



# ГЛАВА 18

## Управление окном приложения

Создать окно и управлять им позволяет класс `QWidget`. Он наследует классы `QObject` и `QPaintDevice` и, в свою очередь, является базовым классом для всех визуальных компонентов, поэтому любой компонент, не имеющий родителя, обладает своим собственным окном. В этой главе мы рассмотрим методы класса `QWidget` применительно к окну верхнего уровня, однако следует помнить, что те же самые методы можно применять и к любым компонентам. Так, метод, позволяющий управлять размерами окна, можно использовать и для изменения размеров компонента, имеющего родителя. Тем не менее, некоторые методы имеет смысл использовать только для окон верхнего уровня, — например, метод, позволяющий изменить текст в заголовке окна, не имеет смысла использовать в обычных компонентах.

Для создания окна верхнего уровня, помимо класса `QWidget`, можно использовать и другие классы, которые являются его наследниками, — например, `QFrame` (окно с рамкой) или `QDialog` (диалоговое окно). При использовании класса `QDialog` окно будет выравниваться по центру экрана или родительского окна и иметь в заголовке только две кнопки: **Справка** и **Закрыть**. Кроме того, можно использовать класс `QMainWindow`, который представляет главное окно приложения с меню, панелями инструментов и строкой состояния. Использование классов `QDialog` и `QMainWindow` имеет свои различия, которые мы рассмотрим в отдельных главах.

### 18.1. Создание и отображение окна

Самый простой способ создать пустое окно показан в листинге 18.1.

#### Листинг 18.1. Создание и отображение окна

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

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget() # Создаем окно
window.setWindowTitle("Заголовок окна") # Указываем заголовок
window.resize(300, 50) # Минимальные размеры
window.show() # Отображаем окно
sys.exit(app.exec_())
```

Конструктор класса `QWidget` имеет следующий формат:

```
<Объект> = QWidget ( [parent=<Родитель>] [, flags=<Тип окна>] )
```

В параметре `parent` указывается ссылка на родительский компонент. Если параметр не указан или имеет значение `None`, компонент будет обладать своим собственным окном. Если в параметре `flags` указан тип окна, то компонент, имея родителя, также будет обладать своим собственным окном, но окажется привязан к родителю. Это позволяет, например, создать модальное окно, которое станет блокировать только окно родителя, а не все окна приложения. Какие именно значения можно указать в параметре `flags`, мы рассмотрим в следующем разделе.

Указать ссылку на родительский компонент и, возможно, тип окна уже после создания объекта позволяет метод `setParent()`. Формат метода:

```
setParent (<Родитель> [, <Тип окна>])
```

Получить ссылку на родительский компонент можно с помощью метода `parentWidget()`. Если компонент не имеет родителя, возвращается значение `None`.

Для изменения текста в заголовке окна предназначен метод `setWindowTitle()`. Формат метода:

```
setWindowTitle(<Текст, отображаемый в заголовке>)
```

Метод `windowTitle()` позволяет получить текст, выводящийся в заголовке окна.

После создания окна необходимо вызвать метод `show()`, чтобы вывести окно на экран. Для скрытия окна предназначен метод `hide()`. Для отображения и скрытия компонентов можно также пользоваться методом `setVisible(<Флаг>)`. Если параметром этого метода передано значение `True`, компонент будет отображен, а если значение `False` — скрыт. Пример отображения окна:

```
window.setVisible(True)
```

Проверить, видим компонент в настоящее время или нет, позволяет метод `isVisible()`, который возвращает `True`, если компонент видим, и `False` — в противном случае. Кроме того, можно воспользоваться методом `isHidden()` — он возвращает `True`, если компонент скрыт, и `False` — в противном случае.

## 18.2. Указание типа окна

При использовании класса `QWidget` окно по умолчанию создается с заголовком, в котором расположены значок, при нажатии на который выводится оконное меню, текст заголовка и кнопки **Свернуть**, **Развернуть** и **Закрыть**. Указать другой тип создаваемого окна позволяет метод `setWindowFlags()` или параметр `flags` в конструкторе класса `QWidget`. Обратите внимание, что метод `setWindowFlags()` должен вызываться перед отображением окна. Формат метода:

```
setWindowFlags(<Тип окна>)
```

В параметре `<Тип окна>` можно указать следующие атрибуты из класса `QtCore.Qt`:

- ◆ `Widget` — тип по умолчанию для класса `QWidget`;
- ◆ `Window` — указывает, что компонент является окном, независимо от того, имеет он родителя или нет. Окно выводится с рамкой и заголовком, в котором расположены кнопки **Свернуть**, **Развернуть** и **Закрыть**. По умолчанию размеры окна можно изменять с помощью мыши;

- ◆ Dialog — диалоговое окно. Выводится с рамкой и заголовком, в котором расположены кнопки Справка и Закрыть. Размеры окна можно изменять с помощью мыши. Это значение по умолчанию для класса QDialog. Пример указания типа для диалогового окна:

```
window.setWindowFlags(QtCore.Qt.Dialog)
```
- ◆ Sheet и Drawer — окна в стиле Apple Macintosh;
- ◆ Popup — указывает, что окно представляет собой всплывающее меню. Оно выводится без рамки и заголовка и, кроме того, может отбрасывать тень. Изменить размеры окна с помощью мыши нельзя;
- ◆ Tool — сообщает, что окно представляет собой панель инструментов. Оно выводится с рамкой и заголовком (меньшим по высоте, чем обычное окно), в котором расположена кнопка Закрыть. Размеры окна можно изменять с помощью мыши;
- ◆ ToolTip — указывает, что окно представляет собой всплывающую подсказку. Оно выводится без рамки и заголовка. Изменить размеры окна с помощью мыши нельзя;
- ◆ SplashScreen — сообщает, что окно представляет собой заставку. Оно выводится без рамки и заголовка. Изменить размеры окна с помощью мыши нельзя. Это значение по умолчанию для класса QSplashScreen;
- ◆ Desktop — указывает, что окно представляет собой рабочий стол. Оно вообще не отображается на экране;
- ◆ SubWindow — сообщает, что окно представляет собой дочерний компонент, независимо от того, имеет он родителя или нет. Выводится оно с рамкой и заголовком (меньшим по высоте, чем у обычного окна), но без кнопок. Изменить размеры окна с помощью мыши нельзя;
- ◆ ForeignWindow — указывает, что окно создано другим процессом;
- ◆ CoverWindow — окно, представляющее минимизированное приложение на некоторых мобильных платформах.

Получить тип окна в программе позволяет метод `windowType()`.

Для окон верхнего уровня можно через оператор `|` дополнительно указать следующие атрибуты из класса `QtCore.Qt` (здесь упомянуты только наиболее часто используемые атрибуты, полный их список ищите в документации):

- ◆ MSWindowsFixedSizeDialogHint — запрещает изменение размеров окна. Кнопка Развернуть в заголовке окна становится неактивной;
- ◆ FramelessWindowHint — убирает рамку и заголовок окна. Изменять размеры окна и перемещать его нельзя;
- ◆ NoDropShadowWindowHint — убирает отбрасываемую окном тень;
- ◆ CustomizeWindowHint — убирает рамку и заголовок окна, но добавляет эффект объемности. Размеры окна можно изменять;
- ◆ WindowTitleHint — добавляет заголовок окна. Выведем для примера окно фиксированного размера с заголовком, в котором находится только текст:

```
window.setWindowFlags(QtCore.Qt.Window |  
                      QtCore.Qt.FramelessWindowHint |  
                      QtCore.Qt.WindowTitleHint)
```
- ◆ WindowSystemMenuHint — добавляет оконное меню и кнопку Закрыть;

- ◆ WindowMinimizeButtonHint — добавляет в заголовок кнопку **Свернуть**;
- ◆ WindowMaximizeButtonHint — добавляет в заголовок кнопку **Развернуть**;
- ◆ WindowMinMaxButtonsHint — добавляет в заголовок кнопки **Свернуть** и **Развернуть**;
- ◆ WindowCloseButtonHint — добавляет кнопку **Закрыть**;
- ◆ WindowContextHelpButtonHint — добавляет кнопку **Справка**;
- ◆ WindowStaysOnTopHint — сообщает системе, что окно всегда должно отображаться поверх всех других окон;
- ◆ WindowStaysOnBottomHint — сообщает системе, что окно всегда должно быть расположено позади всех других окон.

Получить все установленные флаги из программы позволяет метод `windowFlags()`.

### 18.3. Изменение и получение размеров окна

Для изменения размеров окна предназначены следующие методы:

- ◆ `resize(<Ширина>, <Высота>)` — изменяет текущий размер окна. Если содержимое окна не помещается в установленный размер, то размер будет выбран так, чтобы компоненты поместились без искажения при условии, что используются менеджеры геометрии. Следовательно, заданный размер может не соответствовать реальному размеру окна. Если используется абсолютное позиционирование, компоненты могут оказаться наполовину или полностью за пределами видимой части окна. В качестве параметра можно также указать экземпляр класса `QSize` из модуля `QtCore`:

```
window.resize(100, 70)  
window.resize(QtCore.QSize(100, 70))
```

- ◆ `setGeometry(<X>, <Y>, <Ширина>, <Высота>)` — изменяет одновременно положение компонента и его текущий размер. Первые два параметра задают координаты левого верхнего угла (относительно родительского компонента), а третий и четвертый параметры — ширину и высоту. В качестве параметра можно также указать экземпляр класса `QRect` из модуля `QtCore`:

```
window.setGeometry(100, 100, 100, 70)  
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
```

- ◆ `setFixedSize(<Ширина>, <Высота>)` — задает фиксированный размер. Изменить размеры окна с помощью мыши нельзя. Кнопка **Развернуть** в заголовке окна становится неактивной. В качестве параметра можно также указать экземпляр класса `QSize`:

```
window.setFixedSize(100, 70)  
window.setFixedSize(QtCore.QSize(100, 70))
```

- ◆ `setFixedWidth(<Ширина>)` — задает фиксированный размер только по ширине. Изменить ширину окна с помощью мыши нельзя;

- ◆ `setFixedHeight(<Высота>)` — задает фиксированный размер только по высоте. Изменить высоту окна с помощью мыши нельзя;

- ◆ `setMinimumSize(<Ширина>, <Высота>)` — задает минимальные размеры окна. В качестве параметра можно также указать экземпляр класса `QSize`:

```
window.setMinimumSize(100, 70)  
window.setMinimumSize(QtCore.QSize(100, 70))
```

- ◆ `setMinimumWidth(<Ширина>)` и `setMinimumHeight(<Высота>)` — задают минимальную ширину и высоту соответственно;
- ◆ `setMaximumSize(<Ширина>, <Высота>)` — задает максимальный размер окна. В качестве параметра можно также указать экземпляр класса `QSize`:

```
window.setMaximumSize(100, 70)
window.setMaximumSize(QtCore.QSize(100, 70))
```
- ◆ `setMaximumWidth(<Ширина>)` и `setMaximumHeight(<Высота>)` — задают максимальную ширину и высоту соответственно;
- ◆ `setBaseSize(<Ширина>, <Высота>)` — задает базовые размеры. В качестве параметра можно также указать экземпляр класса `QSize`:

```
window.setBaseSize(500, 500)
window.setBaseSize(QtCore.QSize(500, 500))
```
- ◆ `adjustSize()` — подгоняет размеры компонента под содержимое. При этом учитываются рекомендуемые размеры, возвращаемые методом `sizeHint()`.

Получить размеры позволяют следующие методы:

- ◆ `width()` и `height()` — возвращают текущую ширину и высоту соответственно:

```
window.resize(50, 70)
print(window.width(), window.height()) # 50 70
```
- ◆ `size()` — возвращает экземпляр класса `QSize`, содержащий текущие размеры:

```
window.resize(50, 70)
print(window.size().width(), window.size().height()) # 50 70
```
- ◆ `minimumSize()` — возвращает экземпляр класса `QSize`, содержащий минимальные размеры;
- ◆ `minimumWidth()` и `minimumHeight()` — возвращают минимальную ширину и высоту соответственно;
- ◆ `maximumSize()` — возвращает экземпляр класса `QSize`, содержащий максимальные размеры;
- ◆ `maximumWidth()` и `maximumHeight()` — возвращают максимальную ширину и высоту соответственно;
- ◆ `baseSize()` — возвращает экземпляр класса `QSize`, содержащий базовые размеры;
- ◆ `sizeHint()` — возвращает экземпляр класса `QSize`, содержащий рекомендуемые размеры компонента. Если таковые являются отрицательными, считается, что нет рекомендуемого размера;
- ◆ `minimumSizeHint()` — возвращает экземпляр класса `QSize`, содержащий рекомендуемый минимальный размер компонента. Если возвращаемые размеры являются отрицательными, то считается, что нет рекомендуемого минимального размера;
- ◆ `rect()` — возвращает экземпляр класса `QRect`, содержащий координаты и размеры прямоугольника, в который вписан компонент:

```
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
rect = window.rect()
print(rect.left(), rect.top()) # 0 0
print(rect.width(), rect.height()) # 100 70
```
- ◆ `geometry()` — возвращает экземпляр класса `QRect`, содержащий координаты относительно родительского компонента:

```
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
rect = window.geometry()
print(rect.left(), rect.top())      # 100 100
print(rect.width(), rect.height()) # 100 70
```

При изменении и получении размеров окна следует учитывать, что:

- ◆ размеры не включают высоту заголовка окна и ширину границ;
- ◆ размер компонентов может изменяться в зависимости от настроек стиля. Например, на разных компьютерах может быть задан шрифт разного наименования и размера, поэтому от указания фиксированных размеров лучше отказаться;
- ◆ размер окна может изменяться в промежутке между получением значения и действиями, выполняющими обработку этих значений в программе. Например, сразу после получения размера пользователь может изменить размеры окна с помощью мыши.

Чтобы получить размеры окна, включающие высоту заголовка и ширину границ, следует воспользоваться методом `frameSize()`, который возвращает экземпляр класса `QSize`. Обратите внимание, что полные размеры окна доступны только после его отображения, — до этого момента они совпадают с размерами клиентской области окна, без учета высоты заголовка и ширины границ. Пример получения полного размера окна:

```
window.resize(200, 70)                      # Задаем размеры
# ...
window.show()                                # Отображаем окно
print(window.width(), window.height())        # 200 70
print(window.frameSize().width(),
      window.frameSize().height())            # 208 104
```

Чтобы получить координаты окна с учетом высоты заголовка и ширины границ, следует воспользоваться методом `frameGeometry()`. И в этом случае полные размеры окна доступны только после отображения окна. Метод возвращает экземпляр класса `QRect`:

```
window.setGeometry(100, 100, 200, 70)
# ...
window.show()                                # Отображаем окно
rect = window.geometry()
print(rect.left(), rect.top())                # 100 100
print(rect.width(), rect.height())            # 200 70
rect = window.frameGeometry()
print(rect.left(), rect.top())                # 96 70
print(rect.width(), rect.height())            # 208 104
```

## 18.4. Местоположение окна на экране и управление им

Задать местоположение окна на экране монитора позволяют следующие методы:

- ◆ `move(<X>, <Y>)` — задает положение компонента относительно родителя с учетом высоты заголовка и ширины границ. В качестве параметра можно также указать экземпляр класса `QPoint` из модуля `QtCore`.

Пример вывода окна в левом верхнем углу экрана:

```
window.move(0, 0)
window.move(QtCore.QPoint(0, 0))
```

- ◆ `setGeometry(<X>, <Y>, <Ширина>, <Высота>)` — изменяет одновременно положение компонента и его текущие размеры. Первые два параметра задают координаты левого верхнего угла относительно родительского компонента, а третий и четвертый параметры — ширину и высоту. Обратите внимание, что метод не учитывает высоту заголовка и ширину границ, поэтому, если указать координаты `(0, 0)`, заголовок окна и левая граница окна окажутся за пределами экрана. В качестве параметра можно также задать экземпляр класса `QRect` из модуля `QtCore`:

```
window.setGeometry(100, 100, 100, 70)
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
```

### **Внимание!**

Начало координат расположено в левом верхнем углу. Положительная ось X направлена вправо, а положительная ось Y — вниз.

Получить позицию окна позволяют следующие методы:

- ◆ `x()` и `y()` — возвращают координаты левого верхнего угла окна относительно родителя по осям X и Y соответственно. Методы учитывают высоту заголовка и ширину границ:

```
window.move(10, 10)
print(window.x(), window.y()) # 10 10
```

- ◆ `pos()` — возвращает экземпляр класса `QPoint`, содержащий координаты левого верхнего угла окна относительно родителя. Метод учитывает высоту заголовка и ширину границ:

```
window.move(10, 10)
print(window.pos().x(), window.pos().y()) # 10 10
```

- ◆ `geometry()` — возвращает экземпляр класса `QRect`, содержащий координаты относительно родительского компонента. Метод не учитывает высоту заголовка и ширину границ:

```
window.resize(300, 100)
window.move(10, 10)
rect = window.geometry()
print(rect.left(), rect.top()) # 14 40
print(rect.width(), rect.height()) # 300 100
```

- ◆ `frameGeometry()` — возвращает экземпляр класса `QRect`, содержащий координаты с учетом высоты заголовка и ширины границ. Полные размеры окна доступны только после отображения окна:

```
window.resize(300, 100)
window.move(10, 10)
rect = window.frameGeometry()
print(rect.left(), rect.top()) # 10 10
print(rect.width(), rect.height()) # 308 134
```

Для отображения окна по центру экрана, у правой или нижней его границы необходимо знать размеры экрана. Для получения размеров экрана вначале следует вызвать статический метод `QApplication.desktop()`, который возвращает ссылку на компонент рабочего стола, представленный экземпляром класса `QDesktopWidget` из модуля `QtWidgets`. Получить размеры экрана позволяют следующие методы этого класса:

- ◆ `width()` — возвращает ширину всего экрана в пикселях;
- ◆ `height()` — возвращает высоту всего экрана в пикселях.

Примеры:

```
desktop = QtGui.QApplication.desktop()
print(desktop.width(), desktop.height()) # 1440 900
```

- ◆ `screenGeometry()` — возвращает экземпляр класса `QRect`, содержащий координаты всего экрана:

```
desktop = QtGui.QApplication.desktop()
rect = desktop.screenGeometry()
print(rect.left(), rect.top()) # 0 0
print(rect.width(), rect.height()) # 1440 900
```

- ◆ `availableGeometry()` — возвращает экземпляр класса `QRect`, содержащий координаты только доступной части экрана (без размера панели задач):

```
desktop = QtGui.QApplication.desktop()
rect = desktop.availableGeometry()
print(rect.left(), rect.top()) # 0 0
print(rect.width(), rect.height()) # 1440 818
```

Пример отображения окна приблизительно по центру экрана показан в листинге 18.2.

#### Листинг 18.2. Вывод окна приблизительно по центру экрана

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

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Вывод окна по центру экрана")
window.resize(300, 100)
desktop = QtWidgets.QApplication.desktop()
x = (desktop.width() - window.width()) // 2
y = (desktop.height() - window.height()) // 2
window.move(x, y)
window.show()
sys.exit(app.exec_())
```

В этом примере мы воспользовались методами `width()` и `height()`, которые не учитывают высоту заголовка и ширину границ. В большинстве случаев этого способа достаточно. Если же при выравнивании необходима точность, то для получения размеров окна можно воспользоваться методом `frameSize()`. Однако этот метод возвращает корректные значения лишь после отображения окна. Если код выравнивания по центру расположить после вызова метода `show()`, окно вначале отобразится в одном месте экрана, а затем переместится в центр, что вызовет неприятное мелькание. Чтобы исключить такое мелькание, следует вначале отобразить окно за пределами экрана, а затем переместить его в центр экрана (листинг 18.3).

#### Листинг 18.3. Вывод окна точно по центру экрана

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

```
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Вывод окна по центру экрана")
window.resize(300, 100)
window.move(window.width() * -2, 0)
window.show()
desktop = QtWidgets.QApplication.desktop()
x = (desktop.width() - window.frameSize().width()) // 2
y = (desktop.height() - window.frameSize().height()) // 2
window.move(x, y)
sys.exit(app.exec_())
```

Этот способ можно также использовать для выравнивания окна по правому краю экрана. Например, чтобы расположить окно в правом верхнем углу экрана, необходимо заменить код из предыдущего примера, выравнивающий окно по центру, следующим кодом:

```
desktop = QtWidgets.QApplication.desktop()
x = desktop.width() - window.frameSize().width()
window.move(x, 0)
```

Если попробовать вывести окно в правом нижнем углу, может возникнуть проблема, поскольку в операционной системе Windows в нижней части экрана обычно располагается панель задач, и окно частично окажется под ней. Здесь нам пригодится метод `availableGeometry()`, позволяющий получить высоту панели задач, расположенной в нижней части экрана, следующим образом:

```
desktop = QtWidgets.QApplication.desktop()
taskBarHeight = (desktop.screenGeometry().height() -
                 desktop.availableGeometry().height())
```

Следует также заметить, что в некоторых операционных системах панель задач допускается прикреплять к любой стороне экрана. Кроме того, экран может быть разделен на несколько рабочих столов. Все это необходимо учитывать при размещении окна (за более подробной информацией обращайтесь к документации по классу `QDesktopWidget`).

## 18.5. Указание координат и размеров

В двух предыдущих разделах были упомянуты классы `QPoint`, `QSize` и `QRect`. Класс `QPoint` описывает координаты точки, класс `QSize` — размеры, а класс `QRect` — координаты и размеры прямоугольной области. Все эти классы определены в модуле `QtCore`. Рассмотрим их более подробно.

### ПРИМЕЧАНИЕ

Классы `QPoint`, `QSize` и `QRect` предназначены для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать классы `QPointF`, `QSizeF` и `QRectF` соответственно. Эти классы также определены в модуле `QtCore`.

### 18.5.1. Класс `QPoint`: координаты точки

Класс `QPoint` описывает координаты точки. Для создания экземпляра класса предназначены следующие форматы конструкторов:

```
<Объект> = QPoint()
<Объект> = QPoint(<X>, <Y>)
<Объект> = QPoint(<QPoint>)
```

Первый конструктор создает экземпляр класса с нулевыми координатами:

```
>>> from PyQt5 import QtCore
>>> p = QtCore.QPoint()
>>> p.x(), p.y()
(0, 0)
```

Второй конструктор позволяет явно указать координаты точки:

```
>>> p = QtCore.QPoint(10, 88)
>>> p.x(), p.y()
(10, 88)
```

Третий конструктор создает новый экземпляр на основе другого экземпляра:

```
>>> p = QtCore.QPoint(QtCore.QPoint(10, 88))
>>> p.x(), p.y()
(10, 88)
```

Через экземпляр класса доступны следующие методы:

- ◆ `x()` и `y()` — возвращают координаты по осям X и Y соответственно;
- ◆ `setX(<X>)` и `setY(<Y>)` — задают координаты по осям X и Y соответственно;
- ◆ `isNull()` — возвращает `True`, если координаты равны нулю, и `False` — в противном случае:

```
>>> p = QtCore.QPoint()
>>> p.isNull()
True
>>> p.setX(10); p.setY(88)
>>> p.x(), p.y()
(10, 88)
```

- ◆ `manhattanLength()` — возвращает сумму абсолютных значений координат:

```
>>> QtCore.QPoint(10, 88).manhattanLength()
98
```

Над двумя экземплярами класса `QPoint` можно выполнять операции `+`, `+=`, `-` (минус), `-=`, `==` и `!=`. Для смены знака координат можно воспользоваться унарным оператором `-`. Кроме того, экземпляр класса `QPoint` можно умножить или разделить на вещественное число (операторами `*`, `*=`, `/` и `/=`):

```
>>> p1 = QtCore.QPoint(10, 20); p2 = QtCore.QPoint(5, 9)
>>> p1 + p2, p1 - p2
(PyQt5.QtCore.QPoint(15, 29), PyQt5.QtCore.QPoint(5, 11))
>>> p1 * 2.5, p1 / 2.0
(PyQt5.QtCore.QPoint(25, 50), PyQt5.QtCore.QPoint(5, 10))
>>> -p1, p1 == p2, p1 != p2
(PyQt5.QtCore.QPoint(-10, -20), False, True)
```

## 18.5.2. Класс QSize: размеры прямоугольной области

Класс QSize описывает размеры прямоугольной области. Для создания экземпляра класса предназначены следующие форматы конструкторов:

```
<Объект> = QSize()
<Объект> = QSize(<Ширина>, <Высота>)
<Объект> = QSize(<QSize>)
```

Первый конструктор создает экземпляр класса с отрицательной шириной и высотой. Второй конструктор позволяет явно указать ширину и высоту. Третий конструктор создает новый экземпляр на основе другого экземпляра:

```
>>> from PyQt5 import QtCore
>>> s1=QtCore.QSize(); s2=QtCore.QSize(10, 55); s3=QtCore.QSize(s2)
>>> s1
PyQt5.QtCore.QSize(-1, -1)
>>> s2, s3
(PyQt5.QtCore.QSize(10, 55), PyQt5.QtCore.QSize(10, 55))
```

Через экземпляр класса доступны следующие методы:

- ◆ `width()` и `height()` — возвращают ширину и высоту соответственно;
- ◆ `setWidth(<Ширина>)` и `setHeight(<Высота>)` — задают ширину и высоту соответственно.

Примеры:

```
>>> s = QtCore.QSize()
>>> s.setWidth(10); s.setHeight(55)
>>> s.width(), s.height()
(10, 55)
```

- ◆ `isNull()` — возвращает `True`, если ширина и высота равны нулю, и `False` — в противном случае;
- ◆ `isValid()` — возвращает `True`, если ширина и высота больше или равны нулю, и `False` — в противном случае;
- ◆ `isEmpty()` — возвращает `True`, если один параметр (ширина или высота) меньше или равен нулю, и `False` — в противном случае;
- ◆ `scale(<QSize>, <Тип преобразования>)` — производит изменение размеров области в соответствии со значением параметра `<Тип преобразования>`. Метод изменяет текущий объект и ничего не возвращает.

Форматы метода:

```
scale(<QSize>, <Тип преобразования>)
scale(<Ширина>, <Высота>, <Тип преобразования>)
```

В параметре `<Тип преобразования>` могут быть указаны следующие атрибуты из класса `QtCore.Qt`:

- `IgnoreAspectRatio` — 0 — свободно изменяет размеры без сохранения пропорций сторон;
- `KeepAspectRatio` — 1 — производится попытка масштабирования старой области внутри новой области без нарушения пропорций;
- `KeepAspectRatioByExpanding` — 2 — производится попытка полностью заполнить новую область без нарушения пропорций старой области.

Если новая ширина или высота имеет значение 0, размеры изменяются непосредственно без сохранения пропорций, вне зависимости от значения параметра <Тип преобразования>.

Примеры:

```
>>> s = QtCore.QSize(50, 20)
>>> s.scale(70, 60, QtCore.Qt.IgnoreAspectRatio); s
PyQt5.QtCore.QSize(70, 60)
>>> s = QtCore.QSize(50, 20)
>>> s.scale(70, 60, QtCore.Qt.KeepAspectRatio); s
PyQt5.QtCore.QSize(70, 28)
>>> s = QtCore.QSize(50, 20)
>>> s.scale(70, 60, QtCore.Qt.KeepAspectRatioByExpanding); s
PyQt5.QtCore.QSize(150, 60)
```

◆ `scaled()` — то же самое, что `scale()`, но не изменяет сам объект, а возвращает новый экземпляр класса `QSize`, хранящий измененные размеры:

```
>>> s1 = QtCore.QSize(50, 20)
>>> s2 = s1.scaled(70, 60, QtCore.Qt.IgnoreAspectRatio)
>>> s1, s2
(PyQt5.QtCore.QSize(50, 20), PyQt5.QtCore.QSize(70, 60))
```

◆ `boundedTo(<QSize>)` — возвращает экземпляр класса `QSize`, который содержит минимальную ширину и высоту из текущих размеров и размеров, указанных в параметре:

```
>>> s = QtCore.QSize(50, 20)
>>> s.boundedTo(QtCore.QSize(400, 5))
PyQt5.QtCore.QSize(50, 5)
>>> s.boundedTo(QtCore.QSize(40, 50))
PyQt5.QtCore.QSize(40, 20)
```

◆ `expandedTo(<QSize>)` — возвращает экземпляр класса `QSize`, который содержит максимальную ширину и высоту из текущих размеров и размеров, указанных в параметре:

```
>>> s = QtCore.QSize(50, 20)
>>> s.expandedTo(QtCore.QSize(400, 5))
PyQt5.QtCore.QSize(400, 20)
>>> s.expandedTo(QtCore.QSize(40, 50))
PyQt5.QtCore.QSize(50, 50)
```

◆ `transpose()` — меняет значения местами. Метод изменяет текущий объект и ничего не возвращает:

```
>>> s = QtCore.QSize(50, 20)
>>> s.transpose(); s
PyQt5.QtCore.QSize(20, 50)
```

◆ `transposed()` — то же самое, что `transpose()`, но не изменяет сам объект, а возвращает новый экземпляр класса `QSize` с измененными размерами:

```
>>> s1 = QtCore.QSize(50, 20)
>>> s2 = s1.transposed()
>>> s1, s2
(PyQt5.QtCore.QSize(50, 20), PyQt5.QtCore.QSize(20, 50))
```

Над двумя экземплярами класса `QSize` можно выполнять операции `+`, `+=`, `-` (минус), `-=`, `==` и `!=`. Кроме того, экземпляр класса `QSize` можно умножить или разделить на вещественное число (операторами `*`, `*=`, `/` и `/=`):

```
>>> s1 = QtCore.QSize(50, 20); s2 = QtCore.QSize(10, 5)
>>> s1 + s2, s1 - s2
(PyQt5.QtCore.QSize(60, 25), PyQt5.QtCore.QSize(40, 15))
>>> s1 * 2.5, s1 / 2
(PyQt5.QtCore.QSize(125, 50), PyQt5.QtCore.QSize(25, 10))
>>> s1 == s2, s1 != s2
(False, True)
```

### 18.5.3. Класс `QRect`: координаты и размеры прямоугольной области

Класс `QRect` описывает координаты и размеры прямоугольной области. Для создания экземпляра класса предназначены следующие форматы конструктора:

```
<Объект> = QRect()
<Объект> = QRect(<left>, <top>, <Ширина>, <Высота>)
<Объект> = QRect(<Координаты левого верхнего угла>, <Размеры>)
<Объект> = QRect(<Координаты левого верхнего угла>,
                  <Координаты правого нижнего угла>)
<Объект> = QRect(<QRect>)
```

Первый конструктор создает экземпляр класса со значениями по умолчанию. Второй и третий конструкторы позволяют указать координаты левого верхнего угла и размеры области. Во втором конструкторе значения указываются отдельно. В третьем конструкторе координаты задаются с помощью класса `QPoint`, а размеры — с помощью класса `QSize`. Четвертый конструктор позволяет указать координаты левого верхнего угла и правого нижнего угла. В качестве значений указываются экземпляры класса `QPoint`. Пятый конструктор создает новый экземпляр на основе другого экземпляра.

Примеры:

```
>>> from PyQt5 import QtCore
>>> r = QtCore.QRect()
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
(0, 0, -1, -1, 0, 0)
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
(10, 15, 409, 314, 400, 300)
>>> r = QtCore.QRect(QtCore.QPoint(10, 15), QtCore.QSize(400, 300))
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
(10, 15, 409, 314, 400, 300)
>>> r = QtCore.QRect(QtCore.QPoint(10, 15), QtCore.QPoint(409, 314))
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
(10, 15, 409, 314, 400, 300)
>>> QtCore.QRect(r)
PyQt5.QtCore.QRect(10, 15, 400, 300)
```

Изменить значения уже после создания экземпляра позволяют следующие методы:

- ◆ `setLeft(<X1>)`, `setX(<X1>)`, `setTop(<Y1>)` и `setY(<Y1>)` — задают координаты левого верхнего угла по осям X и Y:

```
>>> r = QtCore.QRect()
>>> r.setLeft(10); r.setTop(55); r
PyQt5.QtCore.QRect(10, 55, -10, -55)
>>> r.setX(12); r.setY(81); r
PyQt5.QtCore.QRect(12, 81, -12, -81)
```

- ◆ `setRight(<X2>)` и `setBottom(<Y2>)` — задают координаты правого нижнего угла по осям X и Y:
- >>> r = QtCore.QRect()  
>>> r.setRight(12); r.setBottom(81); r  
PyQt5.QtCore.QRect(0, 0, 13, 82)
- ◆ `setTopLeft(<QPoint>)` — задает координаты левого верхнего угла;
- ◆ `setTopRight(<QPoint>)` — задает координаты правого верхнего угла;
- ◆ `setBottomLeft(<QPoint>)` — задает координаты левого нижнего угла;
- ◆ `setBottomRight(<QPoint>)` — задает координаты правого нижнего угла.

Примеры:

```
>>> r = QtCore.QRect()
>>> r.setTopLeft(QtCore.QPoint(10, 5))
>>> r.setBottomRight(QtCore.QPoint(39, 19)); r
PyQt5.QtCore.QRect(10, 5, 30, 15)
>>> r.setTopRight(QtCore.QPoint(39, 5))
>>> r.setBottomLeft(QtCore.QPoint(10, 19)); r
PyQt5.QtCore.QRect(10, 5, 30, 15)
```

- ◆ `setWidth(<Ширина>)`, `setHeight(<Высота>)` и `setSize(<QSize>)` — задают ширину и высоту области;
- ◆ `setRect(<X1>, <Y1>, <Ширина>, <Высота>)` — задает координаты левого верхнего угла и размеры области;
- ◆ `setCoords(<X1>, <Y1>, <X2>, <Y2>)` — задает координаты левого верхнего и правого нижнего углов.

Примеры:

```
>>> r = QtCore.QRect()
>>> r.setRect(10, 10, 100, 500); r
PyQt5.QtCore.QRect(10, 10, 100, 500)
>>> r.setCoords(10, 10, 109, 509); r
PyQt5.QtCore.QRect(10, 10, 100, 500)
```

- ◆ `marginsAdded(<QMargins>)` — возвращает новый экземпляр класса `QRect`, который представляет текущую область, увеличенную на заданные величины границ. Эти границы указываются в виде экземпляра класса `QMargins` из модуля `QtCore`, конструктор которого имеет следующий формат:

`QMargins(<Граница слева>, <Граница сверху>, <Граница справа>, <Граница снизу>)`

Текущая область при этом не изменяется:

```
>>> r1 = QtCore.QRect(10, 15, 400, 300)
>>> m = QtCore.QMargins(5, 2, 5, 2)
```

```
>>> r2 = r1.marginsAdded(m)
>>> r2
PyQt5.QtCore.QRect(5, 13, 410, 304)
>>> r1
PyQt5.QtCore.QRect(10, 15, 400, 300)
```

- ◆ `marginsRemoved()` — то же самое, что `marginsAdded()`, но уменьшает новую область на заданные величины границ:

```
>>> r1 = QtCore.QRect(10, 15, 400, 300)
>>> m = QtCore.QMargins(2, 10, 2, 10)
>>> r2 = r1.marginsRemoved(m)
>>> r2
PyQt5.QtCore.QRect(12, 25, 396, 280)
>>> r1
PyQt5.QtCore.QRect(10, 15, 400, 300)
```

Переместить область при изменении координат позволяют следующие методы:

- ◆ `moveTo(<X1>, <Y1>)`, `moveTo(<QPoint>)`, `moveLeft(<X1>)` и `moveTop(<Y1>)` — задают новые координаты левого верхнего угла:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.moveTo(0, 0); r
PyQt5.QtCore.QRect(0, 0, 400, 300)
>>> r.moveTo(QtCore.QPoint(10, 10)); r
PyQt5.QtCore.QRect(10, 10, 400, 300)
>>> r.moveLeft(5); r.moveTop(0); r
PyQt5.QtCore.QRect(5, 0, 400, 300)
```

- ◆ `moveRight(<X2>)` и `moveBottom(<Y2>)` — задают новые координаты правого нижнего угла;

- ◆ `moveTopLeft(<QPoint>)` — задает новые координаты левого верхнего угла;

- ◆ `moveTopRight(<QPoint>)` — задает новые координаты правого верхнего угла;

- ◆ `moveBottomLeft(<QPoint>)` — задает новые координаты левого нижнего угла;

- ◆ `moveBottomRight(<QPoint>)` — задает новые координаты правого нижнего угла.

Примеры:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.moveTopLeft(QtCore.QPoint(0, 0)); r
PyQt5.QtCore.QRect(0, 0, 400, 300)
>>> r.moveBottomRight(QtCore.QPoint(599, 499)); r
PyQt5.QtCore.QRect(200, 200, 400, 300)
```

- ◆ `moveCenter(<QPoint>)` — задает новые координаты центра;

- ◆ `translate(<Сдвиг по оси X>, <Сдвиг по оси Y>)` и `translate(<QPoint>)` — задают новые координаты левого верхнего угла относительно текущего значения координат:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.translate(20, 15); r
PyQt5.QtCore.QRect(20, 15, 400, 300)
>>> r.translate(QtCore.QPoint(10, 5)); r
PyQt5.QtCore.QRect(30, 20, 400, 300)
```

- ◆ `translated(<Сдвиг по оси X>, <Сдвиг по оси Y>)` и `translated(<QPoint>)` — аналогичен методу `translate()`, но возвращает новый экземпляр класса `QRect`, а не изменяет текущий;
  - ◆ `adjust(<X1>, <Y1>, <X2>, <Y2>)` — задает новые координаты левого верхнего и правого нижнего углов относительно текущих значений координат:
- ```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.adjust(10, 5, 10, 5); r
PyQt5.QtCore.QRect(10, 5, 400, 300)
```
- ◆ `adjusted(<X1>, <Y1>, <X2>, <Y2>)` — аналогичен методу `adjust()`, но возвращает новый экземпляр класса `QRect`, а не изменяет текущий.

Для получения параметров области предназначены следующие методы:

- ◆ `left()` и `x()` — возвращают координату левого верхнего угла по оси X;
- ◆ `top()` и `y()` — возвращают координату левого верхнего угла по оси Y;
- ◆ `right()` и `bottom()` — возвращают координаты правого нижнего угла по осям X и Y соответственно;
- ◆ `width()` и `height()` — возвращают ширину и высоту соответственно;
- ◆ `size()` — возвращает размеры в виде экземпляра класса `QSize`.

Примеры:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.left(), r.top(), r.x(), r.y(), r.right(), r.bottom()
(10, 15, 10, 15, 409, 314)
>>> r.width(), r.height(), r.size()
(400, 300, PyQt5.QtCore.QSize(400, 300))
```

- ◆ `topLeft()` — возвращает координаты левого верхнего угла;
- ◆ `topRight()` — возвращает координаты правого верхнего угла;
- ◆ `bottomLeft()` — возвращает координаты левого нижнего угла;
- ◆ `bottomRight()` — возвращает координаты правого нижнего угла.

Примеры:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.topLeft(), r.topRight()
(PyQt5.QtCore.QPoint(10, 15), PyQt5.QtCore.QPoint(409, 15))
>>> r.bottomLeft(), r.bottomRight()
(PyQt5.QtCore.QPoint(10, 314), PyQt5.QtCore.QPoint(409, 314))
```

- ◆ `center()` — возвращает координаты центра области. Например, вывести окно по центру доступной области экрана можно так:

```
desktop = QtWidgets.QApplication.desktop()
window.move(desktop.availableGeometry().center() -
            window.rect().center())
```

- ◆ `getRect()` — возвращает кортеж с координатами левого верхнего угла и размерами области;
- ◆ `getCoords()` — возвращает кортеж с координатами левого верхнего и правого нижнего углов:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.getRect(), r.getCoords()
((10, 15, 400, 300), (10, 15, 409, 314))
```

### Прочие методы:

- ◆ `isNull()` — возвращает `True`, если ширина и высота равны нулю, и `False` — в противном случае;
  - ◆ `isValid()` — возвращает `True`, если `left() < right()` и `top() < bottom()`, и `False` — в противном случае;
  - ◆ `isEmpty()` — возвращает `True`, если `left() > right()` или `top() > bottom()`, и `False` — в противном случае;
  - ◆ `normalized()` — исправляет ситуацию, при которой `left() > right()` или `top() > bottom()`, и возвращает новый экземпляр класса `QRect`:
- ```
>>> r = QtCore.QRect(QtCore.QPoint(409, 314), QtCore.QPoint(10, 15))
>>> r
PyQt5.QtCore.QRect(409, 314, -398, -298)
>>> r.normalized()
PyQt5.QtCore.QRect(10, 15, 400, 300)
```
- ◆ `contains(<QPoint>[, <Флаг>])` и `contains(<X>, <Y>[, <Флаг>])` — возвращает `True`, если точка с указанными координатами расположена внутри области или на ее границе, и `False` — в противном случае. Если во втором параметре указано значение `True`, то точка должна быть расположена только внутри области, а не на ее границе. Значение параметра по умолчанию — `False`:
  - ◆ `contains(<QRect>[, <Флаг>])` — возвращает `True`, если указанная область расположена внутри текущей области или на ее краю, и `False` — в противном случае. Если во втором параметре указано значение `True`, то указанная область должна быть расположена только внутри текущей области, а не на ее краю. Значение параметра по умолчанию — `False`:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.contains(QtCore.QRect(0, 0, 20, 5))
True
>>> r.contains(QtCore.QRect(0, 0, 20, 5), True)
False
```

- ◆ `intersects(<QRect>)` — возвращает `True`, если указанная область пересекается с текущей областью, и `False` — в противном случае;
- ◆ `intersected(<QRect>)` — возвращает область, которая расположена на пересечении текущей и указанной областей:

```
>>> r = QtCore.QRect(0, 0, 20, 20)
>>> r.intersects(QtCore.QRect(10, 10, 20, 20))
True
>>> r.intersected(QtCore.QRect(10, 10, 20, 20))
PyQt5.QtCore.QRect(10, 10, 10, 10)
```

- ◆ `united(<QRect>)` — возвращает область, которая охватывает текущую и указанную области:

```
>>> r = QtCore.QRect(0, 0, 20, 20)
>>> r.united(QtCore.QRect(30, 30, 20, 20))
PyQt5.QtCore.QRect(0, 0, 50, 50)
```

Над двумя экземплярами класса  `QRect` можно выполнять операции `&` и `&=` (пересечение), `|` и `|=` (объединение), `in` (проверка на вхождение), `==` и `!=`.

Пример:

```
>>> r1, r2 = QtCore.QRect(0, 0, 20, 20), QtCore.QRect(10, 10, 20, 20)
>>> r1 & r2, r1 | r2
(PyQt5.QtCore.QRect(10, 10, 10, 10), PyQt5.QtCore.QRect(0, 0, 30, 30))
>>> r1 in r2, r1 in QtCore.QRect(0, 0, 30, 30)
(False, True)
>>> r1 == r2, r1 != r2
(False, True)
```

Помимо этого, поддерживаются операторы `+` и `-`, выполняющие увеличение и уменьшение области на заданные величины границ, которые должны быть заданы в виде объекта класса `QMargins`:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> m = QtCore.QMargins(5, 15, 5, 15)
>>> r + m
PyQt5.QtCore.QRect(5, 0, 410, 330)
>>> r - m
PyQt5.QtCore.QRect(15, 30, 390, 270)
```

## 18.6. Разворачивание и сворачивание окна

В заголовке окна расположены кнопки **Свернуть** и **Развернуть**, с помощью которых можно свернуть окно в значок на панели задач или развернуть его на весь экран. Выполнить подобные действия из программы позволяют следующие методы класса `QWidget`:

- ◆ `showMinimized()` — сворачивает окно на панель задач. Эквивалентно нажатию кнопки **Свернуть** в заголовке окна;
- ◆ `showMaximized()` — разворачивает окно до максимального размера. Эквивалентно нажатию кнопки **Развернуть** в заголовке окна;
- ◆ `showFullScreen()` — включает полноэкранный режим отображения окна. Окно отображается без заголовка и границ;
- ◆ `showNormal()` — отменяет сворачивание, максимальный размер и полноэкранный режим, возвращая окно к изначальным размерам;
- ◆ `activateWindow()` — делает окно активным (т. е. имеющим фокус ввода). В Windows, если окно было ранее свернуто в значок на панель задач, оно не будет развернуто в изначальный вид;
- ◆ `setWindowState(<Флаги>)` — изменяет состояние окна в зависимости от переданных флагов. В качестве параметра указывается комбинация следующих атрибутов из класса `QtCore.Qt` через побитовые операторы:

- `WindowNoState` — нормальное состояние окна;
- `WindowMinimized` — окно свернуто;
- `WindowMaximized` — окно максимально развернуто;
- `WindowFullScreen` — полноэкранный режим;
- `WindowActive` — окно имеет фокус ввода, т. е. является активным.

Например, включить полноэкранный режим можно так:

```
window.setWindowState((window.windowState() &
    ~QtCore.Qt.WindowMinimized | QtCore.Qt.WindowMaximized) |
    QtCore.Qt.WindowFullScreen)
```

Проверить текущий статус окна позволяют следующие методы:

- ◆ `isMinimized()` — возвращает `True`, если окно свернуто, и `False` — в противном случае;
- ◆ `isMaximized()` — возвращает `True`, если окно раскрыто до максимальных размеров, и `False` — в противном случае;
- ◆ `isFullScreen()` — возвращает `True`, если включен полноэкранный режим, и `False` — в противном случае;
- ◆ `isActiveWindow()` — возвращает `True`, если окно имеет фокус ввода, и `False` — в противном случае;
- ◆ `windowState()` — возвращает комбинацию флагов, обозначающих текущий статус окна.

Пример проверки использования полноэкранного режима:

```
if window.windowState() & QtCore.Qt.WindowFullScreen:
    print("Полноэкранный режим")
```

Пример разворачивания и сворачивания окна приведен в листинге 18.4.

#### Листинг 18.4. Разворачивание и сворачивание окна

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

class MyWindow(QtWidgets.QWidget):
    def __init__(self, parent=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.btnMin = QtWidgets.QPushButton("Свернуть")
        self.btnClose = QtWidgets.QPushButton("Развернуть")
        self.btnFull = QtWidgets.QPushButton("Полный экран")
        self.btnNormal = QtWidgets.QPushButton("Нормальный размер")
        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(self.btnClose)
        vbox.addWidget(self.btnClose)
        vbox.addWidget(self.btnClose)
        vbox.addWidget(self.btnClose)
        self.setLayout(vbox)
        self.btnClose.clicked.connect(self.on_min)
        self.btnClose.clicked.connect(self.on_max)
        self.btnClose.clicked.connect(self.on_full)
        self.btnClose.clicked.connect(self.on_normal)
```

```
def on_min(self):
    self.showMinimized()
def on_max(self):
    self.showMaximized()
def on_full(self):
    self.showFullScreen()
def on_normal(self):
    self.showNormal()

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Разворачивание и сворачивание окна")
    window.resize(300, 100)
    window.show()
    sys.exit(app.exec_())
```

## 18.7. Управление прозрачностью окна

Сделать окно полупрозрачным позволяет метод `setWindowOpacity()` класса `QWidget`. Формат метода:

```
setWindowOpacity(<вещественное число от 0.0 до 1.0>)
```

Число 0.0 соответствует полностью прозрачному окну, а число 1.0 — отсутствию прозрачности. Для получения степени прозрачности окна из программы предназначен метод `windowOpacity()`. Выведем окно со степенью прозрачности 0.5 (листинг 18.5).

### Листинг 18.5. Полупрозрачное окно

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

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Полупрозрачное окно")
window.resize(300, 100)
window.setWindowOpacity(0.5)
window.show()
print(window.windowOpacity()) # Выведет: 0.4980392156862745
sys.exit(app.exec_())
```

## 18.8. Модальные окна

*Модальным* называется окно, которое не позволяет взаимодействовать с другими окнами в том же приложении, — пока модальное окно не будет закрыто, сделать активным другое окно нельзя. Например, если в программе Microsoft Word выбрать пункт меню **Файл | Сохранить как**, откроется модальное диалоговое окно, позволяющее выбрать путь и название

файла, и, пока это окно не будет закрыто, вы не сможете взаимодействовать с главным окном приложения.

Указать, что окно является модальным, позволяет метод `setWindowModality(<флаг>)` класса `QWidget`. В качестве параметра могут быть указаны следующие атрибуты из класса `QtCore.Qt`:

- ◆ `NonModal` — 0 — окно не является модальным (поведение по умолчанию);
- ◆ `WindowModal` — 1 — окно блокирует только родительские окна в пределах иерархии;
- ◆ `ApplicationModal` — 2 — окно блокирует все окна в приложении.

Окна, открытые из модального окна, не блокируются. Следует также учитывать, что метод `setWindowModality()` должен быть вызван до отображения окна.

Получить текущее значение модальности позволяет метод `windowModality()`. Проверить, является ли окно модальным, можно с помощью метода `isModal()` — он возвращает `True`, если окно является модальным, и `False` — в противном случае.

Создадим два независимых окна. В первом окне разместим кнопку, по нажатию которой откроется модальное окно, — оно будет блокировать только первое окно, но не второе. При открытии модального окна отобразим его примерно по центру родительского окна (листинг 18.6).

#### Листинг 18.6. Модальные окна

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

def show_modal_window():
    global modalWindow
    modalWindow = QtWidgets.QWidget(window1, QtCore.Qt.Window)
    modalWindow.setWindowTitle("Модальное окно")
    modalWindow.resize(200, 50)
    modalWindow.setWindowModality(QtCore.Qt.WindowModal)
    modalWindow.setAttribute(QtCore.Qt.WA_DeleteOnClose, True)
    modalWindow.move(window1.geometry().center() - modalWindow.rect().center() - QtCore.QPoint(4, 30))
    modalWindow.show()

app = QtWidgets.QApplication(sys.argv)
window1 = QtWidgets.QWidget()
window1.setWindowTitle("Обычное окно")
window1.resize(300, 100)
button = QtWidgets.QPushButton("Открыть модальное окно")
button.clicked.connect(show_modal_window)
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(button)
window1.setLayout(vbox)
window1.show()

window2 = QtWidgets.QWidget()
window2.setWindowTitle("Это окно не будет блокировано при WindowModal")
```

```
window2.resize(500, 100)
window2.show()

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

Если запустить приложение и нажать кнопку **Открыть модальное окно**, откроется окно, выровненное примерно по центру родительского окна (произвести точное выравнивание вы сможете самостоятельно). При этом получить доступ к родительскому окну можно только после закрытия модального окна, второе же окно блокировано не будет. Если заменить атрибут `WindowModal` атрибутом `ApplicationModal`, оба окна будут блокированы.

Обратите внимание, что в конструктор модального окна мы передали ссылку на первое окно и атрибут `Window`. Если не указать ссылку, то окно блокировано не будет, а если не указать атрибут, окно вообще не откроется. Кроме того, мы объявили переменную `modalWindow` глобальной, иначе при достижении конца функции переменная выйдет из области видимости, и окно будет автоматически удалено. Чтобы объект окна автоматически удалялся при закрытии окна, атрибуту `WA_DeleteOnClose` в методе `setAttribute()` было присвоено значение `True`.

Модальные окна в большинстве случаев являются диалоговыми. Для работы с такими окнами в PyQt предназначен класс `QDialog`, который автоматически выравнивает окно по центру экрана или родительского окна. Кроме того, этот класс предоставляет множество специальных методов, позволяющих дождаться закрытия окна, определить статус завершения и выполнить другие действия. Подробно класс `QDialog` мы рассмотрим в главе 26.

## 18.9. Смена значка в заголовке окна

По умолчанию в левом верхнем углу окна отображается стандартный значок. Отобразить другой значок позволяет метод `setWindowIcon()` класса `QWidget`. В качестве параметра метод принимает экземпляр класса `QIcon` из модуля `QtGui` (см. разд. 24.3.4).

Чтобы загрузить значок из файла, следует передать путь к файлу конструктору класса `QIcon`. Если указан относительный путь, поиск файла будет производиться относительно текущего рабочего каталога. Получить список поддерживаемых форматов файлов можно с помощью статического метода `supportedImageFormats()` класса `QImageReader`, объявленного в модуле `QtGui`. Метод возвращает список с экземплярами класса `QByteArray`. Получим список поддерживаемых форматов:

```
>>> from PyQt5 import QtGui
>>> for i in QtGui.QImageReader.supportedImageFormats():
    print(str(i, "ascii").upper(), end=" ")
```

Вот результат выполнения этого примера на компьютере одного из авторов:

BMP CUR GIF ICNS ICO JPEG JPEG PBM PGM PNG PPM SVG SVGZ TGA TIF TIFF WBMP WEBP XBM XPM

Если для окна не указать значок, будет использоваться значок приложения, установленный с помощью метода `setWindowIcon()` класса `QApplication`. В качестве параметра метод также принимает экземпляр класса `QIcon`.

Вместо загрузки значка из файла можно воспользоваться одним из встроенных значков. Загрузить стандартный значок позволяет следующий код:

```
ico = window.style().standardIcon(QtWidgets.QStyle.SP_MessageBoxCritical)
window.setWindowIcon(ico)
```

Посмотреть список всех встроенных значков можно в документации к классу `QStyle` (см. <https://doc.qt.io/qt-5/qstyle.html#StandardPixmap-enum>).

В качестве примера создадим значок размером 16 на 16 пикселов в формате PNG и сохраним его в одном каталоге с программой, после чего установим этот значок для окна и всего приложения (листинг 18.7).

#### Листинг 18.7. Смена значка в заголовке окна

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

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Смена значка в заголовке окна")
window.resize(300, 100)
ico = QtGui.QIcon("icon.png")           # Значок для окна
window.setWindowIcon(ico)               # Значок приложения
app.setWindowIcon(ico)
window.show()
sys.exit(app.exec_())
```

## 18.10. Изменение цвета фона окна

Чтобы изменить цвет фона окна (или компонента), следует установить палитру с настроенной ролью `Window` (или `Background`). Цветовая палитра содержит цвета для каждой роли и состояния компонента. Указать состояние компонента позволяют следующие атрибуты из класса `QPalette` (модуль `QtGui`):

- ◆ `Active` и `Normal` — 0 — компонент активен (окно находится в фокусе ввода);
- ◆ `Disabled` — 1 — компонент недоступен;
- ◆ `Inactive` — 2 — компонент неактивен (окно находится вне фокуса ввода).

Получить текущую палитру компонента позволяет его метод `palette()`. Чтобы изменить цвет для какой-либо роли и состояния, следует воспользоваться методом `setColor()` класса `QPalette`. Формат метода:

```
setColor([<Состояние>, ]<Роль>, <Цвет>)
```

В параметре `<Роль>` указывается, для какого элемента изменяется цвет. Например, атрибут `Window` (или `Background`) изменяет цвет фона, а `WindowText` (или `Foreground`) — цвет текста. Полный список атрибутов имеется в документации по классу `QPalette` (см. <https://doc.qt.io/qt-5/qpalette.html>).

В параметре `<Цвет>` указывается цвет элемента. В качестве значения можно указать атрибут из класса `QtCore.Qt` (например, `black`, `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.).

После настройки палитры необходимо вызвать метод `setPalette()` компонента и передать этому методу измененный объект палитры. Следует помнить, что компоненты-потомки по умолчанию имеют прозрачный фон и не перерисовываются автоматически. Чтобы вклю-

чить перерисовку, необходимо передать значение `True` методу `setAutoFillBackground()` окна.

Изменить цвет фона также можно с помощью CSS-атрибута `background-color`. Для этого следует передать таблицу стилей в метод `setStyleSheet()` компонента. Таблицы стилей могут быть внешними (подключение через командную строку), установленными на уровне приложения (с помощью метода `setStyleSheet()` класса `QApplication`) или установленными на уровне компонента (с помощью метода `setStyleSheet()` класса `QWidget`). Атрибуты, установленные последними, обычно перекрывают значения аналогичных атрибутов, указанных ранее. Если вы занимались веб-программированием, то язык CSS (каскадные таблицы стилей) вам уже знаком, а если нет, придется дополнительно его изучить.

Создадим окно с надписью. Для активного окна установим зеленый цвет, а для неактивного — красный. Цвет фона надписи сделаем белым. Для изменения фона окна используем палитру, а для изменения цвета фона надписи — CSS-атрибут `background-color` (листинг 18.8).

#### Листинг 18.8. Изменение цвета фона окна

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

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Изменение цвета фона окна")
window.resize(300, 100)
pal = window.palette()
pal.setColor(QtGui.QPalette.Normal, QtGui.QPalette.Window,
            QtGui.QColor("#008800"))
pal.setColor(QtGui.QPalette.Inactive, QtGui.QPalette.Window,
            QtGui.QColor("#ff0000"))
window.setPalette(pal)
label = QtWidgets.QLabel("Текст надписи")
label.setAlignment(QtCore.Qt.AlignHCenter)
label.setStyleSheet ("background-color: #ffffff; ")
label.setAutoFillBackground(True)
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(label)
window.setLayout(vbox)
window.show()
sys.exit(app.exec_())
```

## 18.11. Вывод изображения в качестве фона

В качестве фона окна (или компонента) можно использовать изображение. Для этого необходимо получить текущую палитру компонента с помощью метода `palette()`, а затем вызвать метод `setBrush()` класса `QPalette`. Формат метода:

```
setBrush([<Состояние>, ]<Роль>, <QBrush>)
```

Первые два параметра аналогичны соответствующим параметрам в методе `setColor()`, который мы рассматривали в предыдущем разделе. В третьем параметре указывается кисть — экземпляр класса `QBrush` из модуля `QtGui`. Форматы конструктора класса:

```
<Объект> = QBrush(<Стиль кисти>)
<Объект> = QBrush(<Цвет>[, <Стиль кисти>=SolidPattern])
<Объект> = QBrush(<Цвет>, <QPixmap>)
<Объект> = QBrush(<QPixmap>)
<Объект> = QBrush(<QImage>)
<Объект> = QBrush(<QBrush>)
<Объект> = QBrush(<QGradient>)
```

В параметре `<Стиль кисти>` указываются атрибуты из класса `QtCore.Qt`, задающие стиль кисти, — например: `NoBrush`, `SolidPattern`, `Dense1Pattern`, `Dense2Pattern`, `Dense3Pattern`, `Dense4Pattern`, `Dense5Pattern`, `Dense6Pattern`, `Dense7Pattern`, `CrossPattern` и др. С помощью этого параметра можно сделать цвет сплошным (`SolidPattern`) или имеющим текстуру (например, атрибут `CrossPattern` задает текстуру в виде сетки).

В параметре `<Цвет>` указывается цвет кисти. В качестве значения можно указать атрибут из класса `QtCore.Qt` (например, `black`, `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.). При этом установка сплошного цвета фона окна может выглядеть так:

```
pal = window.palette()
pal.setBrush(QtGui.QPalette.Normal, QtGui.QPalette.Window,
    QtGui.QBrush(QtGui.QColor("#008800"), QtCore.Qt.SolidPattern))
window.setPalette(pal)
```

Параметры `<QPixmap>` и `<QImage>` позволяют передать объекты изображений. Конструкторы этих классов принимают путь к файлу — абсолютный или относительный.

Параметр `<QBrush>` позволяет создать кисть на основе другой кисти, а параметр `<QGradient>` — на основе градиента, представленного объектом класса `QGradient` (см. главу 24).

После настройки палитры необходимо вызвать метод `setPalette()` и передать ему измененный объект палитры. Следует помнить, что компоненты-потомки по умолчанию имеют прозрачный фон и не перерисовываются автоматически. Чтобы включить перерисовку, необходимо передать значение `True` в метод `setAutoFillBackground()`.

Указать, какое изображение используется в качестве фона, также можно с помощью CSS-атрибутов `background` и `background-image`. С помощью CSS-атрибута `background-repeat` можно дополнительно указать режим повтора фонового рисунка. Он может принимать значения `repeat` (повтор по горизонтали и вертикали), `repeat-x` (повтор только по горизонтали), `repeat-y` (повтор только по вертикали) и `no-repeat` (не повторяется).

Создадим окно с надписью. Для активного окна установим одно изображение (с помощью изменения палитры), а для надписи — другое (с помощью CSS-атрибута `background-image`) (листинг 18.9).

#### Листинг 18.9. Использование изображения в качестве фона

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

```
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowTitle("Изображение в качестве фона")
window.resize(300, 100)
pal = window.palette()
pal.setBrush(QtGui.QPalette.Normal, QtGui.QPalette.Window,
            QtGui.QBrush(QtGui.QPixmap("background1.jpg")))
window.setPalette(pal)
label = QtWidgets.QLabel("Текст надписи")
label.setAlignment(QtCore.Qt.AlignCenter)
label.setStyleSheet("background-image: url(background2.jpg);")
label.setAutoFillBackground(True)
vbox = QtWidgets.QVBoxLayout()
vbox.addWidget(label)
window.setLayout(vbox)
window.show()
sys.exit(app.exec_())
```

## 18.12. Создание окна произвольной формы

Чтобы создать окно произвольной формы, нужно выполнить следующие шаги:

1. Создать изображение нужной формы с прозрачным фоном и сохранить его, например, в формате PNG.
2. Создать экземпляр класса `QPixmap`, передав конструктору класса абсолютный или относительный путь к изображению.
3. Установить изображение в качестве фона окна с помощью палитры.
4. Отделить альфа-канал с помощью метода `mask()` класса `QPixmap`.
5. Передать получившуюся маску в метод `setMask()` окна.
6. Убрать рамку окна, например, передав комбинацию следующих флагов:  
`QtCore.Qt.Window | QtCore.Qt.FramelessWindowHint`

Если для создания окна используется класс `QLabel`, то вместо установки палитры можно передать экземпляр класса `QPixmap` в метод `setPixmap()`, а маску — в метод `setMask()`.

Для примера создадим круглое окно с кнопкой, с помощью которой можно закрыть окно. Для использования в качестве маски создадим изображение в формате PNG, установим для него прозрачный фон и нарисуем белый круг, занимающий все это изображение. Окно выведем без заголовка и границ (листинг 18.10).

### Листинг 18.10. Создание окна произвольной формы

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

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget()
window.setWindowFlags(QtCore.Qt.Window | QtCore.Qt.FramelessWindowHint)
```

```
window.setWindowTitle("Создание окна произвольной формы")
window.resize(300, 300)
pixmap = QtGui.QPixmap("mask.png")
pal = window.palette()
pal.setBrush(QtGui.QPalette.Normal, QtGui.QPalette.Window,
            QtGui.QBrush(pixmap))
pal.setBrush(QtGui.QPalette.Inactive, QtGui.QPalette.Window,
            QtGui.QBrush(pixmap))
window.setPalette(pal)
window.setMask(pixmap.mask())
button = QtWidgets.QPushButton("Закрыть окно", window)
button.setFixedSize(150, 30)
button.move(75, 135)
button.clicked.connect(QtWidgets.qApp.quit)
window.show()
sys.exit(app.exec_())
```

Получившееся окно можно увидеть на рис. 18.1.

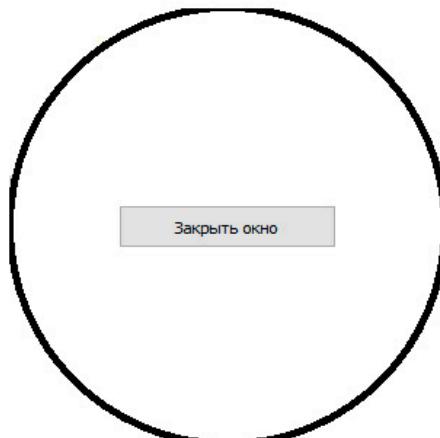


Рис. 18.1. Окно круглой формы

## 18.13. Всплывающие подсказки

При работе с программой у пользователя могут возникать вопросы о предназначении того или иного компонента. Обычно для информирования пользователя служат надписи, расположенные над компонентом или левее его. Но часто либо место в окне ограничено, либо вывод этих надписей портит весь дизайн окна. В таких случаях принято выводить текст подсказки в отдельном окне без рамки при наведении указателя мыши на компонент. Подсказка автоматически скроется после увода курсора мыши или спустя определенное время.

В PyQt нет необходимости создавать окно с подсказкой самому и следить за перемещениями указателя мыши — весь этот процесс автоматизирован и максимально упрощен. Чтобы создать всплывающие подсказки для окна или любого другого компонента и управлять ими, нужно воспользоваться следующими методами класса `QWidget`:

- ◆ `setToolTip(<Текст>)` — задает текст всплывающей подсказки. В качестве параметра можно указать простой текст или HTML-код. Чтобы отключить вывод подсказки, достаточно передать в этот метод пустую строку;
- ◆ `toolTip()` — возвращает текст всплывающей подсказки;
- ◆ `setToolTipDuration(<Время>)` — задает время, в течение которого всплывающая подсказка будет присутствовать на экране. Значение должно быть указано в миллисекундах. Если задать значение `-1`, PyQt будет сама вычислять необходимое время, основываясь на длине текста подсказки (это поведение по умолчанию);
- ◆ `toolTipDuration()` — возвращает время, в течение которого всплывающая подсказка будет присутствовать на экране;
- ◆ `setWhatsThis(<Текст>)` — задает текст справки. Обычно этот метод используется для вывода информации большего объема, чем во всплывающей подсказке. У диалоговых окон в заголовке окна есть кнопка **Справка**, по нажатию которой курсор принимает вид стрелки со знаком вопроса, — чтобы в таком случае отобразить текст справки, следует нажать эту кнопку и щелкнуть на компоненте. Можно также сделать компонент активным и нажать комбинацию клавиш `<Shift>+<F1>`. В качестве параметра можно указать простой текст или HTML-код. Чтобы отключить вывод подсказки, достаточно передать в этот метод пустую строку;
- ◆ `whatsThis()` — возвращает текст справки.

Создадим окно с кнопкой и зададим для них текст всплывающих подсказок и текст справки (листинг 18.11).

#### Листинг 18.11. Всплывающие подсказки

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

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget(flags=QtCore.Qt.Dialog)
window.setWindowTitle("Всплывающие подсказки")
window.resize(300, 70)
button = QtWidgets.QPushButton("Закрыть окно", window)
button.setFixedSize(150, 30)
button.move(75, 20)
button.setToolTip("Это всплывающая подсказка для кнопки")
button.setToolTipDuration(3000)
window.setToolTip("Это всплывающая подсказка для окна")
button.setToolTipDuration(5000)
button.setWhatsThis("Это справка для кнопки")
window.setWhatsThis("Это справка для окна")
button.clicked.connect(QtWidgets.qApp.quit)
window.show()
sys.exit(app.exec_())
```

## 18.14. Программное закрытие окна

В предыдущих разделах для закрытия окна мы использовали слот `quit()` и метод `exit([returnValue=0])` объекта приложения. Однако эти методы не только закрывают текущее окно, но и завершают выполнение всего приложения. Чтобы закрыть только текущее окно, следует воспользоваться методом `close()` класса `QWidget`. Метод возвращает значение `True`, если окно успешно закрыто, и `False` — в противном случае. Закрыть сразу все окна приложения позволяет слот `closeAllWindows()` класса `QApplication`.

Если для окна атрибут `WA_DeleteOnClose` из класса `QtCore.Qt` установлен в значение `True`, после закрытия окна объект окна будет автоматически удален, в противном случае окно просто скрывается. Значение атрибута можно изменить с помощью метода `setAttribute()`:

```
window.setAttribute(QtCore.Qt.WA_DeleteOnClose, True)
```

После вызова метода `close()` или нажатия кнопки **Закрыть** в заголовке окна генерируется событие `QEvent.Close`. Если внутри класса определить метод с предопределенным названием `closeEvent()`, это событие можно перехватить и обработать. В качестве параметра метод принимает объект класса `QCloseEvent`, который поддерживает методы `accept()` (позволяет закрыть окно) и `ignore()` (запрещает закрытие окна). Вызывая эти методы, можно контролировать процесс закрытия окна.

В качестве примера закроем окно по нажатию кнопки (листинг 18.12).

### Листинг 18.12. Программное закрытие окна

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

app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QWidget(flags=QtCore.Qt.Dialog)
window.setWindowTitle("Закрытие окна из программы")
window.resize(300, 70)
button = QtWidgets.QPushButton("Закрыть окно", window)
button.setFixedSize(150, 30)
button.move(75, 20)
button.clicked.connect(window.close)
window.show()
sys.exit(app.exec_())
```

#### ПРИМЕЧАНИЕ

Закрыв последнее окно приложения, мы тем самым автоматически завершим и само приложение. Не забываем об этом.

## 18.15. Использование таблиц стилей CSS для оформления окон

Вернемся к методу `setStyleSheet()`, упомянутому в разд. 18.10 и предназначенному для задания таблиц стилей у приложений и отдельных элементов управления. С помощью таблиц стилей можно задавать не только цвет фона и фоновое изображение, но и другие параметры оформления.

Метод `setStyleSheet()` поддерживается классами `QWidget` (и всеми его подклассами) и `QApplication`. Следовательно, его можно вызвать у:

- ◆ самого приложения — тогда заданные в таблице стилей параметры оформления будут применены ко всем элементам управления всех окон приложения;
- ◆ отдельного окна — тогда эти параметры будут действовать в пределах данного окна;
- ◆ отдельного элемента управления — тогда они будут действовать только на этот элемент управления.

При указании таблицы стилей у приложения и окна можно использовать привычный нам по CSS формат **объявления стилей**:

```
<Селектор> {<Определение стилей>}
```

<Селектор> записывается в следующем формате:

```
<Основной селектор> [<Дополнительный селектор>] [<Псевдокласс>] [<Псевдоселектор>]
```

Параметр `<Основной селектор>` указывает на класс элемента управления. Его можно указать в одном из следующих форматов:

- ◆ \* (звездочка) — указывает на все элементы управления (*универсальный селектор*). Например, так можно задать для всех элементов управления зеленый цвет текста:  
`* {color: green;}`
- ◆ <Класс> — указывает на элементы управления, относящиеся к заданному <Классу> и его подклассам. Задание красного цвета текста для всех элементов управления, относящихся к классу `QAbstractButton` и его подклассам, т. е. для командных кнопок, флагков и переключателей, осуществляется так:

```
QAbstractButton {color: red;}
```

- ◆ .<Класс> — указывает только на элементы управления, относящиеся к заданному <Классу>, но не к его подклассам. Указание полужирного шрифта для всех элементов управления, относящихся к классу `QPushButton` (командных кнопок), но не для его подклассов, осуществляется так:

```
.QPushButton {font-weight: bold;}
```

Параметр `<Дополнительный селектор>` задает дополнительные параметры элемента управления. Его форматы:

- ◆ [`<Свойство>=<Значение>`] — указанное <Свойство> элемента управления должно иметь заданное <Значение>. Так мы задаем полужирный шрифт для кнопки, чье свойство `default` имеет значение `true`, т. е. для кнопки по умолчанию:

```
QPushButton[default="true"] {font-weight: bold;}
```

- ◆ #<Имя> — указывает на элемент управления, для которого было задано <Имя>. <Имя> можно задать вызовом у элемента управления метода `setObjectName(<Имя>)`, а получить — вызовом метода `objectName()`. Так выполняется указание красного цвета текста для кнопки с именем `btnRed`:

```
QPushButton#btnRed {color: red;}
```

Параметр `<Псевдокласс>` указывает на отдельную составную часть сложного элемента управления. Он записывается в формате `:<Обозначение составной части>`. Вот пример указания графического изображения для кнопки разворачивания раскрывающегося списка (обозначение этой составной части — `down-arrow`):

```
QComboBox::down-arrow {image: url(arrow.png);}
```

Параметр <Псевдоселектор> указывает на состояние элемента управления (должна ли быть кнопка нажата, должен ли флагок быть установленным и т. п.). Он может быть записан в двух форматах:

- ◆ :<Обозначение состояния> — элемент управления должен находиться в указанном состоянии. Вот пример указания белого цвета фона для кнопки, когда она нажата (это состояние имеет обозначение `pressed`):

```
QPushButton:pressed {background-color: white;}
```

- ◆ :!<Обозначение состояния> — элемент управления должен находиться в любом состоянии, кроме указанного. Вот пример указания желтого цвета фона для кнопки, когда она не нажата:

```
QPushButton:!pressed {background-color: yellow;}
```

Можно указать сразу несколько псевдоселекторов, расположив их непосредственно друг за другом — тогда селектор будет указывать на элемент управления, находящийся одновременно во всех состояниях, которые обозначены этими селекторами. Вот пример указания черного цвета фона и белого цвета текста для кнопки, которая нажата и над которой находится курсор мыши (обозначение — `hover`):

```
QPushButton:pressed:hover {color: white; background-color: black;}
```

Если нужно указать стиль для элемента управления, вложенного в другой элемент управления, применяется следующий формат указания селектора:

<Селектор "внешнего" элемента><Разделитель><Селектор вложенного элемента>

Поддерживаются два варианта параметра <Разделитель>:

- ◆ пробел — <Вложенный элемент> не обязательно должен быть вложен непосредственно во <"Внешний">. Так мы указываем зеленый цвет фона для всех надписей (`QLabel`), вложенных в группу (`QGroupBox`) и вложенные в нее элементы:

```
QGroupBox QLabel {background-color: green;}
```

- ◆ > — <Вложенный элемент> обязательно должен быть вложен непосредственно во <"Внешний">. Так мы укажем синий цвет текста для всех надписей, непосредственно вложенных в группу:

```
QGroupBox>QLabel {color: blue;}
```

В стиле можно указать сразу несколько селекторов, записав их через запятую — тогда стиль будет применен к элементам управления, на которые указывают эти селекторы. Вот пример задания зеленого цвета фона для кнопок и надписей:

```
QLabel, QPushButton {background-color: green;}
```

В CSS элементы страницы наследуют параметры оформления от их родителей. Но в PyQt это не так. Скажем, если мы укажем для группы красный цвет текста:

```
app.setStyleSheet("QGroupBox {color: red; }")
```

вложенные в эту группу элементы не унаследуют его и будут иметь цвет текста, заданный по умолчанию. Нам придется задать для них нужный цвет явно:

```
app.setStyleSheet("QGroupBox, QGroupBox * {color: red; }")
```

Начиная с версии PyQt 5.7, поддерживается возможность указать библиотеке, что все элементы-потомки должны наследовать параметры оформления у родителя. Для этого достаточно вызвать у класса `QCoreApplication` статический метод `setAttribute`, передав ему

в качестве первого параметра значение атрибута AA\_UseStyleSheetPropagationInWidgetStyles класса QtCore.Qt, а в качестве второго параметра — значение True:

```
QtCore.QCoreApplication.setAttribute(QtCore.Qt.AA_UseStyleSheetPropagationInWidgetStyles, True)
```

Чтобы отключить такую возможность, достаточно вызвать этот метод еще раз, указав в нем вторым параметром False.

И, наконец, при вызове метода setStyleSheet() у элемента управления, для которого следует задать таблицу стилей, в последней не указываются ни селектор, ни фигурные скобки — они просто не нужны.

Отметим, что в случае PyQt, как и в CSS, также действуют правила каскадности. Так, таблица стилей, заданная для окна, имеет больший приоритет, нежели таковая, указанная для приложения, а стиль, что был задан для элемента управления, имеет наивысший приоритет. Помимо этого, более специфические стили имеют больший приоритет, чем менее специфические; так, стиль с селектором, в чей состав входит имя элемента управления, перекроет стиль с селектором любого другого типа.

За более подробным описанием поддерживаемых PyQt псевдоклассов, псевдоселекторов и особенностей указания стилей для отдельных классов элементов управления обращайтесь по интернет-адресу <https://doc.qt.io/qt-5/stylesheets-reference.html>.

Листинг 18.13 показывает пример задания таблиц стилей для элементов управления различными способами. Результат выполнения приведенного в нем кода можно увидеть на рис. 18.2.

#### Листинг 18.13. Использование таблиц стилей для указания оформления

```
# -*- coding: utf-8 -*-
from PyQt5 import QtWidgets
import sys
app = QtWidgets.QApplication(sys.argv)
# На уровне приложения задаем синий цвет текста для надписей, вложенных в группы,
# и курсивное начертание текста кнопок
app.setStyleSheet("QGroupBox QLabel {color: blue;} QPushButton {font-style: italic;}")
window = QtWidgets.QWidget()
window.setWindowTitle("Таблицы стилей")
# На уровне окна задаем зеленый цвет текста для надписи с именем first и
# красный цвет текста для надписи, на которую наведен курсор мыши
window.setStyleSheet("QLabel#first {color: green;} QLabel:hover {color: red;}")
window.resize(200, 150)
# Создаем три надписи
lbl1 = QtWidgets.QLabel("Зеленый текст")
# Указываем для первой надписи имя first
lbl1.setObjectName("first")
lbl2 = QtWidgets.QLabel("Полужирный текст")
# Для второй надписи указываем полужирный шрифт
lbl2.setStyleSheet("font-weight: bold")
lbl3 = QtWidgets.QLabel("Синий текст")
# Создаем кнопку
btn = QtWidgets.QPushButton("Курсивный текст")
```

```
# Создаем группу
box = QtWidgets.QGroupBox("Группа")
# Создаем контейнер, помещаем в него третью надпись и вставляем в группу
bbox = QtWidgets.QVBoxLayout()
bbox.addWidget(lbl3)
box.setLayout(bbox)
# Создаем еще один контейнер, помещаем в него две первые надписи, группу и кнопку и
# вставляем в окно
mainbox = QtWidgets.QVBoxLayout()
mainbox.addWidget(lbl1)
mainbox.addWidget(lbl2)
mainbox.addWidget(box)
mainbox.addWidget(btn)
window.setLayout(mainbox)
# Выводим окно и запускаем приложение
window.show()
sys.exit(app.exec_())
```

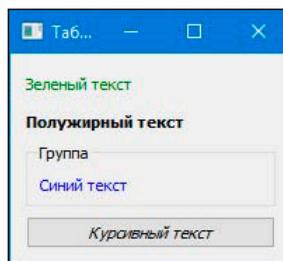


Рис. 18.2. Пример использования таблиц стилей