qApp.quit)
```

```
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()      # Создаем экземпляр класса
    window.setWindowTitle("ООП-стиль создания окна")
    window.resize(300, 70)
    window.show()            # Отображаем окно
    sys.exit(app.exec_())    # Запускаем цикл обработки событий
```

В первых двух строках кода, как обычно, указывается кодировка файла и импортируются необходимые модули. На этот раз, помимо уже знакомого модуля `QtWidgets`, нам понадобится модуль `QtCore`, в котором объявлены атрибуты, задающие, в том числе, и режим выравнивания текста в объекте надписи.

Далее мы определяем класс `MyWindow`, который наследует класс `QWidget`:

```
class MyWindow(QtWidgets.QWidget):
```

Можно наследовать и другие классы, являющиеся наследниками `QWidget`, — например, `QFrame` (окно с рамкой) или `QDialog` (диалоговое окно). При наследовании класса `QDialog` окно будет выравниваться по центру экрана (или по центру родительского окна) и иметь в заголовке окна только две кнопки: **Справка** и **Закрыть**. Кроме того, можно наследовать класс `QMainWindow`, который представляет главное окно приложения с меню, панелями инструментов и строкой состояния. Наследование класса `QMainWindow` имеет свои отличия, которые мы рассмотрим в *главе 27*.

Выражение:

```
def __init__(self, parent=None):
```

определяет конструктор класса. В качестве параметров он принимает ссылки на экземпляр класса (`self`) и на родительский компонент (`parent`). Родительский компонент может отсутствовать, поэтому в определении конструктора параметру присваивается значение по умолчанию (`None`). Внутри метода `__init__()` вызывается конструктор базового класса, и ему передается ссылка на родительский компонент:

```
QtWidgets.QWidget.__init__(self, parent)
```

Следующие выражения внутри конструктора создают объекты надписи, кнопки и контейнера, затем добавляют компоненты в контейнер, а сам контейнер — в основное окно. Следует обратить внимание на то, что объекты надписи и кнопки сохраняются в атрибутах экземпляра класса. В дальнейшем из методов класса можно управлять этими объектами — например, изменять текст надписи. Если объекты не сохранить, то получить к ним доступ будет не так просто.

В предыдущем примере (см. листинг 17.1) мы выравнивали надпись с помощью HTML-тегов. Однако выравнивание можно задать и вызовом метода `setAlignment()`, которому следует передать атрибут `AlignHCenter` из модуля `QtCore`:

```
self.label.setAlignment(QtCore.Qt.AlignHCenter)
```

В выражении, назначающем обработчик сигнала:

```
self.btnExit.clicked.connect(QtWidgets.qApp.quit)
```

мы получаем доступ к объекту приложения через рассмотренный ранее атрибут `qApp` модуля `QtWidgets`.

Создание объекта приложения и экземпляра класса `MyWindow` производится внутри условия:

```
if __name__ == "__main__":
```

Если вы внимательно читали первую часть книги, то уже знаете, что атрибут модуля `__name__` будет содержать значение `__main__` только в случае запуска модуля как главной программы. Если модуль импортировать, этот атрибут будет содержать другое значение. Поэтому весь последующий код создания объекта приложения и объекта окна выполняется только при запуске программы двойным щелчком на значке файла. Может возникнуть вопрос, зачем это нужно? Дело в том, что одним из преимуществ ООП-стиля программирования является повторное использование кода. Следовательно, можно импортировать модуль и использовать класс `MyWindow` в другом приложении.

Рассмотрим эту возможность на примере, для чего сохраним код из листинга 17.2 в файле с именем `MyWindow.py`, а затем создадим в той же папке еще один файл (например, с именем `test.pyw`) и вставим в него код из листинга 17.3.

Листинг 17.3. Повторное использование кода при ООП-стиле

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

class MyDialog(QtWidgets.QDialog):
    def __init__(self, parent=None):
        QtWidgets.QDialog.__init__(self, parent)
        self.myWidget = MyWindow.MyWindow()
        self.myWidget.vbox.setContentsMargins(0, 0, 0, 0)
        self.button = QtWidgets.QPushButton("Изменить надпись")
        mainBox = QtWidgets.QVBoxLayout()
        mainBox.addWidget(self.myWidget)
        mainBox.addWidget(self.button)
        self.setLayout(mainBox)
        self.button.clicked.connect(self.on_clicked)

    def on_clicked(self):
        self.myWidget.label.setText("Новая надпись")
        self.button.setDisabled(True)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyDialog()
    window.setWindowTitle("Преимущество ООП-стиля")
    window.resize(300, 100)
    window.show()
    sys.exit(app.exec_())
```

Теперь запустим файл `test.pyw` двойным щелчком — на экране откроется окно с надписью и двумя кнопками (рис. 17.2). По нажатию на кнопку **Изменить надпись** производится изме-

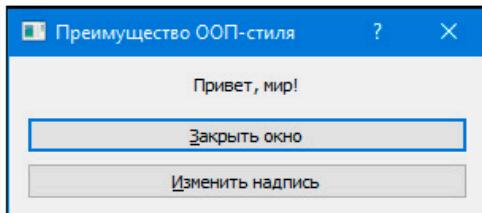


Рис. 17.2. Результат выполнения листинга 17.3

нение текста надписи, и кнопка делается неактивной. Нажатие кнопки **Закрыть окно** будет по-прежнему завершать выполнение приложения.

В этом примере мы создали класс `MyDialog`, который наследует класс `QDialog`. Поэтому при выводе окно автоматически выравнивается по центру экрана, а в заголовке окна выводятся только две кнопки: **Справка** и **Закрыть**. Внутри конструктора мы создаем экземпляр класса `myWindow` и сохраняем его в атрибуте `myWidget`:

```
self.myWidget = MyWindow.MyWindow()
```

С его помощью позже мы получим доступ ко всем атрибутам класса `myWindow`. Например, в следующей строке произведем изменение отступа между границами контейнера и границами соседних элементов:

```
self.myWidget.vbox.setContentsMargins(0, 0, 0, 0)
```

В следующих инструкциях внутри конструктора создаются кнопки и контейнер, затем экземпляр класса `MyWindow` и кнопка добавляются в контейнер, а сам контейнер помещается в основное окно.

Выражение:

```
self.button.clicked.connect(self.on_clicked)
```

назначает обработчик нажатия кнопки. В качестве параметра указывается ссылка на метод `on_clicked()`, внутри которого производится изменение текста надписи (с помощью метода `setText()`), и кнопка делается неактивной (с помощью метода `setDisabled()`). Внутри метода `on_clicked()` доступен указатель `self`, через который можно получить доступ к атрибутам классов `MyDialog` и `MyWindow`.

Вот так и производится повторное использование ранее написанного кода: мы создаем класс и сохраняем его внутри отдельного модуля, а чтобы протестировать модуль или использовать его как отдельное приложение, размещаем код создания объекта приложения и объекта окна внутри условия:

```
if __name__ == "__main__":
```

Тогда при запуске с помощью двойного щелчка на значке файла производится выполнение кода как отдельного приложения. Если модуль импортируется, то создание объекта приложения не производится, и мы можем использовать класс в других приложениях. Например, так, как это было сделано в листинге 17.3, или путем наследования класса и добавления или переопределения методов.

В некоторых случаях использование ООП-стиля является обязательным. Например, чтобы обработать нажатие клавиши на клавиатуре, необходимо наследовать какой-либо класс и переопределить в нем метод с предопределенным названием. Какие методы необходимо переопределять, мы рассмотрим при изучении обработки событий.

17.5. Создание окна с помощью программы Qt Designer

Если вы ранее пользовались Visual Studio или Delphi, то вспомните, как с помощью мыши размещали на форме компоненты: щелкали левой кнопкой мыши на нужной кнопке в панели инструментов и перетаскивали компонент на форму, затем с помощью инспектора свойств производили настройку значений некоторых свойств, а остальные свойства получали значения по умолчанию. При этом весь код генерировался автоматически. Произвести аналогичную операцию в PyQt позволяет программа Qt Designer, которая входит в состав этой библиотеки.

К огромному сожалению, в составе последних версий PyQt эта полезная программа отсутствует. Однако ее можно установить отдельно в составе программного пакета PyQt 5 Tools, отдав в командной строке команду:

```
pip3 install pyqt5-tools
```

17.5.1. Создание формы

Запустить Qt Designer можно щелчком на исполняемом файле `designer.exe`, который располагается по пути `<путь, по которому установлен Python>\Lib\site-packages\pyqt5-tools`. К сожалению, через меню **Пуск** это сделать не получится.

В окне **New Form** открывшегося окна (рис. 17.3) выбираем пункт **Widget** и нажимаем кнопку **Create** — откроется окно с пустой формой, на которую с помощью мыши можно перетаскивать компоненты из панели **Widget Box**.

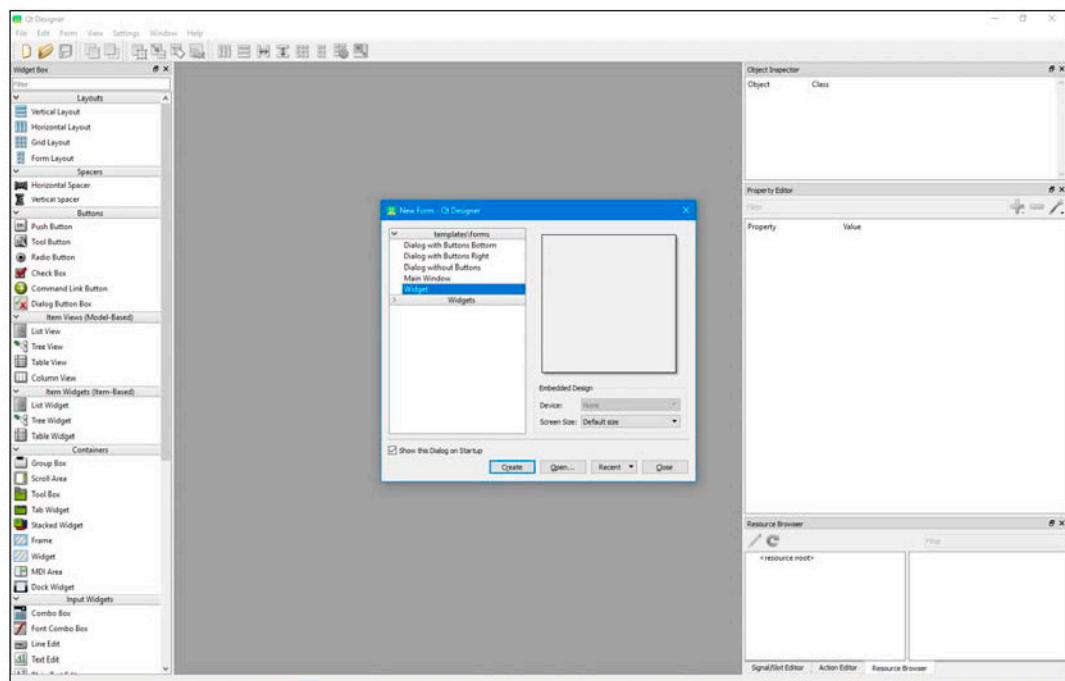


Рис. 17.3. Программа Qt Designer

В качестве примера добавим на форму надпись и кнопку. Для этого на панели **Widget Box** в группе **Display Widgets** щелкнем левой кнопкой мыши на пункте **Label** и, не отпуская кнопку мыши, перетащим компонент на форму. Затем проделаем аналогичную операцию с компонентом **Push Button**, находящимся в группе **Buttons**, и разместим его ниже надписи. Теперь выделим одновременно надпись и кнопку, щелкнем правой кнопкой мыши на любом компоненте и в контекстном меню выберем пункт **Lay out | Lay Out Vertically**. Чтобы компоненты занимали всю область формы, щелкнем правой кнопкой мыши на свободном месте формы и в контекстном меню выберем пункт **Lay out | Lay Out Horizontally**.

Теперь изменим некоторые свойства окна. Для этого в панели **Object Inspector** (рис. 17.4) выделим первый пункт (**Form**), перейдем в панель **Property Editor**, найдем свойство **objectName** и справа от свойства введем значение **MyForm**. Затем найдем свойство **geometry**, щелкнем мышью на значке уголка слева, чтобы отобразить скрытые свойства, и зададим ширину равной 300, а высоту равной 70 (рис. 17.5). — размеры формы автоматически изменятся. Указать текст, который будет отображаться в заголовке окна, позволяет свойство **windowTitle**.

Чтобы изменить свойства надписи, следует выделить компонент с помощью мыши или выбрать соответствующий ему пункт в панели **Object Inspector**. Для примера изменим значение свойства **text** (оно задает текст надписи). После чего найдем свойство **alignment**, щелкнем мышью на значке уголка слева, чтобы отобразить скрытые свойства, и укажем для свойства **Horizontal** значение **AlignHCenter**. Теперь выделим кнопку и изменим значение свойства **objectName** на **btnQuit**, а в свойстве **text** укажем текст надписи, которая будет выводиться на кнопке. (Кстати, изменить текст надписи или кнопки также можно, выполнив двойной щелчок мышью на компоненте.)

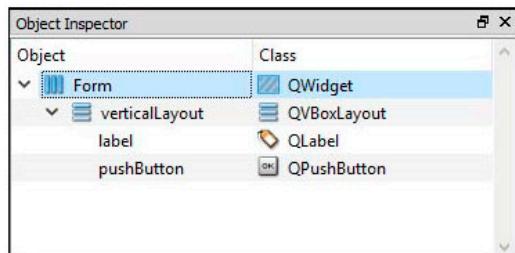


Рис. 17.4. Панель Object Inspector

Property Editor	
Filter	
MyForm : QWidget	
Property	Value
objectName	MyForm
enabled	<input checked="" type="checkbox"/>
geometry	[0, 0, 300 x 70]
X	0
Y	0
Width	300
Height	70
sizePolicy	[Preferred, Preferred, 0, 0]
minimumSize	0 x 0
maximumSize	16777215 x 16777215
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited
font	A [MS Shell Dlg 2, 8]
cursor	Arrow
mouseTracking	<input type="checkbox"/>
tabletTracking	<input type="checkbox"/>
focusPolicy	NoFocus
contextMenuPolicy	DefaultContextMenu

Рис. 17.5. Панель Property Editor

Закончив, выберем в меню **File** пункт **Save** и сохраним готовую форму в файл **MyForm.ui**. При необходимости внести в этот файл какие-либо изменения, его можно открыть в программе Qt Designer, выбрав в меню **File** пункт **Open**.

17.5.2. Использование UI-файла в программе

Как вы можете убедиться, внутри UI-файла содержится текст в XML-формате, а не программный код на языке Python. Следовательно, подключить файл с помощью инструкции `import` не получится. Чтобы использовать UI-файл внутри программы, следует воспользоваться модулем `uic`, который входит в состав библиотеки PyQt. Прежде чем использовать функции из этого модуля, необходимо подключить модуль с помощью инструкции:

```
from PyQt5 import uic
```

Для загрузки UI-файла предназначена функция `loadUi()`. Формат функции:

```
loadUi(<ui-файл>[, <Экземпляр класса>])
```

Если второй параметр не указан, функция возвращает ссылку на объект формы. С помощью этой ссылки можно получить доступ к компонентам формы и, например, назначить обработчики сигналов (листинг 17.4). Имена компонентов задаются в программе Qt Designer в свойстве `objectName`.

Листинг 17.4. Использование функции `loadUi()`. Вариант 1

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets, uic
import sys
app = QtWidgets.QApplication(sys.argv)
window = uic.loadUi("MyForm.ui")
window.btnExit.clicked.connect(app.quit)
window.show()
sys.exit(app.exec_())
```

Если во втором параметре указать ссылку на экземпляр класса, то все компоненты формы будут доступны через указатель `self` (листинг 17.5).

Листинг 17.5. Использование функции `loadUi()`. Вариант 2

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

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        uic.loadUi("MyForm.ui", self)
        self.btnExit.clicked.connect(QtWidgets.qApp.quit)

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

```
window.show()
sys.exit(app.exec_())
```

Загрузить UI-файл позволяет также функция `loadUiType()` — она возвращает кортеж из двух элементов: ссылки на класс формы и ссылки на базовый класс. Так как функция возвращает ссылку на класс, а не на экземпляр класса, мы можем создать множество экземпляров класса. После создания экземпляра класса формы необходимо вызвать метод `setupUi()` и передать ему указатель `self` (листинг 17.6).

Листинг 17.6. Использование функции `loadUiType()`. Вариант 1

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

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        Form, Base = uic.loadUiType("MyForm.ui")
        self.ui = Form()
        self.ui.setupUi(self)
        self.ui.btnExit.clicked.connect(QtWidgets.qApp.quit)
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Загрузить UI-файл можно и вне класса, после чего указать класс формы во втором параметре в списке наследования, — в этом случае наш класс унаследует все методы класса формы (листинг 17.7).

Листинг 17.7. Использование функции `loadUiType()`. Вариант 2

```
from PyQt5 import QtWidgets, uic

Form, Base = uic.loadUiType("MyForm.ui")
class MyWindow(QtWidgets.QWidget, Form):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setupUi(self)
        self.btnExit.clicked.connect(QtWidgets.qApp.quit)

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

17.5.3. Преобразование UI-файла в PY-файл

Вместо подключения UI-файла можно сгенерировать на его основе программный код на языке Python. Для этого служит утилита `pyuic5`, чей исполняемый файл располагается в каталоге `<путь, по которому установлен Python>\Scripts`. Запустим командную строку и перейдем в каталог, в котором находится UI-файл. Для генерации Python-программы выполним команду:

```
pyuic5 MyForm.ui -o ui_MyForm.py
```

В результате будет создан файл `ui_MyForm.py`, который мы уже можем подключить с помощью инструкции `import`. Внутри файла находится класс `Ui_MyForm` с методами `setupUi()` и `retranslateUi()`. При использовании процедурного стиля программирования следует создать экземпляр класса формы, а затем вызвать метод `setupUi()` и передать ему ссылку на экземпляр окна (листинг 17.8).

Листинг 17.8. Использование класса формы. Вариант 1

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys, ui_MyForm
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
ui = ui_MyForm.Ui_MyForm()
ui.setupUi(window)
ui.btnExit.clicked.connect(QtWidgets.qApp.quit)
window.show()
sys.exit(app.exec_())
```

При использовании ООП-стиля программирования следует создать экземпляр класса формы, а затем вызвать метод `setupUi()` и передать ему указатель `self` (листинг 17.9).

Листинг 17.9. Использование класса формы. Вариант 2

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

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.ui = ui_MyForm.Ui_MyForm()
        self.ui.setupUi(self)
        self.ui.btnExit.clicked.connect(QtWidgets.qApp.quit)

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

Класс формы можно указать во втором параметре в списке наследования — в этом случае он унаследует все методы класса формы (листинг 17.10).

Листинг 17.10. Использование класса формы. Вариант 3

```
from PyQt5 import QtWidgets
import ui_MyForm

class MyWindow(QtWidgets.QWidget, ui_MyForm.Ui_MyForm):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setupUi(self)
        self.btnExit.clicked.connect(QtWidgets.qApp.quit)

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

Как видите, в PyQt можно создавать формы, размещать компоненты с помощью мыши, а затем непосредственно подключать UI-файлы в программе или преобразовывать их в Python-код с помощью утилиты `pyuic5`, — все это очень удобно. Тем не менее, чтобы полностью овладеть программированием на PyQt, необходимо уметь создавать код вручную. Поэтому в оставшейся части книги мы больше не станем задействовать программу Qt Designer.

17.6. Модули PyQt 5

В состав библиотеки PyQt 5 входит множество модулей, объединенных в пакет PyQt5. Упомянем самые важные из них:

- ◆ `QtCore` — содержит классы, не связанные с реализацией графического интерфейса. От этого модуля зависят все остальные модули;
- ◆ `QtGui` — содержит классы, реализующие низкоуровневую работу с оконными элементами, обработку сигналов, вывод двухмерной графики и текста и др.;
- ◆ `QtWidgets` — содержит классы, реализующие компоненты графического интерфейса: окна, диалоговые окна, надписи, кнопки, текстовые поля и др.;
- ◆ `QtWebEngineCore` — включает низкоуровневые классы для отображения веб-страниц;
- ◆ `QtWebEngineWidgets` — реализует высокоуровневые компоненты графического интерфейса, предназначенные для вывода веб-страниц и использующие модуль `QtWebEngineCore`;

ПРИМЕЧАНИЕ

Ранее для вывода веб-страниц использовались модули `QtWebKit` и `QtWebKitWidgets`. Однако в версии PyQt 5.5 они были объявлены нерекомендованными для использования, а в версии 5.6 удалены.

- ◆ `QtMultimedia` — включает низкоуровневые классы для работы с мультимедиа;

- ◆ `QtMultimediaWidgets` — реализует высококуровневые компоненты графического интерфейса с мультимедиа, использующие модуль `QtMultimedia`;
- ◆ `QtPrintSupport` — содержит классы, обеспечивающие поддержку печати и предварительного просмотра документов;
- ◆ `QtSql` — включает поддержку работы с базами данных, а также реализацию `SQLite`;
- ◆ `QtSvg` — позволяет работать с векторной графикой (`SVG`);
- ◆ `QtNetwork` — содержит классы, предназначенные для работы с сетью;
- ◆ `QtXml` и `QtXmlPatterns` — предназначены для обработки `XML`;
- ◆ `QtHelp` — содержит инструменты для создания интерактивных справочных систем;
- ◆ `QtWinExtras` — включает поддержку специфических возможностей Microsoft Windows;
- ◆ `Qt` — включает классы из всех модулей сразу.

ПРИМЕЧАНИЕ

Модуль `QtOpenGL`, обеспечивающий поддержку `OpenGL`, в версии PyQt 5.9 был объявлен нерекомендованным к использованию и будет удален в одной из последующих версий этой библиотеки. Его функциональность перенесена в модуль `QtGui`.

Для подключения модулей используется следующий синтаксис:

```
from PyQt5 import <Названия модулей через запятую>
```

Так, например, можно подключить модули `QtCore` и `QtWidgets`:

```
from PyQt5 import QtCore, QtWidgets
```

В этой книге мы не станем рассматривать все упомянутые модули — чтобы получить информацию по не рассмотренным здесь модулям, обращайтесь к соответствующей документации.

17.7. Типы данных в PyQt

Библиотека PyQt является надстройкой над написанной на языке C++ библиотекой Qt. Последняя содержит множество классов, которые расширяют стандартные типы данных языка C++ и реализуют динамические массивы, ассоциативные массивы, множества и др. Все эти классы очень помогают при программировании на языке C++, но для языка Python они не представляют особого интереса, т. к. весь этот функционал содержит стандартные типы данных. Тем не менее, при чтении документации вы столкнетесь с ними, поэтому сейчас мы кратко рассмотрим основные типы:

- ◆ `QByteArray` — массив байтов. Преобразуется в тип `bytes`:

```
>>> from PyQt5 import QtCore  
>>> arr = QtCore.QByteArray(bytes("str", "cp1251"))  
>>> arr  
PyQt5.QtCore.QByteArray(b'str')  
>>> bytes(arr)  
b'str'
```

- ◆ `QVariant` — может хранить данные любого типа. Создать экземпляр этого класса можно вызовом конструктора, передав ему нужное значение. А чтобы преобразовать данные, хранящиеся в экземпляре класса `QVariant`, в тип данных Python, нужно вызвать метод `value()`:

```
>>> from PyQt5 import QtCore  
>>> n = QtCore.QVariant(10)  
>>> n  
<PyQt5.QtCore.QVariant object at 0x00FD8D50>  
>>> n.value()  
10
```

Также можно создать «пустой» экземпляр класса `QVariant`, вызвав конструктор без параметров:

```
>>> QtCore.QVariant() # Пустой объект  
<PyQt5.QtCore.QVariant object at 0x00FD8420>
```

Если какой-либо метод ожидает данные типа `QVariant`, ему можно передать данные любого типа.

Еще этот класс поддерживает метод `typeName()`, возвращающий наименование типа хранящихся в экземпляре данных:

```
>>> from PyQt5 import QtCore  
>>> n = QtCore.QVariant(10)  
>>> n.typeName()  
'int'
```

Кроме того, PyQt 5 поддерживает классы `QDate` (значение даты), `QTime` (значение времени), `QDateTime` (значение даты и времени), `QTextStream` (текстовый поток), `QUrl` (URL-адрес) и некоторые другие.

17.8. Управление основным циклом приложения

Для взаимодействия с системой и обработки возникающих сигналов предназначен основной цикл приложения. После вызова метода `exec_()` программа переходит в бесконечный цикл. Инструкции, расположенные после вызова этого метода, будут выполнены только после завершения работы всего приложения. Цикл автоматически прерывается после закрытия последнего открытого окна приложения. С помощью статического метода `setQuitOnLastWindowClosed()` класса `QApplication` это поведение можно изменить.

Чтобы завершить работу приложения, необходимо вызвать слот `quit()` или метод `exit([returnCode=0])` класса `QApplication`. Поскольку программа находится внутри цикла, вызвать эти методы можно лишь при наступлении какого-либо события, — например, при нажатии пользователем кнопки.

После возникновения любого сигнала основной цикл прерывается, и управление передается в обработчик этого сигнала. После завершения работы обработчика управление возвращается основному циклу приложения.

Если внутри обработчика выполняется длительная операция, программа перестает реагировать на события. В качестве примера изобразим длительный процесс с помощью функции `sleep()` из модуля `time` (листинг 17.11).

Листинг 17.11. Выполнение длительной операции

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

```
def on_clicked():
    time.sleep(10) # "Засыпаем" на 10 секунд

app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Запустить процесс")
button.resize(200, 40)
button.clicked.connect(on_clicked)
button.show()
sys.exit(app.exec_())
```

В этом примере при нажатии кнопки **Запустить процесс** вызывается функция `on_clicked()`, внутри которой мы приостанавливаем выполнение программы на десять секунд и тем самым прерываем основной цикл. Попробуйте нажать кнопку, перекрыть окно другим окном, а затем заново его отобразить, — вам не удастся это сделать, поскольку окно перестает реагировать на любые события, пока не закончится выполнение процесса. Короче говоря, программа просто зависнет.

Длительную операцию можно разбить на несколько этапов и по завершении каждого этапа выходить в основной цикл с помощью статического метода `processEvents([flags=AllEvents])` класса `QCoreApplication`, от которого наследуется класс `QApplication`. Переделаем предыдущую программу, инсенировав с помощью цикла длительную операцию, которая выполняется 20 секунд (листинг 17.12).

Листинг 17.12. Использование метода `processEvents()`

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

def on_clicked():
    button.setDisabled(True) # Делаем кнопку неактивной
    for i in range(1, 21):
        QtWidgets.qApp.processEvents() # Запускаем оборот цикла
        time.sleep(1) # "Засыпаем" на 1 секунду
        print("step -", i)
    button.setDisabled(False) # Делаем кнопку активной

app = QtWidgets.QApplication(sys.argv)
button = QtWidgets.QPushButton("Запустить процесс")
button.resize(200, 40)
button.clicked.connect(on_clicked)
button.show()
sys.exit(app.exec_())
```

В этом примере длительная операция разбита на одинаковые этапы, после выполнения каждого из которых выполняется выход в основной цикл приложения. Теперь при перекрытии окна и повторном его отображении оно будет перерисовано — таким образом, приложение по-прежнему будет взаимодействовать с системой, хотя и с некоторой задержкой.

17.9. Многопоточные приложения

При обработке больших объемов данных не всегда можно равномерно разбить операцию на небольшие по времени этапы, поэтому при использовании метода `processEvents()` возможны проблемы, и тогда имеет смысл вынести длительную операцию в отдельный поток, — в этом случае операция станет выполняться параллельно с основным циклом приложения и не будет его блокировать.

В одном процессе можно запустить сразу несколько независимых потоков, и если ваш компьютер оснащен многоядерным процессором, потоки будут равномерно распределены по его ядрам. За счет этого можно не только избежать блокировки GUI-потока приложения, в котором выполняется обновление его интерфейса, но и значительно увеличить эффективность выполнения кода. Завершение основного цикла приложения приводит к завершению работы всех потоков.

17.9.1. Класс `QThread`: создание потока

Для создания потока в PyQt предназначен класс `QThread`, который объявлен в модуле `QtCore` и наследует класс `QObject`. Конструктор класса `QThread` имеет следующий формат:

```
<Объект> = QThread([parent=None])
```

Чтобы использовать потоки, следует создать класс, который будет наследником класса `QThread`, и определить в нем метод `run()`. Код, расположенный в методе `run()`, будет выполняться в отдельном потоке, а после завершения выполнения метода `run()` этот поток прекратит свое существование. Затем нужно создать экземпляр класса и вызвать метод `start()`, который после запуска потока вызовет метод `run()`. Обратите внимание, что если напрямую вызвать метод `run()`, то код станет выполняться в основном, а не в отдельном потоке. Метод `start()` имеет следующий формат:

```
start([priority=QThread.InheritPriority])
```

Параметр `priority` задает приоритет выполнения потока по отношению к другим потокам. Следует учитывать, что при наличии потока с самым высоким приоритетом поток с самым низким приоритетом в некоторых операционных системах может быть просто проигнорирован. Приведем допустимые значения параметра (в порядке увеличения приоритета) и соответствующие им атрибуты из класса `QThread`:

- ◆ 0 — `IdlePriority` — самый низкий приоритет;
- ◆ 1 — `LowestPriority`;
- ◆ 2 — `LowPriority`;
- ◆ 3 — `NormalPriority`;
- ◆ 4 — `HighPriority`;
- ◆ 5 — `HighestPriority`;
- ◆ 6 — `TimeCriticalPriority` — самый высокий приоритет;
- ◆ 7 — `InheritPriority` — автоматический выбор приоритета (значение по умолчанию).

Задать приоритет потока также позволяет метод `setPriority(<Приоритет>)`. Узнать, какой приоритет использует запущенный поток, можно с помощью метода `priority()`.

После запуска потока генерируется сигнал `started()`, а после завершения — сигнал `finished()`. Назначив обработчики этим сигналам, можно контролировать статус потока из

основного цикла приложения. Если необходимо узнать текущий статус, следует воспользоваться методами `isRunning()` и `isFinished()`: метод `isRunning()` возвращает значение `True`, если поток выполняется, а метод `isFinished()` — значение `True`, если поток закончил выполнение.

Потоки выполняются внутри одного процесса и имеют доступ ко всем глобальным переменным. Однако следует учитывать, что из потока нельзя изменять что-либо в GUI-потоке приложения, — например, выводить текст на надпись. Для изменения данных в GUI-потоке нужно использовать сигналы. Внутри потока у нужного сигнала вызывается метод `emit()`, который, собственно, и выполняет его генерацию. В параметрах метода `emit()` можно указать данные, которые будут переданы обработчику сигнала. А внутри GUI-потока назначаем обработчик этого сигнала и в обработчике пишем код, который и будет обновлять интерфейс приложения.

Рассмотрим использование класса `QThread` на примере (листинг 17.13).

Листинг 17.13. Использование класса `QThread`

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
class MyThread(QtCore.QThread):
    mysignal = QtCore.pyqtSignal(str)
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
    def run(self):
        for i in range(1, 21):
            self.sleep(3)                      # "Засыпаем" на 3 секунды
            # Передача данных из потока через сигнал
            selfmysignal.emit("i = %s" % i)

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.label = QtWidgets.QLabel("Нажмите кнопку для запуска потока")
        self.label.setAlignment(QtCore.Qt.AlignHCenter)
        self.button = QtWidgets.QPushButton("Запустить процесс")
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.button)
        self.setLayout(self.vbox)
        self.mythread = MyThread()           # Создаем экземпляр класса
        self.button.clicked.connect(self.on_clicked)
        self.mythread.started.connect(self.on_started)
        self.mythread.finished.connect(self.on_finished)
        self.mythreadmysignal.connect(self.on_change, QtCore.Qt.QueuedConnection)
    def on_clicked(self):
        self.button.setDisabled(True)      # Делаем кнопку неактивной
        self.mythread.start()             # Запускаем поток
    def on_started(self):                # Вызывается при запуске потока
        self.label.setText("Вызван метод on_started()")
```

```

def on_finished(self):           # Вызывается при завершении потока
    self.label.setText("Вызван метод on_finished()")
    self.button.setDisabled(False) # Делаем кнопку активной
def on_change(self, s):
    self.label.setText(s)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Использование класса QThread")
    window.resize(300, 70)
    window.show()
    sys.exit(app.exec_())

```

Здесь мы создали класс `MyThread`, который является наследником класса `QThread`. В нем мы определили свой собственный сигнал `mysignal`, для чего создали атрибут с таким же именем и занесли в него значение, возвращенное функцией `pyqtSignal()` из модуля `QtCore`. Функции `pyqtSignal()` мы передали в качестве параметра тип `str` (строка Python), тем самым указав PyQt, что вновь определенный сигнал будет принимать единственный параметр строкового типа:

```
mysignal = QtCore.pyqtSignal(str)
```

В том же классе мы определили обязательный для потоков метод `run()` — в нем производится имитация процесса с помощью цикла `for` и метода `sleep()`: каждые три секунды выполняется генерация сигнала `mysignal` и передача текущего значения переменной `i` в составе строки:

```
self.mysignal.emit("i = %s" % i)
```

Внутри конструктора класса `MyWindow` мы назначили обработчик этого сигнала с помощью выражения:

```
self.mythread.mysignal.connect(self.on_change, QtCore.Qt.QueuedConnection)
```

Здесь все нам уже знакомо: у свойства `mysignal` потока, которое представляет одноименный сигнал, вызывается метод `connect()`, и ему первым параметром передается обработчик. Во втором параметре метода `connect()` с помощью атрибута `QueuedConnection` указывается, что сигнал помещается в очередь обработки событий, и обработчик должен выполняться в потоке приемника сигнала, т. е. в GUI-потоке. Из GUI-потока мы можем смело изменять свойства компонентов интерфейса.

Теперь рассмотрим код метода класса `MyWindow`, который станет обработчиком сигнала `mysignal`:

```

def on_change(self, s):
    self.label.setText(s)

```

Второй параметр этого метода служит для приема параметра, переданного этому сигналу. Значение этого параметра будет выведено в надписи с помощью метода `setText()`.

Еще в конструкторе класса `MyWindow` производится создание надписи и кнопки, а затем их размещение внутри вертикального контейнера. Далее выполняется создание экземпляра класса `MyThread` и сохранение его в атрибуте `mythread`. С помощью этого атрибута мы мо-

жем управлять потоком и назначить обработчики сигналов `started()`, `finished()` и `mysignal`. Запуск потока производится с помощью метода `start()` внутри обработчика нажатия кнопки. Чтобы исключить повторный запуск потока, мы с помощью метода `setDisabled()` делаем кнопку неактивной, а после окончания работы потока внутри обработчика сигнала `finished()` опять делаем кнопку активной.

Обратите внимание, что для имитации длительного процесса мы использовали статический метод `sleep()` из класса `QThread`, а не функцию `sleep()` из модуля `time`. Вообще, приостановить выполнение потока позволяют следующие статические методы класса `QThread`:

- ◆ `sleep()` — продолжительность задается в секундах:

```
QtCore.QThread.sleep(3)      # "Засыпаем" на 3 секунды
```

- ◆ `msleep()` — продолжительность задается в миллисекундах:

```
QtCore.QThread.msleep(3000)  # "Засыпаем" на 3 секунды
```

- ◆ `usleep()` — продолжительность задается в микросекундах:

```
QtCore.QThread.usleep(3000000) # "Засыпаем" на 3 секунды
```

Еще один полезный статичный метод класса `QThread` — `yieldCurrentThread()` — немедленно приостанавливает выполнение текущего потока и передает управление следующему ожидающему выполнения потоку, если таковой есть:

```
QtCore.QThread.yieldCurrentThread()
```

17.9.2. Управление циклом внутри потока

Очень часто внутри потока одни и те же инструкции выполняются многократно. Например, при осуществлении мониторинга серверов в Интернете на каждой итерации цикла посыпается запрос к одному и тому же серверу. При этом внутри метода `run()` используется бесконечный цикл, выход из которого производится после окончания опроса всех серверов. В некоторых случаях этот цикл необходимо прервать преждевременно по нажатию кнопки пользователем. Чтобы это стало возможным, в классе, реализующем поток, следует создать атрибут, который будет содержать флаг текущего состояния. Далее на каждой итерации цикла проверяется состояние флага и при его изменении прерывается выполнение цикла. Чтобы изменить значение атрибута, создаем обработчик и связываем его с сигналом `clicked()` соответствующей кнопки. Пример запуска и остановки потока с помощью кнопок приведен в листинге 17.14.

Листинг 17.14. Запуск и остановка потока

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

class MyThread(QtCore.QThread):
    mysignal = QtCore.pyqtSignal(str)
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.running = False  # Флаг выполнения
        self.count = 0
    def run(self):
        self.running = True
        while self.running:
            self.mysignal.emit("Hello, world!")
            self.sleep(1)
        self.running = False
```

```

while self.running:      # Проверяем значение флага
    self.count += 1
    self.mysignal.emit("count = %s" % self.count)
    self.sleep(1)        # Имитируем процесс

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.label = QtWidgets.QLabel("Нажмите кнопку для запуска потока")
        self.label.setAlignment(QtCore.Qt.AlignHCenter)
        self.btnStart = QtWidgets.QPushButton("Запустить поток")
        self.btnStop = QtWidgets.QPushButton("Остановить поток")
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.btnStart)
        self.vbox.addWidget(self.btnStop)
        self.setLayout(self.vbox)
        self.mythread = MyThread()
        self.btnStart.clicked.connect(self.on_start)
        self.btnStop.clicked.connect(self.on_stop)
        self.mythread.mysignal.connect(self.on_change, QtCore.Qt.QueuedConnection)
    def on_start(self):
        if not self.mythread.isRunning():
            self.mythread.start()      # Запускаем поток
    def on_stop(self):
        self.mythread.running = False # Изменяем флаг выполнения
    def on_change(self, s):
        self.label.setText(s)
    def closeEvent(self, event):      # Вызывается при закрытии окна
        self.hide()                  # Скрываем окно
        self.mythread.running = False # Изменяем флаг выполнения
        self.mythread.wait(5000)       # Даем время, чтобы закончить
        event.accept()                # Закрываем окно

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Запуск и остановка потока")
    window.resize(300, 100)
    window.show()
    sys.exit(app.exec_())

```

В этом примере в конструкторе класса `MyThread` создается атрибут `running`, и ему присваивается значение `False`. При запуске потока внутри метода `run()` значение атрибута изменяется на `True`. Затем запускается цикл, в котором атрибут указывается в качестве условия. Как только значение атрибута станет равным значению `False`, цикл будет остановлен.

Внутри конструктора класса `MyWindow` производится создание надписи, двух кнопок и экземпляра класса `MyThread`. Далее назначаются обработчики сигналов. При нажатии кнопки

Запустить поток запустится метод `on_start()`, внутри которого с помощью метода `isRunning()` производится проверка текущего статуса потока. Если поток не запущен, выполняется его запуск вызовом метода `start()`. При нажатии кнопки **Остановить поток** запустится метод `on_stop()`, в котором атрибуту `running` присваивается значение `False`. Это значение является условием выхода из цикла внутри метода `run()`.

Путем изменения значения атрибута можно прервать выполнение цикла только в том случае, если закончилось выполнение очередной итерации. Если поток длительное время ожидает какого-либо события (например, ответа сервера), можно так и не дождаться завершения потока. Чтобы принудительно прервать выполнение потока, следует воспользоваться методом `terminate()`. Однако к этому методу рекомендуется прибегать только в крайнем случае, поскольку прерывание производится в любой части кода. При этом блокировки автоматически не снимаются, а кроме того, можно повредить данные, над которыми производились операции в момент прерывания. После вызова метода `terminate()` следует вызвать метод `wait()`.

При закрытии окна приложение завершает работу, что также приводит к завершению всех потоков. Чтобы предотвратить повреждение данных, следует перехватить событие закрытия окна и дождаться окончания выполнения. Чтобы перехватить событие, необходимо внутри класса создать метод с предопределенным названием, в нашем случае — с названием `closeEvent()`. Этот метод будет автоматически вызван при попытке закрыть окно. В качестве параметра метод принимает объект события `event`, через который можно получить дополнительную информацию о событии. Чтобы закрыть окно внутри метода `closeEvent()`, следует вызвать метод `accept()` объекта события. Если необходимо предотвратить закрытие окна, то следует вызвать метод `ignore()`.

Внутри метода `closeEvent()` мы присваиваем атрибуту `running` значение `False`. Далее с помощью метода `wait()` даем возможность потоку normally завершить работу. В качестве параметра метод `wait()` принимает количество миллисекунд, по истечении которых управление будет передано следующей инструкции. Необходимо учитывать, что это максимальное время: если поток закончит работу раньше, то и метод закончит выполнение раньше. Метод `wait()` возвращает значение `True`, если поток успешно завершил работу, и `False` — в противном случае. Ожидание завершения потока занимает некоторое время, в течение которого окно будет по-прежнему видимым. Чтобы не вводить пользователя в заблуждение, в самом начале метода `closeEvent()` мы скрываем окно вызовом метода `hide()`.

Каждый поток может иметь собственный цикл обработки сигналов, который запускается с помощью метода `exec_()`. В этом случае потоки могут обмениваться сигналами между собой. Чтобы прервать цикл, следует вызвать слот `quit()` или метод `exit([resultCode=0])`. Рассмотрим обмен сигналами между потоками на примере (листинг 17.15).

Листинг 17.15. Обмен сигналами между потоками

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

class Thread1(QtCore.QThread):
    s1 = QtCore.pyqtSignal(int)
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.count = 0
```

```
def run(self):
    self.exec_()          # Запускаем цикл обработки сигналов
def on_start(self):
    self.count += 1
    self.s1.emit(self.count)

class Thread2(QtCore.QThread):
    s2 = QtCore.pyqtSignal(str)
def __init__(self, parent=None):
    QtCore.QThread.__init__(self, parent)
def run(self):
    self.exec_()          # Запускаем цикл обработки сигналов
def on_change(self, i):
    i += 10
    self.s2.emit("%d" % i)
class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.label = QtWidgets.QLabel("Нажмите кнопку")
        self.label.setAlignment(QtCore.Qt.AlignHCenter)
        self.button = QtWidgets.QPushButton("Сгенерировать сигнал")
        self.vbox = QtWidgets.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.button)
        self.setLayout(self.vbox)
        self.thread1 = Thread1()
        self.thread2 = Thread2()
        self.thread1.start()
        self.thread2.start()
        self.button.clicked.connect(self.thread1.on_start)
        self.thread1.s1.connect(self.thread2.on_change)
        self.thread2.s2.connect(self.on_thread2_s2)
    def on_thread2_s2(self, s):
        self.label.setText(s)

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Обмен сигналами между потоками")
    window.resize(300, 70)
    window.show()
    sys.exit(app.exec_())
```

В этом примере мы создали классы Thread1, Thread2 и MyWindow. Первые два класса представляют собой потоки. Внутри них в методе `run()` вызывается метод `exec_()`, который запускает цикл обработки событий. В конструкторе класса MyWindow производится создание надписи, кнопки и экземпляров классов Thread1 и Thread2. Далее выполняется запуск сразу двух потоков.

В следующей инструкции сигнал нажатия кнопки соединяется с методом `on_start()` первого потока. Внутри этого метода производится какая-либо операция (в нашем случае — увеличение значения атрибута `count`), а затем с помощью метода `emit()` генерируется сигнал `s1`, и в параметре передается результат выполнения метода. Сигнал `s1` соединен с методом `on_change()` второго потока. Внутри этого метода также производится какая-либо операция, а затем генерируется сигнал `s2`, и передается результат выполнения метода. В свою очередь сигнал `s2` соединен со слотом `on_thread2_s2` объекта окна, который выводит в надпись значение, переданное с этим сигналом. Таким образом, при нажатии кнопки **Сгенерировать сигнал** вначале будет вызван метод `on_start()` из класса `Thread1`, затем метод `on_change()` из класса `Thread2`, а потом метод `on_thread2_s2` класса `MyWindow`, который выведет результат выполнения на экран.

17.9.3. Модуль `queue`: создание очереди заданий

В предыдущем разделе мы рассмотрели возможность обмена сигналами между потоками. Теперь предположим, что запущены десять потоков, которые ожидают задания в бесконечном цикле. Как передать задание одному потоку, а не всем сразу? И как определить, какому потоку передать задание? Можно, конечно, создать список в глобальном пространстве имен и добавлять задания в этот список, но в этом случае придется решать вопрос о совместном использовании одного ресурса сразу десятью потоками. Ведь если потоки будут получать задания одновременно, то одно задание могут получить сразу несколько потоков, и какому-либо потоку не хватит заданий, — возникнет исключительная ситуация. Попросту говоря, возникает ситуация, когда вы пытаетесь сесть на стул, а другой человек одновременно пытается вытащить его из-под вас. Думаете, что успеете сесть?

Модуль `queue`, входящий в состав стандартной библиотеки Python, позволяет решить эту проблему. Модуль содержит несколько классов, которые реализуют разного рода потоко-безопасные очереди. Опишем эти классы:

- ◆ `Queue` — очередь (первым пришел, первым вышел). Формат конструктора:

```
<Объект> = Queue([maxsize=0])
```

Пример:

```
>>> import queue
>>> q = queue.Queue()
>>> q.put_nowait("elem1")
>>> q.put_nowait("elem2")
>>> q.get_nowait()
'elem1'
>>> q.get_nowait()
'elem2'
```

- ◆ `LifoQueue` — стек (последним пришел, первым вышел). Формат конструктора:

```
<Объект> = LifoQueue([maxsize=0])
```

Пример:

```
>>> q = queue.LifoQueue()
>>> q.put_nowait("elem1")
>>> q.put_nowait("elem2")
>>> q.get_nowait()
'elem2'
```

```
>>> q.get_nowait()
'elem1'

◆ PriorityQueue — очередь с приоритетами. Элементы очереди должны быть кортежами, в которых первым элементом является число, означающее приоритет, а вторым — значение элемента. При получении значения возвращается элемент с наивысшим приоритетом (наименьшим значением в первом параметре кортежа). Формат конструктора класса:  
<Объект> = PriorityQueue([maxsize=0])
```

Пример:

```
>>> q = queue.PriorityQueue()
>>> q.put_nowait((10, "elem1"))
>>> q.put_nowait((3, "elem2"))
>>> q.put_nowait((12, "elem3"))
>>> q.get_nowait()
(3, 'elem2')
>>> q.get_nowait()
(10, 'elem1')
>>> q.get_nowait()
(12, 'elem3')
```

Параметр `maxsize` во всех трех случаях задает максимальное количество элементов, которое может содержать очередь. Если параметр равен нулю (значение по умолчанию) или отрицательному значению, то размер очереди не ограничен.

Эти классы поддерживают следующие методы:

- ◆ `put(<Элемент>[, block=True] [, timeout=None])` — добавляет элемент в очередь. Если в параметре `block` указано значение `True`, поток будет ожидать возможности добавления элемента, — при этом в параметре `timeout` можно указать максимальное время ожидания в секундах. Если элемент не удалось добавить, возбуждается исключение `queue.Full`. В случае передачи параметром `block` значения `False` очередь не будет ожидать, когда появится возможность добавить в нее новый элемент, и в случае невозможности сделать это возбудит исключение `queue.Full` немедленно;
- ◆ `put_nowait(<Элемент>)` — добавление элемента без ожидания. Эквивалентно:
`put(<Элемент>, False)`
- ◆ `get([block=True] [, timeout=None])` — возвращает элемент, при этом удаляя его из очереди. Если в параметре `block` указано значение `True`, поток будет ожидать возможности извлечения элемента, — при этом в параметре `timeout` можно указать максимальное время ожидания в секундах. Если элемент не удалось получить, возбуждается исключение `queue.Empty`. В случае передачи параметром `block` значения `False` очередь не будет ожидать, когда появится возможность извлечь из нее элемент, и в случае невозможности сделать это возбудит исключение `queue.Empty` немедленно;
- ◆ `get_nowait()` — извлечение элемента без ожидания. Эквивалентно вызову `get(False)`;
- ◆ `join()` — блокирует поток, пока не будут обработаны все задания в очереди. Другие потоки после обработки текущего задания должны вызывать метод `task_done()`. Как только все задания окажутся обработанными, поток будет разблокирован;
- ◆ `task_done()` — этот метод должны вызывать потоки после обработки задания;
- ◆ `qsize()` — возвращает приблизительное количество элементов в очереди. Так как доступ к очереди имеют сразу несколько потоков, доверять этому значению не следует — в любой момент времени количество элементов может измениться;

- ◆ `empty()` — возвращает `True`, если очередь пуста, и `False` — в противном случае;
- ◆ `full()` — возвращает `True`, если очередь содержит элементы, и `False` — в противном случае.

Рассмотрим использование очереди в многопоточном приложении на примере (листинг 17.16).

Листинг 17.16. Использование модуля `queue`

```
# -*- coding: utf-8 -*-
from PyQt5 import QtCore, QtWidgets
import queue
class MyThread(QtCore.QThread):
    task_done = QtCore.pyqtSignal(int, int, name = 'taskDone')
    def __init__(self, id, queue, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.id = id
        self.queue = queue
    def run(self):
        while True:
            task = self.queue.get()                      # Получаем задание
            self.sleep(5)                                # Имитируем обработку
            self.task_done.emit(task, self.id)           # Передаем данные обратно
            self.queue.task_done()
class MyWindow(QtWidgets.QPushButton):
    def __init__(self):
        QtWidgets.QPushButton.__init__(self)
        self.setText("Раздать задания")
        self.queue = queue.Queue()                   # Создаем очередь
        self.threads = []
        for i in range(1, 3):                      # Создаем потоки и запускаем
            thread = MyThread(i, self.queue)
            self.threads.append(thread)
            thread.task_done.connect(self.on_task_done, QtCore.Qt.QueuedConnection)
            thread.start()
        self.clicked.connect(self.on_add_task)
    def on_add_task(self):
        for i in range(0, 11):
            self.queue.put(i)                      # Добавляем задания в очередь
    def on_task_done(self, data, id):             # Выводим обработанные данные
        print(data, "- id =", id)
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Использование модуля queue")
    window.resize(300, 30)
    window.show()
    sys.exit(app.exec_())
```

В этом примере конструктор класса MyThread принимает уникальный идентификатор (`id`) и ссылку на очередь (`queue`), которые сохраняются в одноименных атрибутах класса. В методе `run()` внутри бесконечного цикла производится получение элемента из очереди с помощью метода `get()`. Если очередь пуста, поток будет ожидать, пока не появится хотя бы один элемент. Далее производится обработка задания (в нашем случае — просто задержка), а затем обработанные данные передаются главному потоку через сигнал `taskDone`, принимающий два целочисленных параметра. В следующей инструкции с помощью метода `task_done()` указывается, что задание было обработано.

Отметим, что здесь в вызове функции `pyqtSignal()` присутствует именованный параметр `name`:

```
task_done = QtCore.pyqtSignal(int, int, name = 'taskDone')
```

Он задает имя сигнала и может быть полезен в том случае, если это имя отличается от имени атрибута класса, соответствующего сигналу, — как в нашем случае, где имя сигнала `taskDone` отличается от имени атрибута `task_done`. После чего мы можем обращаться к сигналу как по имени соответствующего ему атрибута:

```
self.task_done.emit(task, self.id)
```

так и по имени, заданному в параметре `name` функции `pyqtSignal()`:

```
self.taskDone.emit(task, self.id)
```

Главный поток реализуется с помощью класса `MyWindow`. Обратите внимание, что наследуется класс `QPushButton` (кнопка), а не класс `QWidget`. Все визуальные компоненты являются наследниками класса `QWidget`, поэтому любой компонент, не имеющий родителя, обладает своим собственным окном. В нашем случае используется только кнопка, поэтому можно сразу наследовать класс `QPushButton`.

Внутри конструктора класса `MyWindow` с помощью метода `setText()` задается текст надписи на кнопке, затем создается экземпляр класса `Queue` и сохраняется в атрибуте `queue`. В следующем выражении производится создание списка, в котором будут храниться ссылки на объекты потоков. Сами объекты потоков (в нашем случае их два) создаются внутри цикла и добавляются в список. Внутри цикла производится также назначение обработчика сигнала `taskDone` и запуск потока с помощью метода `start()`. Далее назначается обработчик нажатия кнопки.

При нажатии кнопки **Раздать задания** вызывается метод `on_add_task()`, внутри которого производится добавление заданий в очередь. После этого потоки выходят из цикла ожидания, и каждый из них получает одно уникальное задание. После окончания обработки потоки генерируют сигнал `taskDone` и вызывают метод `task_done()`, информирующий об окончании обработки задания. Главный поток получает сигнал и вызывает метод `on_task_done()`, внутри которого через параметры будут доступны обработанные данные. Так как метод расположен в GUI-потоке, мы можем изменять свойства компонентов и, например, добавить результат в список или таблицу. В нашем же примере результат просто выводится в окно консоли (чтобы увидеть сообщения, следует сохранить файл с расширением `ru`, а не `rw`). После окончания обработки задания потоки снова получают задания. Если очередь окажется пуста, потоки перейдут в режим ожидания заданий.

17.9.4. Классы `QMutex` и `QMutexLocker`

Как вы уже знаете, совместное использование одного ресурса сразу несколькими потоками может привести к непредсказуемому поведению программы или даже аварийному ее

завершению. То есть, доступ к ресурсу в один момент времени должен иметь лишь один поток. Следовательно, внутри программы необходимо предусмотреть возможность блокировки ресурса одним потоком и ожидание его разблокировки другим потоком.

Реализовать блокировку ресурса в PyQt позволяют классы `QMutex` и `QMutexLocker` из модуля `QtCore`.

Конструктор класса `QMutex` создает так называемый *мьютекс* и имеет следующий формат:

```
<Объект> = QMutex([mode=QtCore.QMutex.NonRecursive])
```

Необязательный параметр `mode` может принимать значения `NonRecursive` (поток может запросить блокировку только единожды, а после снятия блокировка может быть запрошена снова, — значение по умолчанию) и `Recursive` (поток может запросить блокировку несколько раз, и чтобы полностью снять блокировку, следует вызвать метод `unlock()` соответствующее количество раз).

Класс `QMutex` поддерживает следующие методы:

- ◆ `lock()` — устанавливает блокировку. Если ресурс был заблокирован другим потоком, работа текущего потока приостанавливается до снятия блокировки;
- ◆ `tryLock([timeout=0])` — устанавливает блокировку. Если блокировка была успешно установлена, метод возвращает значение `True`, если ресурс заблокирован другим потоком — значение `False` без ожидания возможности установить блокировку. Максимальное время ожидания в миллисекундах можно указать в качестве необязательного параметра `timeout`. Если в параметре указано отрицательное значение, то метод `tryLock()` ведет себя аналогично методу `lock()`;
- ◆ `unlock()` — снимает блокировку;
- ◆ `isRecursive()` — возвращает `True`, если конструктору было передано значение `Recursive`.

Рассмотрим использование класса `QMutex` на примере (листинг 17.17).

Листинг 17.17. Использование класса `QMutex`

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

class MyThread(QtCore.QThread):
    x = 10                                # Атрибут класса
    mutex = QtCore.QMutex()                  # Мьютекс
    def __init__(self, id, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.id = id
    def run(self):
        self.change_x()
    def change_x(self):
        MyThread.mutex.lock()                # Блокируем
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 5
        self.sleep(2)
        print("x =", MyThread.x, "id =", self.id)
```

```

MyThread.x += 34
print("x =", MyThread.x, "id =", self.id)
MyThread.mutex.unlock()           # Снимаем блокировку

class MyWindow(QtWidgets.QPushButton):
    def __init__(self):
        QtWidgets.QPushButton.__init__(self)
        self.setText("Запустить")
        self.thread1 = MyThread(1)
        self.thread2 = MyThread(2)
        self.clicked.connect(self.on_start)
    def on_start(self):
        if not self.thread1.isRunning(): self.thread1.start()
        if not self.thread2.isRunning(): self.thread2.start()

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Использование класса QMutex")
    window.resize(300, 30)
    window.show()
    sys.exit(app.exec_())

```

В этом примере в классе MyThread мы создали атрибут x, который доступен всем экземплярам класса. Изменение значения атрибута в одном потоке повлечет изменение значения и в другом потоке. Если потоки будут изменять значение одновременно, то предсказать текущее значение атрибута становится невозможным. Следовательно, изменять значение можно только после установки блокировки.

Чтобы обеспечить блокировку, внутри класса MyThread создается экземпляр класса QMutex и сохраняется в атрибуте mutex. Обратите внимание, что сохранение производится в атрибуте объекта класса, а не в атрибуте экземпляра класса. Чтобы блокировка сработала, необходимо, чтобы защищаемый атрибут и мьютекс находились в одной области видимости. Далее весь код метода change_x(), в котором производится изменение атрибута x, помещается между вызовами методов lock() и unlock() мьютекса, — таким образом гарантируется, что он будет выполнен сначала одним потоком и только потом — другим.

Внутри конструктора класса MyWindow производится создание двух экземпляров класса MyThread и назначение обработчика нажатия кнопки. По нажатию кнопки Запустить будет вызван метод on_start(), внутри которого производится запуск сразу двух потоков одновременно, — при условии, что потоки не были запущены ранее. В результате мы получим в окне консоли следующий результат:

```

x = 10 id = 1
x = 15 id = 1
x = 49 id = 1
x = 49 id = 2
x = 54 id = 2
x = 88 id = 2

```

Как можно видеть, сначала изменение атрибута произвел поток с идентификатором 1, а лишь затем — поток с идентификатором 2. Если блокировку не указать, то результат будет иным:

```
x = 10 id = 1
x = 15 id = 2
x = 20 id = 1
x = 54 id = 1
x = 54 id = 2
x = 88 id = 2
```

В этом случае поток с идентификатором 2 изменил значение атрибута `x` до окончания выполнения метода `change_x()` в потоке с идентификатором 1.

При возникновении исключения внутри метода `change_x()` ресурс останется заблокированным, т. к. вызов метода `unlock()` не будет выполнен. Кроме того, можно по случайности забыть вызвать метод `unlock()`, что также приведет к вечной блокировке.

Исключить подобную ситуацию позволяет класс `QMutexLocker`. Конструктор этого класса принимает объект мьютекса и устанавливает блокировку. После выхода из области видимости будет вызван деструктор класса, внутри которого блокировка автоматически снимется. Следовательно, если создать экземпляр класса `QMutexLocker` в начале метода, то после выхода из метода блокировка будет снята. Переделаем метод `change_x()` из класса `MyThread` и используем класс `QMutexLocker` (листинг 17.18).

Листинг 17.18. Использование класса `QMutexLocker`

```
def change_x(self):
    ml = QtCore.QMutexLocker(MyThread.mutex)
    print("x =", MyThread.x, "id =", self.id)
    MyThread.x += 5
    self.sleep(2)
    print("x =", MyThread.x, "id =", self.id)
    MyThread.x += 34
    print("x =", MyThread.x, "id =", self.id)
    # Блокировка автоматически снимется
```

При использовании класса `QMutexLocker` следует помнить о разнице между областями видимости в языках C++ и Python. В языке C++ область видимости ограничена блоком, которым может являться как функция, так и просто область, ограниченная фигурными скобками. Таким образом, если переменная объявлена внутри блока условного оператора, например, `if`, то при выходе из этого блока переменная уже не будет видна:

```
if (условие) {
    int x = 10; // Объявляем переменную
    // ...
}
// Здесь переменная x уже не видна!
```

В языке Python область видимости гораздо шире. Если мы объявим переменную внутри условного оператора, то она будет видна и после выхода из этого блока:

```
if условие:
    x = 10      # Объявляем переменную
    #
# Здесь переменная x еще видна
```

Таким образом, область видимости локальной переменной в языке Python ограничена функцией, а не любым блоком.

Класс `QMutexLocker` поддерживает протокол менеджеров контекста, который позволяет ограничить область видимости блоком инструкции `with...as`. Этот протокол гарантирует снятие блокировки, даже если внутри инструкции `with...as` будет возбуждено исключение. Переделаем метод `change_x()` из класса `MyThread` снова и используем в этот раз инструкцию `with...as` (листинг 17.19).

Листинг 17.19. Использование инструкции `with...as`

```
def change_x(self):
    with QtCore.QMutexLocker(MyThread.mutex):
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 5
        self.sleep(2)
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 34
        print("x =", MyThread.x, "id =", self.id)
    # Блокировка автоматически снимется
```

Теперь, когда вы уже знаете о возможности блокировки ресурса, следует сделать несколько замечаний:

- ◆ установка и снятие блокировки занимают некоторый промежуток времени, тем самым снижая эффективность всей программы. Поэтому встроенные типы данных не обеспечивают безопасную работу в многопоточном приложении. И прежде чем использовать блокировки, подумайте — может быть, в вашем приложении они и не нужны;
- ◆ второе замечание относится к доступу к защищенному ресурсу из GUI-потока. Ожидание снятия блокировки может заблокировать GUI-поток, и приложение перестанет реагировать на события. Поэтому в таком случае следует использовать сигналы, а не прямой доступ;
- ◆ и последнее замечание относится к *взаимной блокировке*. Если первый поток, владея ресурсом A, захочет получить доступ к ресурсу B, а второй поток, владея ресурсом B, захочет получить доступ к ресурсу A, то оба потока будут ждать снятия блокировки вечно. В этой ситуации следует предусмотреть возможность временного освобождения ресурсов одним из потоков после превышения периода ожидания.

Класс `QMutexLocker` также поддерживает методы `unlock()` и `relock()`. Первый метод выполняет разблокировку мьютекса без уничтожения экземпляра класса `QMutexLocker`, а второй выполняет повторное наложение блокировки.

ПРИМЕЧАНИЕ

Для синхронизации и координации потоков предназначены также классы `QSemaphore` и `QWaitCondition`. За подробной информацией по этим классам обращайтесь к документации по PyQt. Следует также помнить, что в стандартную библиотеку языка Python входят модули `multiprocessing` и `threading`, которые позволяют работать с потоками в любом приложении. Однако при использовании PyQt нужно отдать предпочтение классу `QThread`, т. к. он позволяет работать с сигналами.

17.10. Вывод заставки

В больших приложениях загрузка начальных данных может занимать продолжительное время, в течение которого принято выводить окно-заставку, в котором отображается процесс загрузки. По окончании инициализации приложения окно-заставка скрывается и отображается главное окно.

Для вывода окна-заставки в PyQt предназначен класс `QSplashScreen` из модуля `QtWidgets`. Конструктор класса имеет следующие форматы:

```
<Объект> = QSplashScreen([<Изображение>] [, flags=<Тип окна>])
<Объект> = QSplashScreen(<Родитель> [, <Изображение>] [, flags=<Тип окна>])
```

Параметр `<Родитель>` позволяет указать ссылку на родительский компонент. В параметре `<Изображение>` указывается ссылка на изображение (экземпляр класса `QPixmap`, объявленного в модуле `QtGui`), которое будет отображаться на заставке. Конструктору класса `QPixmap` можно передать путь к файлу с изображением. Параметр `flags` предназначен для указания типа окна — например, чтобы заставка отображалась поверх всех остальных окон, следует передать флаг `WindowStaysOnTopHint`.

Класс `QSplashScreen` поддерживает следующие методы:

- ◆ `show()` — отображает заставку;
- ◆ `finish(<Ссылка на окно>)` — закрывает заставку. В качестве параметра указывается ссылка на главное окно приложения;
- ◆ `showMessage(<Сообщение>[, <Выравнивание>[, <Цвет>]])` — выводит сообщение. Во втором параметре указывается местоположение надписи в окне. По умолчанию надпись выводится в левом верхнем углу окна. В качестве значения можно через оператор `|` указать комбинацию следующих флагов: `AlignTop` (по верху), `AlignCenter` (по центру вертикали и горизонтали), `AlignBottom` (по низу), `AlignHCenter` (по центру горизонтали), `AlignVCenter` (по центру вертикали), `AlignLeft` (по левой стороне), `AlignRight` (по правой стороне). В третьем параметре указывается цвет текста. В качестве значения можно указать атрибут из класса `QtCore.Qt` (например, `black` (по умолчанию), `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.);
- ◆ `clearMessage()` — стирает надпись;
- ◆ `setPixmap(<Изображение>)` — позволяет изменить изображение в окне. В качестве параметра указывается экземпляр класса `QPixmap`;
- ◆ `pixmap()` — возвращает изображение в виде экземпляра класса `QPixmap`.

Пример кода, выводящего заставку, показан в листинге 17.20. А на рис. 17.6 можно увидеть эту заставку воочию.

Листинг 17.20. Вывод заставки

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

class MyWindow(QtWidgets.QPushButton):
    def __init__(self):
```

```
QtWidgets.QPushButton.__init__(self)
self.setText("Закрыть окно")
self.clicked.connect(QtWidgets.qApp.quit)
def load_data(self, sp):
    for i in range(1, 11):           # Инициируем процесс
        time.sleep(2)                # Что то загружаем
        sp.showMessage("Загрузка данных... {0}%".format(i * 10), %
                        QtCore.Qt.AlignHCenter | QtCore.Qt.AlignBottom, QtCore.Qt.black)
        QtWidgets.qApp.processEvents() # Запускаем оборот цикла
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    splash = QtWidgets.QSplashScreen(QtGui.QPixmap("splashscreen.jpg"))
    splash.showMessage("Загрузка данных... 0%", %
                        QtCore.Qt.AlignHCenter | QtCore.Qt.AlignBottom, QtCore.Qt.black)
    splash.show()                  # Отображаем заставку
    QtWidgets.qApp.processEvents() # Запускаем оборот цикла
    window = MyWindow()
    window.setWindowTitle("Использование класса QSplashScreen")
    window.resize(300, 30)
    window.load_data(splash)       # Загружаем данные
    window.show()
    splash.finish(window)         # Скрываем заставку
    sys.exit(app.exec_())
```



Рис. 17.6. Заставка, выводимая на экран кодом из листинга 17.20

17.11. Доступ к документации

Библиотека PyQt включает в себя несколько сотен классов. Понятно, что описать их все в одной книге не представляется возможным, поэтому мы рассмотрим здесь только наиболее часто используемые возможности библиотеки. А чтобы получить полную информацию, следует обратиться к документации.

Самая последняя версия документации в формате HTML доступна по интернет-адресу: <http://pyqt.sourceforge.net/Docs/PyQt5/>. Там рассматриваются основные вопросы PyQt-программирования: работа с сигналами, использование Qt Designer, отличия PyQt5 от PyQt 4 и пр.

ПРИМЕЧАНИЕ

Ранее эта документация поставлялась непосредственно в составе PyQt вместе с примерами программирования. Однако теперь, к сожалению, ни того, ни другого в поставке библиотеки нет.

В правом верхнем углу любой страницы документации находится гиперссылка **Classes**, ведущая на страницу со списком всех классов, что имеются в библиотеке. Название каждого класса является ссылкой на страницу с описанием этого класса, где имеется гиперссылка, в свою очередь ведущая на страницу сайта <http://doc.qt.io/> с полным описанием этого класса.

Также в правом верхнем углу любой страницы имеется гиперссылка **Modules**, ведущая на страницу со списком всех модулей, составляющих PyQt. Аналогично, каждое название модуля является гиперссылкой, ведущей на страницу со списком всех классов, которые определены в этом модуле. Названия классов также являются гиперссылками, ведущими на страницы с описаниями этих классов.

СОВЕТ

Увы, но от документации, опубликованной на сайте <http://pyqt.sourceforge.net/Docs/PyQt5/>, немного толку. Поэтому лучше сразу же обратиться к сайту <http://doc.qt.io/>, где приводится полное описание библиотеки Qt, правда, рассчитанное на разработчиков, которые программируют на C++.