# 8

# Экскурсия по tkinter, часть 1

## «Виджеты, гаджеты, графические интерфейсы... Бог мой!»

В этой главе будет продолжено рассмотрение приемов программирования графических интерфейсов на языке Python. В предыдущей главе рассматривались простые виджеты – кнопки, метки и другие – демонстрирующие основы использования библиотеки tkinter в сценариях на языке Python. Такое упрощение было намеренным: легче охватить взглядом картину графического интерфейса целиком, если не приходится вникать в детали интерфейса виджетов. Но теперь, после знакомства с основами, в этой и следующей главе мы переходим к представлению более сложных объектов виджетов и средств, предоставляемых библиотекой tkinter.

Вы увидите, как разработка сценариев с графическим интерфейсом станет полезным и интересным делом. В этих двух главах мы познакомимся с классами, участвующими в построении элементов интерфейса, которые встречаются в настоящих программах, – ползунков, флажков, меню, прокручиваемых списков, диалогов, графики и так далее. За этими главами последует еще одна, завершающая, посвященная графическим интерфейсам и рассматривающая еще более крупные примеры, в которых применяются приемы и интерфейсы, демонстрировавшиеся во всех предшествующих главах, в которых говорилось о создании графических интерфейсов. В этих же двух главах примеры будут небольшими и самодостаточными, чтобы позволить сосредоточиться на особенностях виджетов.

# Темы этой главы

Формально мы уже использовали ряд простых виджетов в главе 7. Пока мы познакомились с классами `Label`, `Button`, `Frame` и `Tk` и попутно изучили понятия управления компоновкой в методе `pack`. Несмотря на свою простоту, все эти классы достаточно полно представляют интерфейсы библиотеки tkinter в целом и служат рабочими лошадками в типичных графических интерфейсах. Например, контейнеры `Frame` служат основой иерархической структуры отображения.

В этой и следующей главах мы исследуем дополнительные параметры уже знакомых графических элементов и, отойдя от основ, расскажем об остальной части набора виджетов tkinter. Ниже перечислены некоторые виджеты и темы, которые будут рассматриваться в данной главе:

- Виджеты `Toplevel` и `Tk`
- Виджеты `Message` и `Entry`
- Виджеты `Checkbutton`, `Radiobutton` и `Scale`
- Изображения: объекты `PhotoImage` и `BitmapImage`
- Параметры настройки виджетов и окон
- Диалоги: стандартные и пользовательские
- Низкоуровневое связывание событий
- Объекты связанных переменных tkinter
- Использование библиотеки Python обработки изображений – расширения PIL (Python Imaging Library) – для работы с изображениями других типов

Глава 9 завершает краткий рассказ, представляя остальные элементы инструментария библиотеки tkinter: меню, текст, холсты, анимацию и другие.

Чтобы сделать этот обзор интереснее, я также попутно введу некоторые идеи повторного использования компонентов. Например, некоторые более поздние примеры будут написаны с использованием компонентов, реализованных для предыдущих примеров. Хотя эти две главы знакомят с интерфейсами, основанными на виджетах, тем не менее данная книга написана также о программировании на языке Python в целом – как будет показано, программирование с использованием библиотеки tkinter в сценариях на языке Python может быть значительно более содержательным, чем просто рисование кружков и стрелок.

# Настройка внешнего вида виджетов

До сих пор все кнопки и метки в наших примерах выводились с оформлением по умолчанию, стандартном для соответствующей платформы. В Windows это обычно означает, что они выводятся серым цветом, как

в цветовой схеме, установленной на моем компьютере. Однако библиотека tkinter позволяет придавать виджетам любой другой внешний вид с помощью ряда параметров настройки виджетов и компоновки.

Поскольку обычно я не могу устоять перед соблазном определить для виджетов в примерах собственные настройки, хочу осветить эту тему в самом начале обзора. В примере 8.1 приводятся некоторые параметры настройки, доступные в tkinter.

*Пример 8.1. PP4E\Gui\Tour\config-label.py*

```
from tkinter import *
root = Tk()
labelfont = ('times', 20, 'bold')      # семейство, размер, стиль
widget = Label(root, text='Hello config world')
widget.config(bg='black', fg='yellow') # желтый текст на черном фоне
widget.config(font=labelfont)          # использовать увеличенный шрифт
widget.config(height=3, width=20)      # начальный размер: строк,символов
widget.pack(expand=YES, fill=BOTH)
root.mainloop()
```

Запомните, что с помощью метода `config` виджета можно в любой момент переопределить значения его параметров, что позволяет не передавать их все конструктору объекта. В данном случае мы используем эту возможность, чтобы установить параметры, которые определяют окно, изображенное на рис. 8.1.


***Рис. 8.1.*** *Нестандартный внешний вид метки*

Если запустить сценарий на компьютере (увы, я не могу показать здесь в цвете, как это выглядит), то вы увидите, что текст метки выводится желтым цветом на черном фоне, причем шрифт сильно отличается от того, что мы до сих пор видели. Этот сценарий настраивает следующие параметры отображения метки:

*Цвет*

Установкой параметра `bg` метки определяется черный цвет ее фона. Аналогично параметр `fg` изменяет цвет переднего плана (текста) метки на желтый. Эти параметры цвета присутствуют у большинства виджетов tkinter, и в них указывается простое название цвета

(например, 'blue') или шестнадцатеричная строка. Поддерживается большинство знакомых названий цветов (если только вам не довелось работать для компании Crayola[1]). Чтобы более точно определить значение цвета, в этих параметрах можно также передавать строки с шестнадцатеричными значениями – они должны начинаться с символа # и содержать значения насыщенности красного, зеленого и голубого цветов с одинаковым количеством битов для каждого. Например, строка '#ff0000' содержит по восемь битов для каждого цвета и определяет чистый красный цвет – «f» означает в шестнадцатеричном виде четыре единичных бита. Мы еще вернемся к шестнадцатеричному формату, когда встретимся с диалоговым окном выбора цвета далее в этой главе.

*Размер*

Для метки определяется точный размер в виде количества строк в высоту и символов в ширину путем установки параметров height и width. С помощью этих параметров можно увеличивать размеры метки по сравнению с теми, что устанавливаются менеджером компоновки по умолчанию.

*Шрифт*

В этом сценарии выбирается нестандартный шрифт для текста метки путем записи в параметр font кортежа из трех элементов, определяющих семейство шрифта, его размер и стиль (в данном случае: Times, 20 пунктов, полужирный). Стиль шрифта может принимать значения normal, bold, roman, italic, underline, overstrike и их сочетания (например, "bold italic"). Библиотека tkinter гарантирует возможность использования названий семейств шрифтов Times, Courier и Helvetica на всех платформах, однако в некоторых системах могут использоваться и другие (например, system – системный шрифт в Windows). Такие настройки шрифта будут действовать для всех виджетов, содержащих текст, например меток, кнопок, полей ввода, списков и Text (последний может одновременно выводить текст, отображаемый различными шрифтами, с помощью «тегов»). Параметр font сохраняет возможность определять шрифт с помощью более старых определений в стиле X Window – длинных строк с дефисами и звездочками, однако более новая форма определения параметров шрифта в виде кортежа более независима от платформы.

*Параметры компоновки*

Наконец, метка может быть сделана расширяемой и растягиваемой в целом путем установки параметров компоновщика pack, таких как expand и fill, с которыми мы познакомились в предыдущей главе: метка увеличивается вместе с окном. При распахивании окна чер-

[1] Всемирно известный производитель товаров для детского творчества, в том числе цветных карандашей, красок, мелков, фломастеров и пр. – *Прим. ред.*

ный фон заполняет весь экран, а желтый текст помещается в центр – можете попробовать.

В данном сценарии итоговым результатом настроек является метка, по своему внешнему виду коренным образом отличающаяся от тех, что мы создавали до сих пор. Она больше не следует стандартам внешнего вида Windows, но такая согласованность не всегда важна. Для справки отмечу, что библиотека tkinter предоставляет дополнительные способы настройки внешнего вида, не используемые в данном сценарии, но которые могут встретиться вам в других сценариях:

*Границы и рельефность*

Параметр `bd=N` виджета можно использовать для установки ширины границы, а параметр `relief=S` – ее стиля. *S* может принимать значения `FLAT`, `SUNKEN`, `RAISED`, `GROOVE`, `SOLID` или `RIDGE` – все эти константы экспортирует модуль `tkinter`.

*Курсор*

Параметр `cursor` позволяет определять внешний вид указателя мыши при наведении его на виджет. Например, `cursor='gumby'` изменяет стрелку на фигурку зеленого человечка. В число других имен указателей, часто используемых в этой книге, входят `watch`, `pencil`, `cross` и `hand2`.

*Состояние*

Некоторые виджеты поддерживают понятие состояния, влияющее на их внешний вид. Например, виджет с параметром `state=DISABLED` обычно рисуется на экране закрашенным (окрашивается в серый цвет) и делается неактивным. Значение `NORMAL` делает его обычным. Некоторые виджеты поддерживают также состояние `READONLY`, когда сам виджет отображается, как обычно, но он никак не откликается на попытки изменения.

*Отступы (padding)*

Вокруг многих виджетов (кнопок, меток и текста) можно добавить дополнительное пустое пространство с помощью параметров `padx=N` и `pady=N`. Интересно, что эти параметры можно определять и в вызовах метода `pack` (тогда пустое пространство добавляется вокруг виджета в целом), и в самом объекте виджета (в результате увеличивается сам графический элемент).

Чтобы проиллюстрировать некоторые из этих дополнительных настроек, в примере 8.2 создается кнопка, изображенная на рис. 8.2, и изменяется форма указателя мыши, когда он помещается над кнопкой.

*Пример 8.2. PP4E\Gui\Tour\config-button.py*

```
from tkinter import *
widget = Button(text='Spam', padx=10, pady=10)
widget.pack(padx=20, pady=20)
widget.config(cursor='gumby')
```

```
widget.config(bd=8, relief=RAISED)
widget.config(bg='dark green', fg='white')
widget.config(font=('helvetica', 20, 'underline italic'))
mainloop()
```


***Рис. 8.2.** Параметры кнопки в действии*

Чтобы увидеть эффект, создаваемый этими двумя параметрами, попробуйте поиграть с ними на своем компьютере. Большинству виджетов можно придать новый внешний вид таким же способом, и в этой книге мы будем неоднократно встречаться с такими параметрами. Мы встретимся также с операционными параметрами, такими как `focus` (передает фокуса ввода), и другими. У виджетов могут быть десятки параметров, большинство из которых имеет разумные значения по умолчанию, создающие принятый на каждой оконной платформе внешний вид, что является одной из причин простоты использования tkinter. Но при необходимости tkinter позволяет создавать значительно более индивидуальные изображения.

Дополнительные способы применения параметров настройки, обеспечивающих типичный внешний вид виджетов, можно увидеть в разделе «Настройка виджетов с помощью классов» в предыдущей главе и особенно в примерах `ThemedButton`. Теперь, когда вы больше знаете о настройках, вам будет проще понять, как настройки в этих примерах, выполняемые в подклассах виджетов, наследуются всеми экземплярами и подклассами. Новое расширение `ttk`, описываемое в главе 7, также реализует дополнительные способы настройки виджетов, вводя понятие тем оформления. Больше подробностей и ссылки на ресурсы, посвященные `ttk`, вы найдете в предыдущей главе.

## Окна верхнего уровня

Графические интерфейсы, построенные на базе tkinter, всегда имеют корневое окно, которое создается по умолчанию или явно с помощью конструктора объекта `Tk`. Это главное корневое окно открывается

при запуске программы и обычно служит для размещения наиболее важных виджетов. Помимо этого окна сценарии, использующие библиотеку tkinter, могут порождать любое число независимых окон, которые создаются и открываются по требованию, в результате создания объектов виджетов `Toplevel`.

Каждый объект `Toplevel` порождает на экране новое окно и автоматически добавляет его в поток обработки цикла событий программы (для активации новых окон не нужно вызывать метод `mainloop`). В примере 8.3 создается корневое окно и два дополнительных окна.

*Пример 8.3. PP4E\Gui\Tour\toplevel0.py*

```
import sys
from tkinter import Toplevel, Button, Label

win1 = Toplevel() # два независимых окна
win2 = Toplevel() # являющихся частью одного и того же процесса

Button(win1, text='Spam', command=sys.exit).pack()
Button(win2, text='SPAM', command=sys.exit).pack()

Label(text='Popups').pack() # по умолчанию добавляется в корневое окно Tk()
win1.mainloop()
```

Сценарий `toplevel0` получает корневое окно по умолчанию (к которому прикрепляется метка `Label`, потому что для нее не указан родитель) и создает два самостоятельных окна `Toplevel`, которые появляются и действуют независимо от корневого окна, как показано на рис. 8.3.

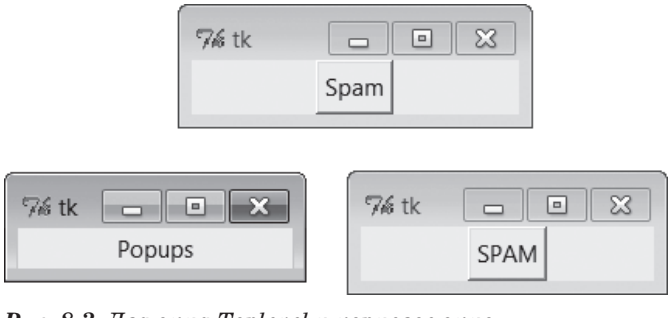


***Рис. 8.3.** Два окна Toplevel и корневое окно*

Два окна `Toplevel` в правой части являются полноценными окнами – они могут независимо сворачиваться, распахиваться на весь экран и так далее. Обычно окна `Toplevel` используются для реализации многооконных интерфейсов, а также модальных и немодальных диалогов (подробнее о диалогах рассказывается в следующем разделе). Они сохраняются до тех пор, пока не будут явно закрыты или пока создавшее их приложение не завершит работу.

Согласно реализации примера щелчок на кнопке с крестиком в правом верхнем углу любого из окон Toplevel закрывает только это окно. С другой стороны, щелчок на любой из кнопок или на крестике главного окна закроет все остальные и завершит программу (подробнее о протоколе завершения рассказывается чуть ниже).

Важно знать, что, хотя окна Toplevels и действуют независимо, они не являются независимыми процессами – если программа завершится, автоматически будут закрыты все ее окна, включая все окна Toplevel, которые она могла создать. Позднее будет показано, как обойти это правило путем запуска независимых программ с графическим интерфейсом.

## Виджеты Toplevel и Tk

Окно Toplevel напоминает Frame тем, что отщепляется в самостоятельное окно и обладает дополнительными методами, позволяющими работать со свойствами окна верхнего уровня. Виджет Tk в общем похож на виджет Toplevel, но используется для представления корневого окна приложения. Окна Toplevel имеют родителя, тогда как окно Tk – нет. Оно является настоящим корнем иерархии виджетов, создаваемых при конструировании графических интерфейсов с помощью библиотеки tkinter.

В примере 8.3 корневое окно Tk было получено даром, потому что для виджета Label был использован родитель по умолчанию, назначаемый при отсутствии первого аргумента в вызове конструктора:

```
Label(text='Popups').pack() # корневое окно Tk() по умолчанию
```

Передача значения None в первом аргументе конструктора виджета (или в именованном аргументе master) также приводит к назначению родителя по умолчанию. В других сценариях корневое окно Tk создается явно, например:

```
root = Tk()
Label(root, text='Popups').pack() # явное создание корневого окна Tk()
root.mainloop()
```

В действительности, из-за того, что графические интерфейсы tkinter строятся в виде иерархии, по умолчанию *всегда* создается хотя бы одно корневое окно Tk, явно, как в данном примере, или нет. Хотя это и не типично, тем не менее в приложении вручную может создаваться несколько корневых окон Tk, при этом программа завершается только после закрытия всех окон Tk. Первое созданное корневое окно Tk – явно, в программном коде, или автоматически, интерпретатором, – используется как родитель по умолчанию для виджетов и других окон, при создании которых родитель не указывается.

В целом корневое окно Tk должно использоваться для отображения какой-либо информации верхнего уровня. Если не прикрепить графические элементы к корневому окну, при запуске сценария оно будет вы-

ведено как странное пустое окно (часто это происходит из-за забывчивости, когда программист создает виджеты, использующие родителя по умолчанию, но забывает вызвать метод компоновщика, выполняющий размещение виджетов). Технически можно подавить создание корневого окна по умолчанию и создать несколько корневых окон с помощью виджета Tk, как показано в примере 8.4.

*Пример 8.4. PP4E\Gui\Tour\toplevel1.py*

```
import tkinter
from tkinter import Tk, Button
tkinter.NoDefaultRoot()

win1 = Tk()             # два независимых корневых окна
win2 = Tk()

Button(win1, text='Spam', command=win1.destroy).pack()
Button(win2, text='SPAM', command=win2.destroy).pack()
win1.mainloop()
```

Если запустить этот сценарий, он создаст только два окна, изображенные на рис. 8.3 (третье корневое окно не будет создано). Но чаще корневой объект Tk используется как главное окно, а виджеты Toplevel – как всплывающие окна приложения.

Обратите внимание: чтобы закрыть только одно окно, вместо функции sys.exit, которая завершает работу всей программы, вызывается метод destroy этого окна – чтобы понять, как действует этот метод, перейдем к изучению протоколов окна.

## Протоколы окна верхнего уровня

Виджеты Tk и Toplevel экспортируют дополнительные методы и свойства, предназначенные для той роли, которую они играют на верхнем уровне, что иллюстрируется примером 8.5.

*Пример 8.5. PP4E\Gui\Tour\toplevel2.py*

```
"""
Открывает три новых окна со стилями
метод destroy() закрывает одно окно, метод quit() закрывает все окна и завершает
приложение (прерывает работу функции mainloop);
окна верхнего уровня имеют заголовки, значки, могут сворачиваться
и восстанавливаться и поддерживают протокол событий wm;
приложение всегда имеет корневое окно, создаваемое по умолчанию или явно,
вызовом конструктора Tk(); все окна верхнего уровня являются контейнерами,
но они никогда не размещаются с помощью менеджера компоновки; объект Toplevel
напоминает фрейм Frame, но в действительности является новым окном и может иметь
собственное меню;
"""

from tkinter import *
```

```
root = Tk() # explicit root

trees = [('The Larch!',         'light blue'),
         ('The Pine!',          'light green'),
         ('The Giant Redwood!', 'red')]

for (tree, color) in trees:
    win = Toplevel(root)                                # новое окно
    win.title('Sing...')                                # установка границ
    win.protocol('WM_DELETE_WINDOW', lambda:None)       # игнорировать закрытие
    win.iconbitmap('py-blue-trans-out.ico')             # вместо значка Tk

    msg = Button(win, text=tree, command=win.destroy)  # закрывает одно окно
    msg.pack(expand=YES, fill=BOTH)
    msg.config(padx=10, pady=10, bd=10, relief=RAISED)
    msg.config(bg='black', fg=color, font=('times', 30, 'bold italic'))

root.title('Lumberjack demo')
Label(root, text='Main window', width=30).pack()
Button(root, text='Quit All', command=root.quit).pack() # завершает программу
root.mainloop()
```

Эта программа добавляет виджеты в корневое окно `Tk`, сразу выводит три окна `Toplevel` с прикрепленными к ним кнопками и использует специальные протоколы верхнего уровня. Если запустить этот пример, он создаст картинку, переданную в черно-белом изображении на рис. 8.4 (на мониторе текст кнопок отображается синим, зеленым и красным цветом).


***Рис. 8.4.** Три настроенных окна Toplevel*

Здесь следует отметить несколько деталей, касающихся функционирования, которые станут более заметными, если вы запустите сценарий на своем компьютере:

*Перехват закрытия:* `protocol`

Этот сценарий перехватывает событие закрытия окна менеджера окон с помощью метода виджета верхнего уровня `protocol`, поэтому при нажатии кнопки `X` в правом верхнем углу какого-либо из трех окон `Toplevel` ничего не происходит. Строка `WM_DELETE_WINDOW` обозначает операцию закрытия. С помощью этого метода можно запретить закрытие окон, кроме как из создаваемых в сценарии виджетов. Создаваемая этим сценарием функция `lambda: None` лишь возвращает значение `None` и больше ничего не делает.

*Закрытие одного окна (и его дочерних окон):* `destroy`

При нажатии на большую черную кнопку в любом из трех дополнительных окон закрывается только это окно, потому что это действие вызывает метод `destroy` виджета. Остальные окна продолжают существовать, как это свойственно диалоговым окнам. Технически вызов этого метода приводит к уничтожению соответствующего виджета и всех остальных виджетов, для которых он является родителем. В случае окон под этим подразумевается все их содержимое. В случае более простых виджетов метод `destroy` уничтожает сам виджет.

Вследствие того, что окна `Toplevel` имеют родителя, эти их отношения могут иметь последствия при применении метода `destroy`. Уничтожение окна, даже первого корневого окна `Tk`, созданного автоматически или явно, – которое является родителем по умолчанию, – приводит к уничтожению всех его дочерних окон. Так как корневые окна `Tk` не имеют родителя, на них никак не действует уничтожение других окон. При этом уничтожение последнего (или единственного) корневого окна `Tk` приводит к завершению программы. Окна `Toplevel` всегда уничтожаются при уничтожении родителя, но их уничтожение никак не влияет на другие окна, для которых они не являются предками. Это делает их идеальными для создания диалогов. Технически виджет `Toplevel` может быть дочерним по отношению к любому виджету и автоматически будет уничтожен вместе с родителем, однако обычно они создаются как потомки окна `Tk`, созданного явно или автоматически.

*Закрытие всех окон:* `quit`

Чтобы закрыть сразу все окна и завершить приложение с графическим интерфейсом (в действительности – активный вызов `mainloop`), кнопка корневого окна вызывает метод `quit`. То есть нажатие кнопки в корневом окне завершает работу приложения. Метод `quit` немедленно завершает приложение в целом и закрывает все его окна. Он может быть вызван относительно любого виджета tkinter, а не только относительно окна верхнего уровня – этот метод имеется также у фреймов, кнопок и других виджетов. Дополнительные подробности о методах `quit` и `destroy` вы найдете в обсуждении метода `bind` и его события `<Destroy>` далее в этой главе.

*Заголовки окон:* `title`

В главе 7 говорилось о методе `title` виджетов окон верхнего уровня (`Tk` и `Toplevel`), позволяющем изменять текст, выводимый в области верхней кромки окна. В данном случае в качестве текста заголовка окна устанавливается строка `'Si...'`, замещающая текст по умолчанию `'tk'`.

*Значки окон:* `iconbitmap`

Метод `iconbitmap` изменяет значок окна верхнего уровня. Он принимает значок или файл с растровым изображением и использует его в качестве графического значка окна, когда оно сворачивается и открывается. Если в Windows передать имя файла с расширением *.ico* (в данном примере используется такой файл, находящийся в текущем каталоге), он заменит значок по умолчанию с красными буквами «Tk», который обычно появляется в левом верхнем углу окна, а также в панели задач Windows. На других платформах вам может потребоваться использовать иные соглашения о файлах со значками, если вызов этого метода в наших примерах не дает желаемого результата (или просто закомментируйте вызов этого метода, если он приводит к аварийному завершению сценариев) – значки обычно являются платформозависимой особенностью, работа с ними зависит от используемого менеджера окон.

*Управление компоновкой*

Окна верхнего уровня служат контейнерами для других виджетов, подобно отдельному фрейму `Frame`. Однако в отличие от фреймов, виджеты – окна верхнего уровня – сами не компонуются (и не размещаются каким-либо другим менеджером компоновки). Для встраивания виджетов этот сценарий передает свои окна в аргументах, определяющих родительское окно, конструкторам меток и кнопок.

Имеется также возможность определять максимальный размер окна (физические размеры экрана в виде кортежа [ширина, высота]) с помощью метода `maxsize()` и устанавливать начальные размеры окна с помощью высокоуровневого метода `geometry("` *`width x height + x + y`* `")`. На практике гораздо проще и удобнее позволить библиотеке tkinter (или вашим пользователям) самой устанавливать размер окон, тем не менее размер экрана может пригодиться при выборе масштаба отображения изображений (смотрите обсуждение PyPhoto в главе 11, например).

Кроме того, графические элементы окон верхнего уровня поддерживают другие типы протоколов, которые будут позднее использованы в данном обзоре:

*Состояние*

Методы `iconify` и `withdraw` объектов окон верхнего уровня позволяют сценариям сворачивать или удалять окна на лету; метод `deiconify` перерисовывает свернутое или удаленное окно. Метод `state` возвра-

щает или изменяет состояние окна – допустимыми значениями, которые могут устанавливаться или возвращаться, являются: `iconic`, `withdrawn`, `zoomed` (в Windows: распахнутое на весь экран с помощью `geometry` или какого-либо другого метода) и `normal` (достаточно большого размера, чтобы вместить все содержимое). Методы `lift` и `lower` поднимают или опускают окно относительно других (метод `lift` является аналогом команды `raise` библиотеки Tk, которое является зарезервированным словом в языке Python). Их использование демонстрируется в сценариях будильника в конце главы 9.

*Меню*

Каждое окно верхнего уровня может иметь собственное меню – виджеты `Tk` и `Toplevel` принимают параметр `menu`, который используется для подключения горизонтальной строки меню с открывающимися списками элементов выбора. Эта строка меню выглядит соответствующим образом на каждой платформе, где выполняется сценарий. Меню будут изучаться в начале главы 9.

Большую часть методов окна верхнего уровня, используемых для взаимодействия с менеджером окон, можно также вызывать под именами с префиксом «wm_». Например, методы `state` и `protocol` можно также вызвать как `wm_state` и `wm_protocol`.

Обратите внимание, что в примере 8.3 при вызове конструктора `Toplevel` ему явно передается родительский виджет – корневое окно `Tk` (то есть `Toplevel(root)`). Окна `Toplevel` можно связывать с родительскими, как и любые другие виджеты, хотя зрительно они не встраиваются в родительские окна. Такой способ написания сценария имел целью избежать одной, на первый взгляд странной, особенности. Если бы окно создавалось так:

```
win = Toplevel()                        # новое окно
```

и корневое окно `Tk` при этом еще не существовало бы, этот вызов создал бы корневое окно `Tk` по умолчанию, которое стало бы родителем для окон `Toplevel`, как при всяком другом вызове графического элемента без передачи аргумента со ссылкой на родителя. Проблема в том, это делает принципиальным местоположение следующей строки:

```
root = Tk()                             # явное создание корня
```

Если поместить эту строку выше вызовов конструктора `Toplevel`, она создаст одно корневое окно, как и предполагается. Но если поставить эту строку ниже вызовов `Toplevel`, то tkinter создаст корневое окно `Tk`, которое будет отлично от созданного сценарием при явном вызове `Tk`. Это приведет к созданию двух корневых окон `Tk`, как в примере 8.4. Переместите вызов `Tk` под вызовы `Toplevel`, перезапустите сценарий, и вы увидите, что я имею в виду – вы получите четвертое, совершенно пустое окно! Чтобы избежать таких странностей, возьмите за правило создавать корневые окна `Tk` в начале сценариев и явным образом.

Все интерфейсы протоколов верхнего уровня доступны только в виджетах окон верхнего уровня, но часто доступ к ним можно получить через атрибут `master` виджета, хранящего ссылку на родительское окно. Например, изменение заголовка окна, в котором содержится фрейм, можно реализовать так:

```
theframe.master.title('Spam demo') # master является окном-контейнером
```

Естественно, делать так можно только при уверенности, что фрейм будет использован только в одном типе окна. Например, прикрепляемые компоненты общего назначения, реализуемые в виде классов, должны оставить право на установку свойств окон за своим приложениями-клиентами.

Для виджетов верхнего уровня существуют другие инструменты, некоторые из которых могут не встретиться в этой книге. Например, в менеджерах окон Unix можно также устанавливать имя значка окна (`iconname`). Поскольку некоторые параметры значков можно применять только в сценариях, выполняемых в Unix, подробности, касающиеся этой темы, смотрите в других ресурсах по библиотекам Tk и tkinter. А сейчас нас ожидает следующая запланированная остановка в нашей экскурсии, где будет рассказано об одном из наиболее частых применений окон верхнего уровня.

# Диалоги

Диалоги – это окна, выводимые сценарием с целью показать или запросить дополнительную информацию. Существует два вида диалогов: модальные и немодальные:

*Модальные*

Эти диалоги блокируют остальную часть интерфейса, пока окно диалога не будет закрыто – выполнение программы будет продолжено после получения диалогом ответа пользователя.

*Немодальные*

Эти диалоги могут оставаться на экране неопределенное время, не создавая помех другим окнам интерфейса, – обычно они в любой момент могут принимать входные данные.

Независимо от модальности диалоги обычно реализуются с помощью объекта окна `Toplevel`, с которым мы познакомились в предыдущем разделе, создаете вы `Toplevel` или нет. Существует три основных способа вывести диалог с помощью библиотеки tkinter: вызовом стандартных диалогов, обращением к современному объекту `Dialog` и путем создания пользовательских диалоговых окон с помощью `Toplevel` и других типов виджетов. Рассмотрим основы использования всех трех схем.

## Стандартные (типичные) диалоги

Вызовы стандартных диалогов проще, поэтому начнем с них. В составе библиотеки tkinter поставляется набор готовых диалогов, реализующих многие из наиболее часто встречающихся окон, генерируемых программами, – диалоги выбора файла, диалоги с сообщениями об ошибках и предупреждениями и диалоги, позволяющие запросить ввод данных. Они называются *стандартными диалогами*, поскольку входят в состав библиотеки tkinter и используют библиотечные вызовы для конкретных платформ, чтобы принять вид, свойственный данной платформе. Например, диалог открытия файла в библиотеке tkinter выглядит как любой другой подобный диалог в Windows.

Все стандартные диалоги являются модальными (они не возвращают управление, пока пользователь не закроет диалог) и блокируют главное окно программы. Сценарии могут настраивать окна этих диалогов, передавая текст сообщения, заголовки и тому подобное. Они очень просты в использовании, поэтому сразу перейдем к примеру 8.6 (который хранится в файле с расширением *.pyw*, чтобы подавить вывод окна консоли в Windows при запуске сценария щелчком мыши):

*Пример 8.6. PP4E\Gui\Tour\dlg1.pyw*

```
from tkinter import *
from tkinter.messagebox import *

def callback():
    if askyesno('Verify', 'Do you really want to quit?'):
        showwarning('Yes', 'Quit not yet implemented')
    else:
        showinfo('No', 'Quit has been cancelled')

errmsg = 'Sorry, no Spam allowed!'
Button(text='Quit', command=callback).pack(fill=X)
Button(text='Spam', command=(lambda: showerror('Spam', errmsg))).pack(fill=X)
mainloop()
```

Анонимная `lambda`-функция использована здесь в качестве оболочки вызова `showerror`, для передачи двух жестко определенных аргументов (напомню, что обработчики событий не получают аргументов от самой библиотеки tkinter). Если запустить этот сценарий, он создаст главное окно, изображенное на рис. 8.5.

Нажатие кнопки Quit в этом окне выводит диалог (рис. 8.6) – вызовом стандартной функции `askyesno` из модуля `messagebox`, входящего в состав пакета `tkinter`. В Unix и Macintosh этот диалог выглядит иначе, а в Windows выглядит, как показано на рисунке (на практике внешний вид диалога зависит от версии и настроек Windows – в моей системе Window 7 с настройками по умолчанию он выглядит несколько иначе, чем в Windows XP, как было показано в предыдущем издании).

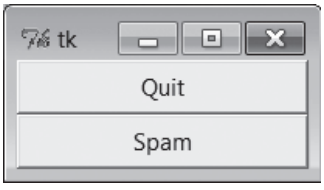


***Рис. 8.5.*** *Главное окно dlg1: кнопки вызывают появление дополнительных окон*

Диалог на рис. 8.6 блокирует программу, пока пользователь не щелкнет по одной из кнопок – при выборе кнопки Yes (или нажатии клавиши Enter) вызов диалога возвращает значение `True`, и сценарий выводит стандартный диалог `showwarning` (рис. 8.7), вызывая функцию `showwarning`.

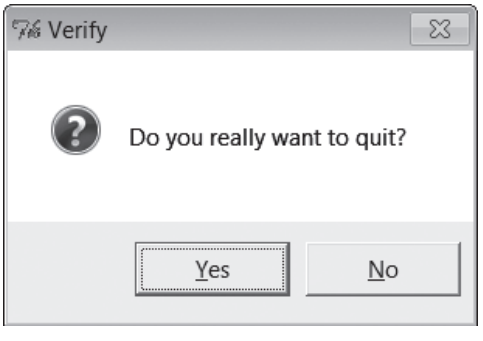


***Рис. 8.6.*** *Диалог askyesno, выводимый сценарием dlg1 (в Windows 7)*

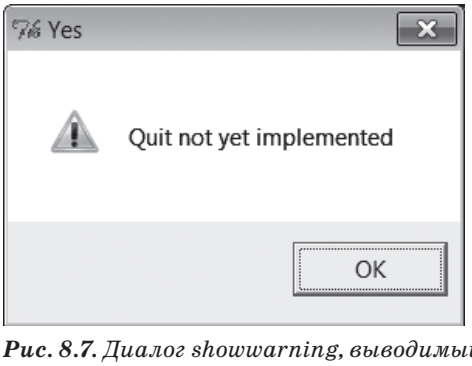


***Рис. 8.7.*** *Диалог showwarning, выводимый сценарием dlg1*

В диалоге на рис. 8.7 пользователь может только нажать кнопку OK. Если щелкнуть на кнопке No в диалоге на рис. 8.6, вызов `showinfo` создаст соответствующее окно диалога (рис. 8.8). Наконец, если в главном окне щелкнуть по кнопке Spam, то с помощью стандартного вызова `showerror` будет создан стандартный диалог `showerror` (рис. 8.9).

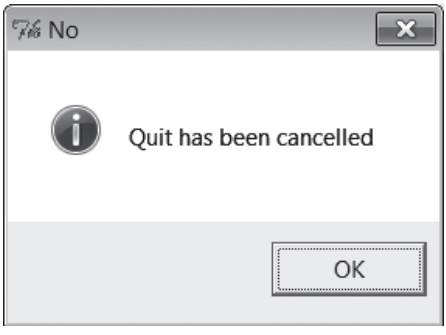


***Рис. 8.8.*** *Диалог showinfo, выводимый сценарием dlg1*

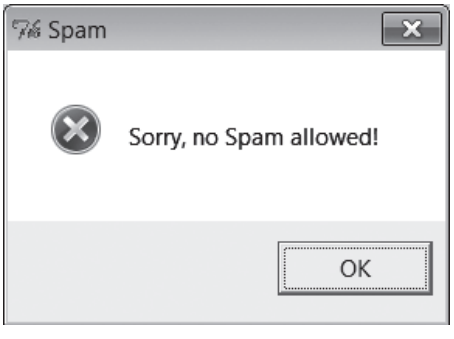


***Рис. 8.9.*** *Диалог showerror, выводимый сценарием dlg1*

Конечно, в результате создается множество всплывающих окон, и не следует злоупотреблять этими диалогами (обычно лучше применять окна с полями ввода, остающиеся на экране длительное время, а не отвлекать пользователя всплывающими окнами). Но в нужных случаях такие всплывающие диалоги сокращают время разработки и обеспечивают привычный внешний вид.

### «Умная» и многократно используемая кнопка Quit

Для некоторых из этих готовых диалогов можно найти лучшее применение. В примере 8.7 реализована прикрепляемая кнопка `Quit`, которая с помощью стандартных диалогов получает подтверждение в ответ на запрос о завершении. Поскольку она реализована в виде класса, ее можно прикреплять и повторно использовать в любом приложении, где требуется кнопка `Quit` с запросом на подтверждение. Так как в этой кнопке использованы стандартные диалоги, она должным образом выглядит на любой платформе.