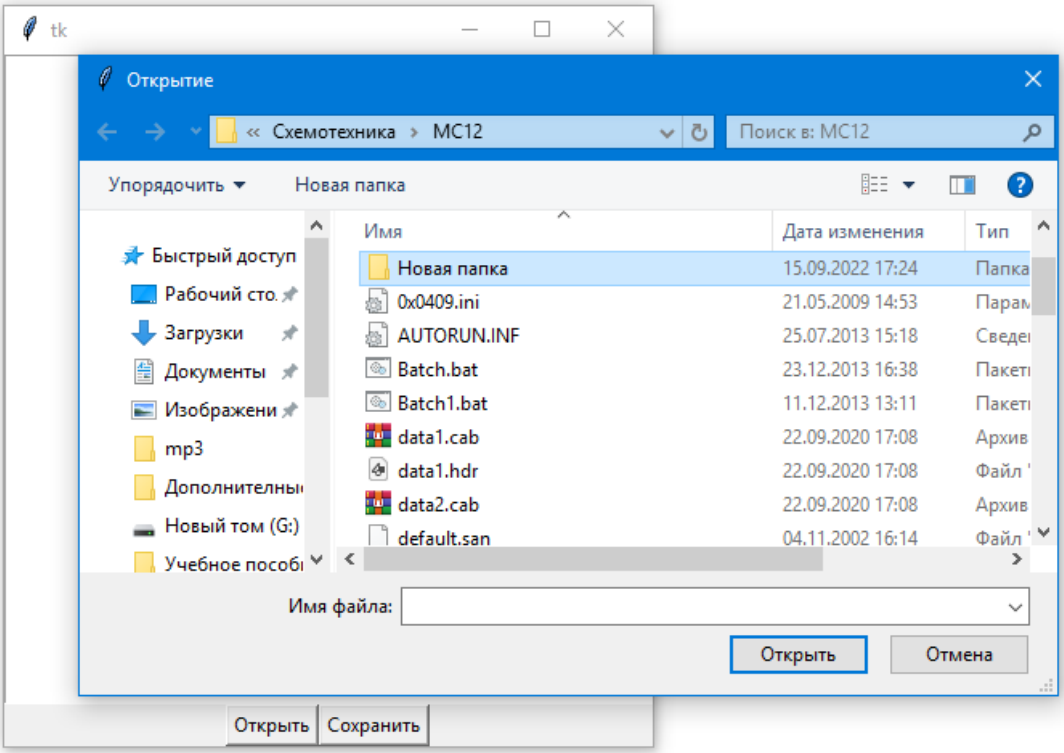
С. А. ВАСИЛЬЕВ, А. А. ЕВДОКИМОВ, А. Д. ОБУХОВ

ПРОГРАММНЫЙ ИНСТРУМЕНТАРИЙ АНАЛИЗА И ОБРАБОТКИ ГРАФИЧЕСКОЙ ИНФОРМАЦИИ



Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2023

Министерство науки и высшего образования Российской Федерации

**Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Тамбовский государственный технический университет»**

С. А. ВАСИЛЬЕВ, А. А. ЕВДОКИМОВ, А. Д. ОБУХОВ

ПРОГРАММНЫЙ ИНСТРУМЕНТАРИЙ АНАЛИЗА И ОБРАБОТКИ ГРАФИЧЕСКОЙ ИНФОРМАЦИИ

Утверждено Ученым советом университета в качестве учебного пособия
для студентов и магистрантов направлений подготовки
09.03.01, 09.04.01 «Информатика и вычислительная техника» дневной формы обучения

Учебное электронное издание

Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2023

УДК 004.9
ББК 3973.22
В19

Рецензенты:

Кандидат технических наук, доцент,
доцент кафедры «Математическое моделирование и информационные технологии»
Института математики, физики и информационных технологий
ФГБОУ ВО «ТГУ имени Г. Р. Державина»
Д. С. Соловьев

Кандидат технических наук,
доцент кафедры «Информационные процессы и управление»
ФГБОУ ВО «ТГТУ»
И. А. Елизаров

Васильев, С. А.

В19 Программный инструментарий анализа и обработки графической информации [Электронное издание] : учебное пособие / С. А. Васильев, А. А. Евдокимов, А. Д. Обухов. – Тамбов : Издательский центр ФГБОУ ВО «ТГТУ», 2023. – 1 электрон. опт. диск (CD-ROM). – Системные требования : ПК не ниже класса Pentium II ; CD-ROM-дисковод ; 1,95 Mb ; RAM ; Windows 95/98/XP ; мышь. – Загл. с экрана.

ISBN 978-5-8265-2612-5

Посвящено решению задач анализа и обработки информации с использованием языков высокого уровня. Представлены разделы по построению графического интерфейса на Python и Java. Рассмотрены основные библиотеки программного кода для работы с графическим интерфейсом. Приведены примеры создания типовых элементов интерфейса, взаимодействия с ними, выполнения операций по обработке графической информации.

Предназначено для студентов и магистрантов направлений подготовки 09.03.01, 09.04.01 «Информатика и вычислительная техника» дневной формы обучения.

УДК 004.9
ББК 3973.22

ISBN 978-5-8265-2612-5

ПРЕДИСЛОВИЕ

Эффективное изучение алгоритмов интеллектуального анализа графических изображений, а также графической визуализации 2D- и 3D-объектов предполагает их оперативное компьютерное моделирование на современных языках программирования. Обычно на момент изучения алгоритмов, связанных с обработкой изображений, студент обладает начальными знаниями в области программирования, что затрудняет в полном объеме исследовать изучаемые алгоритмы.

С чем же придется столкнуться студенту для апробации изучаемого материала? Обычно в подобных работах приходится организовать графический интерфейс для взаимодействия с самим изображением, а также с варьируемыми параметрами изучаемого алгоритма. Это и распределение окон ввода исходных данных с отработкой настроек режимов ввода. В качестве исходных данных могут быть как числовые значения, так текстовые. Кроме этого, в качестве исходных данных возможны готовые статические и динамические изображения в виде файлов, подготовленные в одном из стандартных графических форматов. Иногда требуется вывести конкретные пикселы изображения в определенные места окна вывода. Часто требуется и обратная задача – считать с экрана конкретные пикселы.

Даже ввод исходных данных порой является нетривиальной задачей для студента с точки зрения скорости разработки приложения. Ведь это может быть ввод чисел, текстов или всевозможных файлов данных.

Да и реализация интерактивного ввода точек в рабочее окно вывода графики с помощью “мышки” порой вызывает затруднение у студентов. Если мы коснулись манипулятора – “мышь”, то и тут возникают различные задачи не только по вводу исходных данных в виде точек на экран, но и их редактирование: выделение, добавление, удаление, перемещение и т.п.

Начальная подготовка студентов в области языков программирования до поступления в университет не одинаковая. И отработка на практике алгоритмов интеллектуального анализа графических изображений в рамках изучаемых специализированных дисциплин порой вызывает затруднения не в понимании самих алгоритмов, а в программном инструментарии при их реализации. Данное учебное пособие предназначено для помощи студентам при формировании графических интерфейсов в программах на языках программирования: Python и Java. Пособие должно послужить оперативным помощником студента при освоении прикладных алгоритмов обработки 2D- и 3D-изображений.

1. ЯЗЫК PYTHON

1.1. ГРАФИЧЕСКИЙ ИНТЕРФЕЙС ПОЛЬЗОВАТЕЛЯ В ЯЗЫКЕ PYTHON

Для организации графического интерфейса пользователя – GUI (graphical user interface) в языке **Python** обычно используют кросс-платформенную библиотеку **Tkinter.** Эта библиотека является стандартной библиотекой языка **Python**. Визуальные элементы библиотеки **Tkinter** отображаются через собственные элементы текущей операционной системы, поэтому приложения, созданные с помощью **Tkinter**, выглядят так, как будто они принадлежат той платформе, на которой они работают. Если требуется изменить дизайн графического интерфейса, то лучше использовать, например, библиотеку **PyQt5**.

При написании GUI необходимо придерживаться следующего списка этапов:

1. Создать главное окно.
2. Создать виджеты и выполнить конфигурацию их свойств (опций).
3. Определить события, то есть то, на что будет реагировать программа.
4. Описать обработчики событий, то есть то, как будет реагировать программа.
5. Расположить виджеты в главном окне.
6. Запустить цикл обработки событий.

Под виджетом будем понимать не просто какой-то графический элемент окна, а функционально исполняемый графический элемент. Можно сказать, что виджет – это маленькая программа с графическими атрибутами, которая выполняет ту или иную функцию. Подробнее о виджетах будет описано ниже.

1.2. ОРГАНИЗАЦИЯ ГЛАВНОГО ОКНА

Обычно программа GUI организует главное окно приложения. Для организации простейшего окна на экране монитора компьютера можем воспользоваться следующим скриптом, например, как:

```
from tkinter import *

root = Tk()
root.title("Заголовок окна")
root.geometry("400x200")

root.mainloop()
```

Здесь **from tkinter import** * импорт всего содержимого библиотеки **tkinter.**

Объект окна верхнего уровня создается от класса **Tk** модуля **tkinter**. Переменную, связываемую с объектом, часто называют **root** (корень): **root = Tk()**. Созданное окно будет закреплено за переменной **root**, и через эту переменную мы в дальнейшем сможем изменять атрибуты окна. Например, с помощью метода **title()** можно установить заголовок окна. В программе это выглядит, как **root.title("Заголовок окна")**.

С помощью метода **geometry()** мы можем управлять размером окна. Для установки размера в метод **geometry()** передается строка в формате "ШиринаxВысота". В приведенном примере размер окна задан как 400×200 – **root.geometry("400×200").**

Если необходимо организовать окно с размером, определяемым выводимым изображением, то метод **geometry()** опускается в программе. То есть метод **geometry()** вообще отсутствует в тексте программы.

Чтобы окно показалось на экране монитора необходимо отработать метод **mainloop()**, который запускает цикл обработки событий окна для взаимодействия с пользователем. Конкретная реализация показа окна выглядит как **root.mainloop().** В результате при запуске скрипта мы увидим такое пустое окно, как на рис. 1.1.

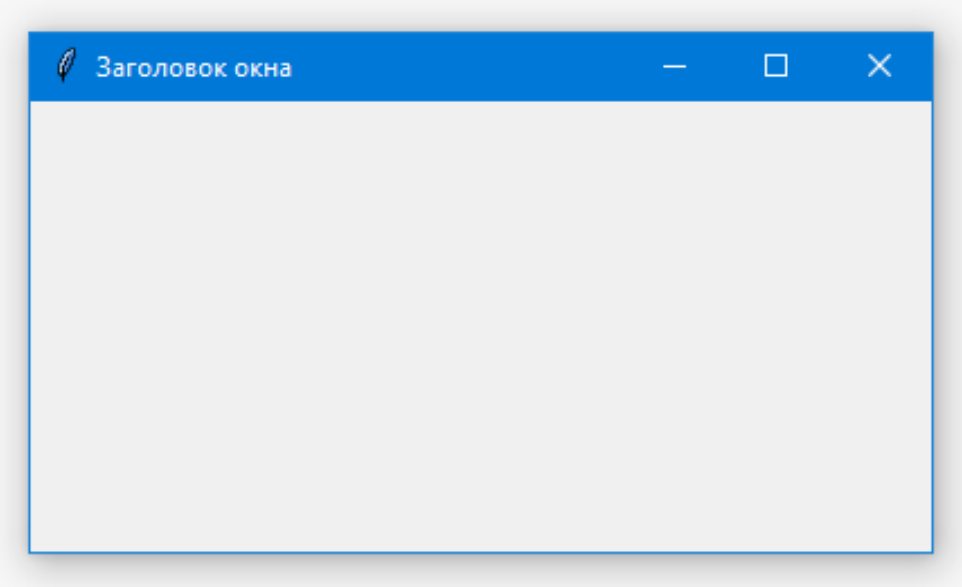


Рис. 1.1. Пустое окно

1.3. ВИДЖЕТЫ

Рассмотрим наиболее используемые виджеты библиотеки **tkinter** для исследования алгоритмов компьютерной графики. Дело в том, что по одному названию бывает тяжело понять функционал виджета. А внешний вид порой указывает на его возможности. После того, как рассмотрим внешний вид виджетов, далее погрузимся в их программирование. Часто используемые виджеты представлены в табл. 1.1.

Таблица 1.1

Виджет	Функционал
`Label`	Строка (или несколько строк) текста без возможности редактирования
`Button`	Кнопка
`Entry`	Поле ввода одной строки данных
`Text`	Интерактивный ввод многострочного текста
`Radiobutton`	Переключатель группы параметров
`Checkbutton`	Независимые переключатели в группе
`Listbox`	Выбор пункта из предлагаемого списка
`Spinbox`	Выбор в однострочном окне по двум кнопкам значения из списка
`Frame`	Механизм организации группы виджетов внутри окна
`Menu`	Выпадающие пункты списка под словами
`Scale`	Представление пользователю выбора какого-то значения из определенного диапазона
`Canvas`	Для формирования объекта-холст, необходимого при рисовании графики

1.3.1. Модуль Label

Текст “**Надпись**” в окне – это и есть результат работы виджета `Label`.

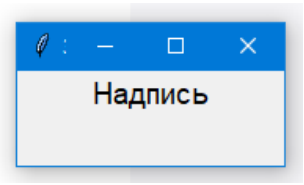


Рис. 1.2. Виджет `Label`

Для формирования подобной надписи необходимо организовать объект `lab = Label(root, text="Надпись")`. Создание объекта виджета еще не означает, что в окне приложения появится сам виджет. Для появления виджета в окне необходимо отработать менеджер размещения (подробнее о нем будет изложено ниже) и отработать метод `mainloop` экземпляра `Tk`, что является главным циклом обработки событий.

У класса `Label` много зарезервировано визуальных параметров. Остановимся на некоторых. Например, параметр `fg` (`foreground`) задает цвет текста, `bg` (`background`) задает фоновый цвет, `font` – шрифт текста. Значение цвета можно вводить в текстовой форме, например `fg=”Red”` или `fg=”#f00”`. Иногда более удобно вводить цвет в числовом формате для более тонкой настройки. Цвет задается в формате **RGB**. Каждая составляющая цвета из-

меняется в диапазоне **0-15**. Но так как в данном формате используются шестнадцатеричные числа, то диапазон изменения яркостей основных цветов сосредоточен в диапазоне **0-f**. Тогда, например, если нам необходим красный цвет максимальной яркости, то в качестве первой цифры мы вводим в шестнадцатеричной системе максимальное значение – **f**, а для зеленого и синего цвета **0**. И получится **fg=”#f00”** – красный цвет.

Если распределить в окне виджет **Label** с настройкой:

```
lab = Label(root, text="Надпись", bg="#0f0",fg="#f00", font=("Arial",14)),
```

то в окне приложения мы увидим текст с шрифтом **Arial** и размером 14, красный текст и на зеленом фоне (рис. 1.3).

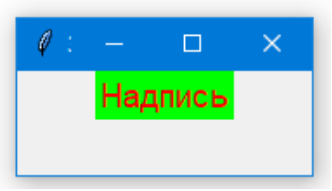


Рис. 1.3. Пример виджета Label с цветным текстом

Код приложения:

```
from tkinter import *

root = Tk()
root.title("Заголовок окна программы")
root.geometry("170x60")

lab=Label(root, text="Надпись", bg="#0f0", fg="#f00",
        font=("Arial",14))

lab.pack()
root.mainloop()
```

На рисунке 1.3 заголовок окна мы не видим из-за маленьких начальных размеров окна. Но как только мы растянем мышкой это окно, то заголовок появится.

1.3.2. Модуль Button

Объект-**Button** (кнопка) создается в программе вызовом класса **Button** модуля **tkinter**. Попробуем организовать одну кнопку с текстом “Ввод” в главном окне приложения, как на рис. 1.4.

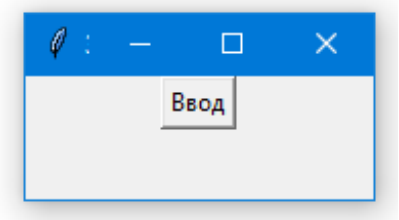


Рис. 1.4. Пример виджета Button

На языке Python это выглядит просто:

```
from tkinter import *

root = Tk()
root.title("Окно")
root.geometry("170x60")

but = Button(text="Ввод")
but.pack()

root.mainloop()
```

Здесь создается объект **but** из класса **Button** и в параметр **text** записываем строку "**Ввод**". После чего данный виджет встраивается в окно, и показывается на мониторе компьютера окно с кнопкой. Все аналогично, как для виджета **Label.** Стоит заметить, что в данном примере мы имеем одно окно – главное (**root**). И поэтому объект **root** можно не указывать в классе **Button.** А если организуется дочернее окно по отношению к главному, то для указания, что кнопка принадлежит этому новому окну, необходимо указывать в параметрах **Button** имя объекта дочернего окна.

Кнопка, как и любой виджет, имеет ряд параметров, которые влияют на ее визуализацию. Эти параметры мы можем настроить через конструктор **Button**. Например:

```
from tkinter import *

root = Tk()
root.title("Окно")
root.geometry("160x60")

but = Button(text="Ввод",       # текст кнопки
             background="#777", # фоновый цвет кнопки
             foreground="#ddd", # цвет текста
             font="14")         # высота шрифта
but.pack()

root.mainloop()
```

После интерпретации данного кода появится окно с кнопкой (рис. 1.5).

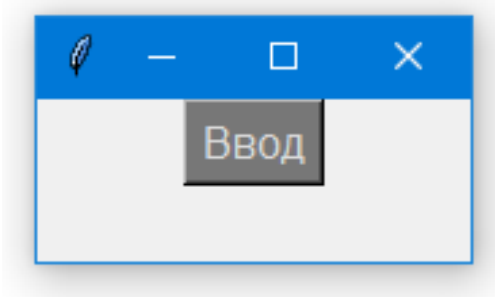


Рис. 1.5. Пример виджета Button с внешними изменениями

Визуализационных параметров виджета **Button** много. Рассмотрим некоторые из них. Вообще, конструктор **Button** может принимать следующие параметры: **Button(master,options). master** – ссылка на родительское окно (контейнер). Как пояснялось выше, так как мы работаем с главным окном, то эту ссылку можно не указывать. Второй параметр **options** – это набор параметров для отображения кнопки. Некоторые из этих параметров представлены в табл. 1.2.

Таблица 1.2

Параметры	Функционал
activebackground	Цвет кнопки при нажатом состоянии
activeforeground	Цвет текста в кнопке, когда кнопка нажата
bd	Толщина границы (по умолчанию **bd=2**)
bg/background	Фоновый цвет кнопки
fg/foreground	Цвет текста в кнопке
font	Шрифт текста кнопки, например – **font="Arial 12"**
height	Высота кнопки
highlightcolor	Цвет кнопки, когда она в фокусе (например, по кн. Tab)
image	Изображение (картинка) на фоне кнопки
justify	Выравнивание текста в поле кнопки. Значение LEFT выравнивает текст по левому краю, CENTER – по центру, RIGHT –по правому краю
padx	Отступ от границ кнопки до ее текста справа и слева
pady	Отступ от границ кнопки до ее текста сверху и снизу
state	Устанавливает состояние кнопки: DISABLED, ACTIVE и NORMAL (по умолчанию)
text	Текст кнопки
width	Ширина кнопки

Попробуем сформировать в главном окне приложения две кнопки с разными атрибутами визуализации (рис. 1.6). Обратите внимание, что высота кнопки и ширина задаются в количествах символов.

```
from tkinter import *

root = Tk()
root.title("Заголовок окна")
root.geometry("260x120")

but1 = Button(text="Кнопка 1",    # текст кнопки
```

```
                background="#777", # фоновый цвет кнопки
                foreground="#ddd", # цвет текста
                font="Arial 12",   # высота шрифта
                height=3,
                width=10)

but2 = Button(text="Кнопка 2",    # текст кнопки
                background="#0f0", # фоновый цвет кнопки
                foreground="#000", # цвет текста
                font="Arial 16",   # высота шрифта
                height=1,
                width=16)

but1.pack()
but2.pack()

root.mainloop()
```

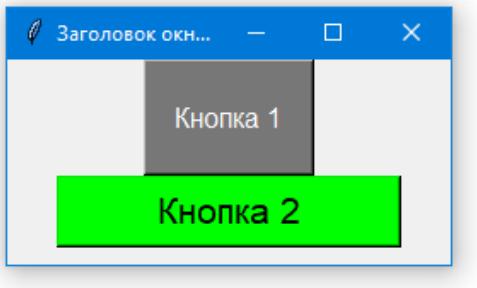


Рис. 1.6. Пример двух кнопок с разными атрибутами

Настроить кнопку можно и после ее создания. Для этого существует метод **configure** (или **config**). Например, так:

```
from tkinter import *

root = Tk()
root.title("Заголовок окна")
root.geometry("100x40")

but = Button(root) # создаем виджет
but.configure(text="Кнопка")
but.pack()

root.mainloop()
```

Основная функция кнопки – это отработка “нажатия”, т.е. виджет должен быть завязан на отработку внешних событий и соответствующих реакций на эти события. Например, клик по кнопке левой клавишей мыши и, как результат, вызов определенной функции в про-

грамме. Один из простых способов для отслеживания “нажатия” на кнопку – это использование параметра `command` в конструкторе кнопки. Для этого в программе объявляем, например, функцию **`clicked()`** и в ней реализуем реакцию на “нажатие”.

```
from tkinter import *

def clicked():
    but.configure(text="Нажали!") # изменяем текст кнопки

root = Tk()
root.title("Окно")
root.geometry("100x40")

but = Button(text="Нажми",
             font=("Roboto Bold", 14),
             command=clicked)
but.pack()

root.mainloop()
```

После запуска программы мы видим кнопку, как на левом скриншоте, и после нажатия на кнопку сменится название кнопки (справа) (рис. 1.7). Изменение текста не происходит во время нажатия кнопки. Только после “отпускания” кнопки сработает **`clicked()`**.

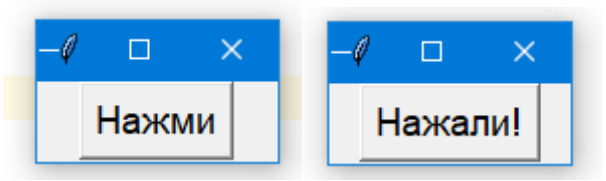


Рис. 1.7. Пример двух кнопок в разных состояниях

Если в приведенном примере используется одно событие – клик левой клавишей мыши через параметр **`command`**, то для более гибкого управления взаимодействием виджетов, событий и ответных действий на события используется метод **`bind`** библиотеки **`tkinter.`** Синтаксис использования данного метода не сложный: **`bind(событие, имя процедуры[,”+”])`**.

События обычно записываются в угловых скобках. Часто используемые события:

- **`<Button-1>`** – клик левой кнопкой мыши;
- **`<Button-2>`** – клик средней кнопкой мыши;
- **`<Button-3>`** – клик правой кнопкой мыши;
- **`<Duble-Button-1>`** – клик левой кнопкой мыши;
- **`<Motion>`** – движение мыши;
- **`<Return>`**-клавиша Enter, если кнопка в фокусе.

Второй аргумент метода **bind** сообщает, какая функция будет вызываться при обработке события.

Третий необязательный аргумент – строка **"+"**, которая означает, что обработчик события добавляется к уже существующим обработчикам.

Пробуем использовать на практике метод **bind**. В окне разместим текст **Label** и кнопку **Button**. Нажмем левой кнопкой мыши на кнопку и увидим, что изменится текст в кнопке и его цвет. Если правой кнопкой нажать на текст окна, то текст изменится и изменится его цвет.

```
from tkinter import *

def change_1(event):
    b["text"] = "Нажали!"
    b["fg"] = "red"

def change_2(event):
    l["text"] = "Правая клавиша"
    l["fg"] = "red"

root = Tk()
root.title("Окно")

l=Label(text="Надпись",
        bg="#0f0",
        fg="#f00")

b=Button(text="Нажми",
         fg="#000",
         width=10,
         height=3)

b.bind("<Button-1>", change_1)
l.bind("<Button-3>", change_2)

b.pack()
l.pack()

root.mainloop()
```

1.3.3. Модуль Entry

Виджет **Entry** представляет поле для ввода текста. Конструктор **Entry** принимает следующие параметры:

Таблица 1.3

Параметры	Функционал
bg	Цвет фона
fg	Цвет текста
font	Шрифт текста
bd	Толщина границы
cursor	Курсор указателя мыши при наведении на текстовое поле
justify	Выравнивание текста (LIFT, CENTER и RIGHT)
selectbackground	Цвет фона выделенного фрагмента текста
selectforeground	Цвет выделенного фрагмента текста
width	Ширина виджета

Из текстового поля виджета **Entry** можно взять текст. За это действие отвечает метод **get()**. В текстовое поле можно вставить текст методом **insert(index, str)**, где **index** – позиция вставки, а **str** – строка текста для вставки. Так же можно удалить текст методом **delete(first, last=None)**, где **first** – индекс позиции, с которого удаляются символы до позиции **last**. Если требуется удалить символы до конца строки, то записывают **END**. Например, **delete(4,END)**.

Рассмотрим маленький пример использования виджета **Entry**.

```
from tkinter import *

def end_to_lab(event):
    lab["text"] = ent.get()
    ent.delete(0, END)

root = Tk()
root.title("Окно приложения")
root.geometry("250x100")

ent = Entry(width=30)
but = Button(text="Переписать")
lab = Label(width=30, bg="black", fg="white")

but.bind("<Button>", end_to_lab)

ent.pack()
but.pack()
lab.pack()

root.mainloop()
```

Здесь в главном окне выставлены три виджета: строка ввода **Entry** длиной в 30 символов, кнопка **Button** с надписью “Переписать” и виджет **Label**. После ввода теста и нажатия кнопки в строке под кнопкой выведется текст из строки ввода. При этом строка ввода очистится.

1.3.4. Модуль Checkbutton

Это простой виджет – флажок, который может находиться в двух устойчивых состояниях: отмеченный или неотмеченный.

Конструктор **Checkbutton** может принимать следующие параметры:

Checkbutton(master,options)

В данном случае **master** – ссылка на родительское окно (контейнер). Как пояснялось выше, так как мы работаем с главным окном, то эту ссылку можно не указывать. Параметр **options,** на самом деле, это набор параметров для отображения данного виджета. Некоторые из этих параметров представлены в табл. 1.4.

Таблица 1.4

Параметры	Функционал
activebackground	Фоновый цвет флажка в нажатом состоянии
activeforeground	Цвет текста флажка в нажатом состоянии
bg	Фоновый цвет флажка
bd	Граница вокруг флажка
bitmap	Монохромное изображение для флажка
cursor	Курсор указателя мыши при наведении на элемент
justify	Выравнивание текста (**LIFT**, **CENTER** и **RIGHT**)
offvalue	Значение ассоциированной с флажком переменной **IntVar** в неотмеченном состоянии, по умолчанию равно 0
onvalue	Значение ассоциированной с флажком переменной **IntVar** в отмеченном состоянии, по умолчанию равно 1
command	Ссылка на функцию, которая вызывается при нажатии на флажок
font	Шрифт текста
fg	Цвет текста
disabledforeground	Цвет текста в состоянии **DISABLED**
height	Высота элемента
image	Графическое изображение, отображаемое на элементе
padx	Отступы слева и справа от текста до границы флажка
pady	Отступы сверху и снизу от текста до границы флажка
selectcolor	Цвет тела (квадратика) флажка
selectimage	Изображение на флажке, когда он находится в отмеченном состоянии
state	Состояние элемента: **NORMAL** (по умолчанию), **DISABLE** и **ACTIVE**
text	Текст элемента
variable	Ссылка на переменную, типа **IntVar**, в которой хранится состояние флажка
Width	Ширина виджета

Организуем один виджет **Checkbutton** в главном окне приложения. И пусть его числовое значение (0/1) будет показываться в виджете **Label** на этом же экране. Пример программы поставленной задачи:

```
from tkinter import *

root = Tk()
root.title("Окно")
root.geometry("200x60")

i = IntVar()

c = Checkbutton(text="День/Ночь", variable=i)
c.pack()

l = Label(textvariable=i)
l.pack()

root.mainloop()
```

Результат работы программы для двух состояний **Checkbutton** представлено на рис. 1.8.

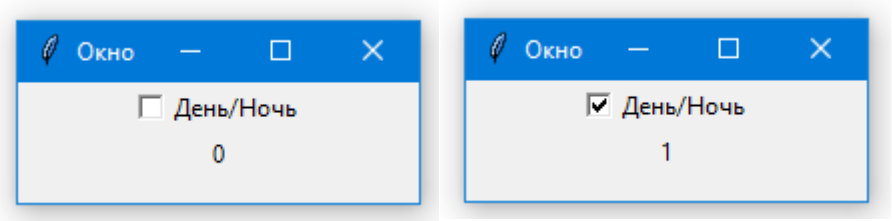


Рис. 1.8. Пример двух состояний виджета Checkbutton

В **Checkbutton** имеется возможность привязки к специальной компоненте **IntVar** через параметр **variable**. В нажатом состоянии связанный компонент **IntVar** имеет значение 1, а в не нажатом – 0. Эти значения берутся по умолчанию, но они могут быть переопределены на другие значения. Например, если нам необходимо при ненажатом состоянии получить значение **2**, а при нажатом – **3**. Для этого у виджета **Checkbutton** имеются параметры **onvalue** и **offvalue:**

c=Checkbutton(text="День/Ночь",variable=i,onvalue=3,offvalue=2).

В итоге через **IntVar** мы можем получать состояние **Checkbutton.**

Допустим, нам необходимо, чтобы виджет **Checkbutton** изначально выводился в окне в нажатом состоянии. Для этого мы должны воспользоваться методом **set** функции **IntVar()** из **Tkinter**, связанной с виджетом через переменную **variable**. Пример выглядит так:

```
i = IntVar()
i.set(1)
```

Прочитать состояние **Checkbutton** можно так: **value = i.get()**. В переменной value запишется число, соответствующее определенному состоянию виджета. По умолчанию неактивный **Checkbutton** выставит **i.get()** значение 0, а взведенное состояние – 1. Но эти значения можно изначально переопределить параметрами **offvalue** и **onvalue**. Обычно настраивают параметр **onvalue,** так как **offvalue** может иметь нулевые значения.

Если требуется вывести в окне несколько виджетов **Checkbutton**, например два, то достаточно сформировать два объекта c1 и c2 класса **Checkbutton** и вставить в окно (рис. 1.9):

```
from tkinter import *

root = Tk()
root.title("Окно")
root.geometry("200x100")

cam = IntVar()

c1 = Checkbutton(text="Камера 1", variable = cam, onvalue = 1)
c1.pack()

c2 = Checkbutton(text="Камера 2", variable = cam, onvalue = 2)
c2.pack()

root.mainloop()
```

Опросив значение **cam.get()**, можно определить состояние виджетов **Checkbutton**.

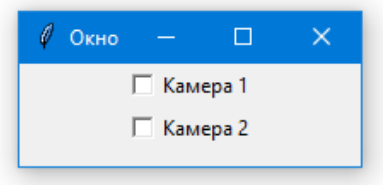


Рис. 1.9. Пример двух виджетов Checkbutton

В случае, когда требуется изначально активизировать, например, первый виджет, как на рис. 1.10, то для этого достаточно настроить активность первого виджета оператором **cam.set(1)**, вставив его в программу.

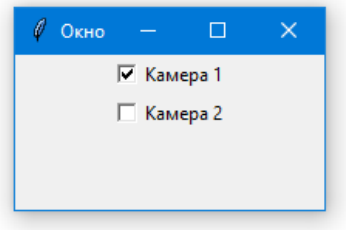


Рис. 1.10. Пример предустановки двух виджетов Checkbutton

```
cam = IntVar()
cam.set(1)
c1 = Checkbutton(text="Камера 1", variable = cam, onvalue = 1)
c1.pack()

c2 = Checkbutton(text="Камера 2", variable = cam, onvalue = 2)
c2.pack()
```

1.3.5. Модуль Radiobutton

Если Checkbutton – это простой автономный переключательный пункт в окне, т.е. не связанный с подобными виджетами своим состоянием, то **Radiobutton** – связанные в одну группу переключатели. Если какой-то пункт виджета **Radiobutton** взведен в активное состояние, то другие пункты в этой группе будут автоматически сброшены (рис. 1.11).

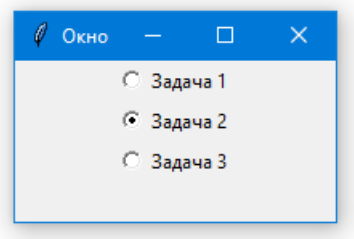


Рис. 1.11. Пример виджета Radiobutton

Код программы данного примера приведен ниже.

```
from tkinter import *

root = Tk()
root.title("Окно")
root.geometry("200x100")

var = IntVar()
var.set(2)

rad0=Radiobutton(root, text="Задача 1", variable = var, value = 1)
rad1=Radiobutton(root, text="Задача 2", variable = var, value = 2)
rad2=Radiobutton(root, text="Задача 3", variable = var, value = 3)

rad0.pack()
rad1.pack()
rad2.pack()

root.mainloop()
```

Обратите внимание, мы сразу активизировали второй пункт **Radiobutton**, отработав функцию **var.set(2)**.

Для настройки связанных переключателей конструктор **Radiobutton** принимает два параметра: **Radiobutton(master, options)**. Первый параметр – **master** – представляет ссылку на родительское окно, а **options** объединяет набор следующих параметров:

Таблица 1.5

Параметры	Функционал
activebackground	Фоновый цвет флажка в нажатом состоянии
activeforeground	Цвет текста флажка в нажатом состоянии
bg	Фоновый цвет переключателя
bitmap	Монохромное изображение для переключателя
borderwidth	Граница вокруг переключателя
command	Ссылка на функцию, которая вызывается при нажатии на данный пункт переключателя
cursor	Курсор указателя мыши при наведении на элемент
font	Шрифт текста
fg	Цвет текста
height	Высота эл-та в строках текста (по умолчанию равно 1)
image	Графическое изображение, отображаемое на элементе
justify	Выравнивание текста (**LIFT**, **CENTER** и **RIGHT**)
padx	Отступы слева и справа от текста до границы переключателя
pady	Отступы сверху и снизу от текста до границы переключателя
relif	Стиль переключателя (по умолчанию FLAT)
selectcolor	Цвет тела (кружка) переключателя
selectimage	Изображение на переключателе, когда он находится в отмеченном состоянии
state	Состояние элемента: **NORMAL** (по умолчанию), **DISABLE** и **ACTIVE**
text	Текст элемента
textvariable	Устанавливает привязку к переменной **StringVar**, которая задает текст переключателя
uderline	Индекс подчеркнутого символа в тексте элемента переключателя
variable	Ссылка на переменную, типа **IntVar**, в которой хранится состояние флажка
value	Значение переключателя
width	Ширина виджета

Возьмем некоторые параметры из табл. 1.5 и немного усложним программу с виджетами **Radiobutton**. Допустим, нам требуется выбрать из четырех цветов один для графического приложения (рис. 1.12). Пока не будем обращать внимание на качество размещения виджетов в окне. Для точного размещения виджетов в окне будут рассмотрены методы ниже.

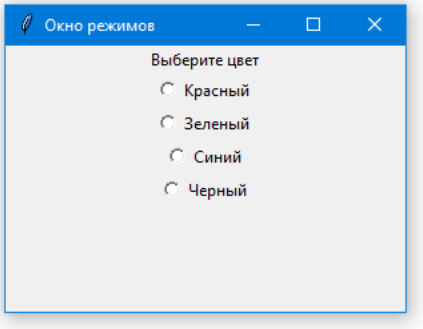


Рис. 1.12. Пример виджета Radiobutton для выбора цвета

Воспользуемся параметрами **Radiobutton**:

- **value** – для закрепления числового значения за пунктом переключателя;
- **variable** – для связи с переменной **IntVar**, в которой хранится состояние переключателя;
- **command** – для ссылки на функцию, которая вызывается при нажатии на данный пункт переключателя.

В цикле сформируем виджеты **Radiobutton** и разместим методом **pack()** их в окне приложения (рис. 1.13). Для подтверждения, что был выбран правильный цвет, в окне приложения разместим виджет **Label** и в эту строку будем записывать название выбранного цвета. Этим будет заниматься функция **Select()**.

```
from tkinter import *

Colors = [("Красный", 1), ("Зеленый", 2), ("Синий", 3), ("Черный", 4)]

def Select():
    l = Color.get()
    if l == 1:
        sel.config(text="Выбран красный")
    elif l == 2:
        sel.config(text="Выбран зеленый")
    elif l == 3:
        sel.config(text="Выбран синий")
    elif l == 4:
        sel.config(text="Выбран черный")

root = Tk()
root.title("Окно")
root.geometry("300x200")

header = Label(text="Выберите цвет")
header.pack()

Color = IntVar()

for c, val in Colors:
    r = Radiobutton(text = c, variable = Color, value = val, command = Select)
    r.pack()
sel = Label(padx = 15, pady = 10)
sel.pack()

root.mainloop()
```

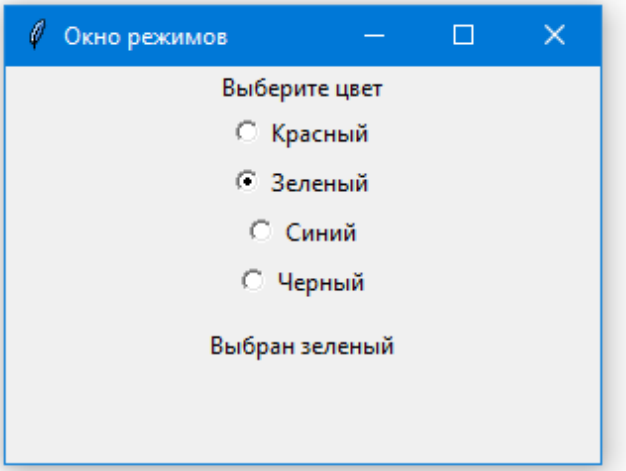


Рис. 1.13. Пример выбора цвета

1.3.6. Модуль Listbox

`Listbox` предназначен для работы со списком. Пользователю предоставляется возможность выбора из списка один (по умолчанию) или несколько пунктов (требуется настройка). Настроить виджет можно в конструкторе **`Listbox`**, используя параметры из табл. 1.6.

Таблица 1.6

Параметры	Функционал
`bg`	Фоновый цвет виджета
`bd`	Толщина границы вокруг элемента
`cursor`	Курсор указателя мыши при наведении на **`Listbox`**
`font`	Шрифт текста
`fg`	Цвет текста
`height`	Высота элемента в строках текста (по умолчанию равно 10 строк)
`highlightcolor`	Цвет элемента, когда он получает фокус
`highlightthickness`	Толщина границы элемента, когда он находится в фокусе
`relief`	Стиль элемента (по умолчанию **`SUNKEN`**)
`selectbackground`	Фоновый цвет для выделенного элемента
`selectmode`	Определяет, сколько элементов могут быть выделены. Может принимать следующие значения: **`BROWSE`**, **`SINGLE`**, **`MULTIPLE`**, **`EXTENDED`**. Например, если необходимо включить множественное выделение элементов, то можно использовать значения **`MULTIPLE`** или **`EXTENDED`**
`width`	Ширина элемента в символах (по умолчанию – 20 символов)
`xscrollcommand`	Задает горизонтальную прокрутку
`yscrollcommand`	Устанавливает вертикальную прокрутку

Методы **`Listbox`** представлены в табл. 1.7.

Таблица 1.7

Метод	Функционал
get(first,last=None)	Возвращает кортеж, который содержит текст элементов с индексами из диапазона **[first, last]**. Если второй параметр опущен, возвращается только текст элемента с индексом **first**
delete(first,last=None)	Удаляет элементы с индексами из диапазона **[first, last]**. Если второй параметр опущен, то удаляет только один элемент по индексу **first**
insert(index,element)	Вставляет в список элемент по заданному индексу
curselection()	Возвращает набор индексов выделенных элементов списка
size()	Возвращает количество элементов в списке виджета

Сформируем простой список и покажем его в виджете Listbox.

```
from tkinter import *

languages = ["Python", "JavaScript", "C#", "Java",
             "Ruby", "Kotlin", "Swoft", "SQL"]

root = Tk()
root.title("Языки программирования")
root.geometry("260x180")

languages_listbox = Listbox()
for language in languages:
        languages_listbox.insert(END, language)
languages_listbox.pack()

root.mainloop()
```

Здесь объект **language_listbox** создан из класса **Listbox**. В цикле **for** методом **insert** вставляем в виджет названия языков программирования из списка **languages.** Результатом работы данной программы будет окно со списком языков программирования. На скриншоте показано, что выделен мышкой язык “Ruby” (рис. 1.14).

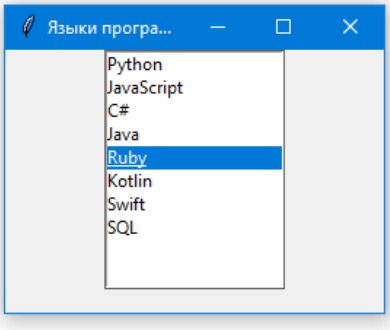


Рис. 1.14. Пример списка языков программирования

Если список большой и не все пункты его помещаются в окне виджета **Listbox**, то желательно воспользоваться элементом **Scrollbar**.

Рассмотрим пример работы виджета **Listbox**, когда требуется добавлять и удалять пункты в списке. Дополнительно в окно приложения введем поле для ввода текста для пункта и две кнопки: “Добавить” и “Удалить”.

```
from tkinter import *

def delete():
        selection = languages_listbox.curelection()
        languagea_listbox.delete(selection[0])

def add():
        new_language = language_entry.get()
        languagea_listbox.insert(0, new_language)


root = Tk()
root.title("Параметры  Listbox")
root.geometry("260x180")

language_entry = Entry(width=40)
language_entry.grid(column=0, row=0, padx=6, pady=6)

add_button = Button(text="Добавить", command=add)
add_button.grid(column=1, row=0, padx=6, pady=6)

languages_listbox = Listbox()
languages_listbox.grid(column=0, row=1, columnspan=2,
                       sticky=W+E, padx=5, pady=5)

#формируем начальный список параметров
languages_listbox.insert(END, "bg")
languages_listbox.insert(END, "dg")
languages_listbox.insert(END, "cursor")
languages_listbox.insert(END, "font")

delete_button = Button(text="Удалить", command=delete)
delete_button.grid(row=2, column=1, padx=5, pady=5)

root.mainloop()
```

Первая кнопка закреплена за функцией **add()**, которая внутри себя вызывает метод **insert** объекта **scrollbar.** Текстовые значения будут вставляться с первого места в списке. Для этого в первом параметре **insert** записано значение 0. Кнопка “**Удалить**” связана с функцией **delete()**. Внутри этой функции определяем индекс выделенного элемента списка

`selection = languages_listbox.curselection()`

и по данному индексу удаляем одну запись в списке

`languages_listbox.delete(selection[0])`.

На рисунке 1.15 представлен результат работы программы.

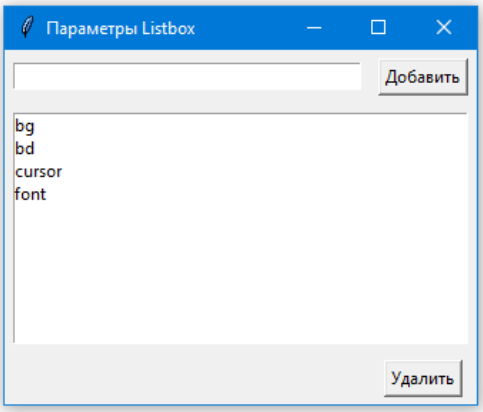


Рис. 1.15. Пример `Listbox` с добавлением и удалением элементов списка

1.3.7. Модуль Spinbox

В отличие от предыдущего виджета виджет **`Spinbox`** представляет пользователю выбрать одно из предложенных числовых или строковых значений. Значения для выбора появляются в однострочном окне и благодаря двум кнопкам. Кнопки необходимы для перемещения по скрытому списку предлагаемых значений. Пример виджета **`Spinbox`** представлен на рис. 1.16.

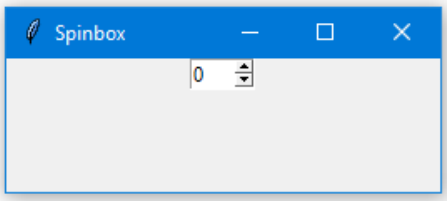


Рис. 1.16. Пример Spinbox

Код программы моделирования окна с **`Spinbox`**:

```
from tkinter import *

root = Tk()
root.title("Spinbox")
root.geometry("260x80")

spin = Spinbox(root, from_=0, to=255, width=4)
spin.pack()

root.mainloop()
```

Из программы видно, что в окне **Spinbox** кнопками вверх и вниз можно перебирать значения из диапазона 0-255.

Рассмотрим некоторые параметры **Spinbox** (табл. 1.8).

Таблица 1.8

Параметр	Функционал
textvariable	Связывается с метапеременной, хранящей значение, которое будет выводиться в виджете
from_	Минимальное значение в окне виджета
to	Максимальное значение в окне виджета
increment	Указывает шаг изменения значений в окне виджета
values	Указатель на список величин, который будет перебираться в виде кортежа строк
wrap	Управление зацикливанием. Если **False**, то зацикливания нет. Если **True**, то после достижения минимального значения в списке появится максимальное, и наоборот
state	Состояние виджета. **NORMAL** – доступное состояние (по умолчанию) и **DISABLED** – недоступное состояние
command	Связывает с функцией, которая будет вызываться при нажатиях на кнопки виджета
width	Ширина элемента в символах (по умолчанию – 20 символов)
font	Шрифт
fg	Цвет текста
bg	Фоновый цвет виджета
relief	Стиль элемента: **FLAT**, **RAISED**, **SUNKEN**, **RIDGE** и **GROOVE** (по умолчанию **SUNKEN**)
bd	Толщина рамки вокруг виджета (по умолчанию – 2)
cursor	Курсор указателя мыши при наведении на виджет **Spinbox**
height	Высота элемента в строках текста (по умолчанию равно 10 строк)
highlightcolor	Цвет элемента, когда он получает фокус
highlightthickness	Толщина границы элемента, когда он находится в фокусе
selectbackground	Фоновый цвет для выделенного элемента
buttonbackground	Цвет фона кнопок
repeatinterval	Временной интервал, после которого начнется автоматическое изменение значения предлагаемого списка при нажатой кнопке изменения (по умолчанию 400 миллисекунд)

Рассмотрим практическое применение виджета Spinbox. Допустим, имеется окно, в котором присутствуют два виджета – **Spinbox** и **Button**. И пусть мы желаем после выбора значения в **Spinbox** вывести это значение для контроля функцией **print**. Подобное приложение можно организовать следующим образом:

```
from tkinter import *

def print_values():
    print("Spinbox: {}".format(spin.get()))

root = Tk()
root.title("Spinbox")
root.geometry("260x80")

spin = Spinbox(root, from_=0, to=255, width=4)
spin.pack()

btn = Button(root, text="Вывести знаение", command=print_values)
btn.pack()

root.mainloop()
```

Обратите внимание, что в организованной функции **print_values()** значение из **Spinbox** выбирается функцией **spin.get().** А сама функция **print_values()** вызывается через параметр **command** конструктора **Button**. В итоге мы имеем следующее окно приложения, приведенное на рис. 1.17.

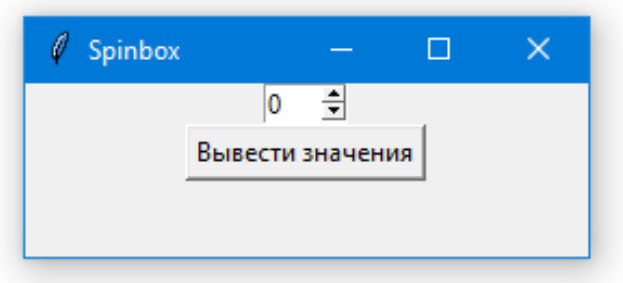


Рис. 1.17. Пример Spinbox с контрольной печатью

По умолчанию виджет **Spinbox** начинает показ первого значения согласно параметру **from_**. Если требуется изменить начальное показывающее значение, то можно воспользоваться методом **set** и метапеременной **IntVar()**. Ниже приводится пример, как можно переопределить начальное значение, например, **5** вместо **0** для показа. Следует заметить, что изменяется только показывающее значение, а не минимальное значение в списке для выбора.

```
from tkinter import *

def print_values():
    print("Spinbox: {}".format(spin.get()))

root = Tk()
root.title("Spinbox")
root.geometry("260x80")

var = IntVar()
var.set(5)

spin = Spinbox(root, from_=0, to=255, width=4)
spin.pack()

btn = Button(root, text="Вывести значение", command=print_values)
btn.pack()

root.mainloop()
```

Здесь `var = IntVar()` определяет метапеременную **var**, а в следующей строке в эту переменную записываем значение **5**.

1.3.8. Модуль Scale

Виджет **Scale** – это шкала (рис. 1.18). Пользователю предоставляется возможность виртуальным регулятором настроить необходимое значение, а не выбрать из списка, как в виджете **Spinbox**.

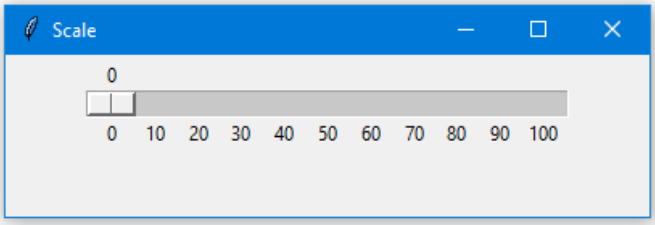


Рис. 1.18. Пример Scale

Нижеприведенная программа моделирует виджет **Scale**.

```
from tkinter import *

root = Tk()
root.title("Scale")
root.geometry("400x100")

sca = Scale(root, orient=HORIZONTAL, length=300,
            from_=0, to=100, tickinterval=10, resolution=5)
```

```
sca.pack()

root.mainloop()
```

Свойства шкалы определяются параметрами конструктора **Scale** (табл. 1.9).

Таблица 1.9

Параметр	Функционал
master	Родительский виджет
orient	Ориентация шкалы: **HORIZONTAL** или **VERTICAL**
font	Шрифт и размер текста
length	Длина шкалы в пикселях
from_	Начальное значение шкалы
to	Конечное значение шкалы
tickinteval	Интервал отображения меток шкалы
resolution	Минимальный шаг перемещения рычажка шкалы (по умолчанию 1)

Для работы с выбранным значением из шкалы воспользуемся методом **get()**, например, как в представленном примере.

```
from tkinter import *

root = Tk()
root.title("Scale")
root.geometry("400x100")

def print_values():
    print("Scale: {}".format(sca.get()))

sca = Scale(root, orient=HORIZONTAL, length=300,
            from_=0, to=100, tickinterval=10, resolution=5)

btn = Button(root,text="Вывести значения", command=print_values)

sca.pack()
btn.pack()

root.mainloop()
```

В программе, кроме шкалы, существует кнопка “**Вывести значение**”, которой вызывается функция **print_values**, где **sca.get()** “снимает” показание шкалы и выступает в роли аргумента функции **print**. Рабочее окно приложения имеет вид, представленный на рис. 1.19.

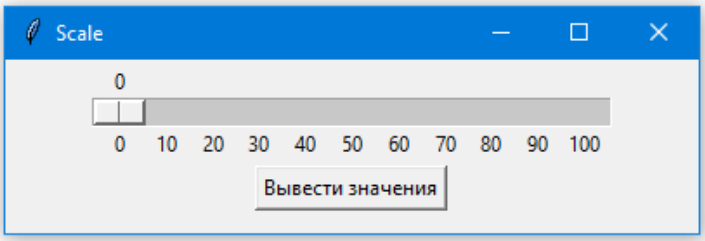


Рис. 1.19. Пример `Scale` с кнопкой `Button`

1.3.9. Модуль messagebox

На практике часто приходится встречаться с ситуацией, когда необходимо подтверждать правильность действий в полях виджетов. В пакете **`tkinter`** для этой цели имеется специальный модуль **`messagebox`**, имеющий в своем составе внутренние функции окон уведомления. Окна уведомления могут быть разные. Например, диалоговое окно с двумя кнопками: “**Да**” и “**Нет**” может быть получено функцией **`messagebox.askyesno()`**. Таким окном мы можем гарантировать правильность ввода данных в предшествующем виджете. Следует заметить, что **`messagebox.askyesno()`** формирует не только окно с двумя кнопками, но и графический логотип вопроса в окне. Рассмотрим, как все это происходит на примере. Но предварительно мы должны в программе импортировать модуль **`messagebox`**, кроме импортирования самого пакета **`tkinter`**.

Например, нам требуется в отдельном окне приложения ввести некоторые данные в поле **`label`**. И эти данные должны переслаться в основную программу по нажатию кнопки **`Button`** с текстом “**Передать**”. Но для подстраховки от неправильных данных (ошиблись при вводе) имеется возможность продублировать нажатие кнопки “**Передать**” отдельным вопросным окном с двумя кнопками – “**Да**” и “**Нет**”. И вот здесь нам поможет функция **`askyesno()`** модуля **`messagebox()`**. Данная функция выводит озаглавленное окно с вопросом, графический образ знака вопрос, текст самого вопроса и две кнопки – “**Да**” и “**Нет**”. При этом кнопка “**Да**” уже активна для выбора. Код программы с подобным алгоритмом работы приведен ниже:

```
from tkinter import *
from tkinter import messagebox as mb

def check():
    ar = mb.askyesno(title="Вопрос!", message="Передать данные?")
    if ar:
        s = entry.get()
        entry.delete(0, END)
        label["text"] = s

root = Tk()
```

```
root.title("Ввод данных")
root.geometry("250x100")
entry = Entry()
entry.pack(pady=10)
Button(text="Передать", command=check).pack()

label = Label(height=3)
label.pack()

root.mainloop()
```

Кнопка “**Передать**” закрепляет факт нажатия на кнопку с функцией **check()**. В случае вызова этой функции появляется окно с заголовком “**Вопрос!**”, с внутренним текстом “**Передать данные?**” и с двумя кнопками: “**Да**” и “**Нет**”. В случае нажатия на кнопку “**Да**” объект **ar** принимает значение **True** и поэтому срабатывает условный оператор в функции **check()**. В условном операторе **if** считывается в переменную **s** введенные данные из поля **Entry**, очищается это поле и данные переносятся в виджет **label**. Последнее действие не обязательное, но для обучающего примера это сигнал, что действительно введенные данные передаются в программу. Результат подобного диалога представлен на рис. 1.20.

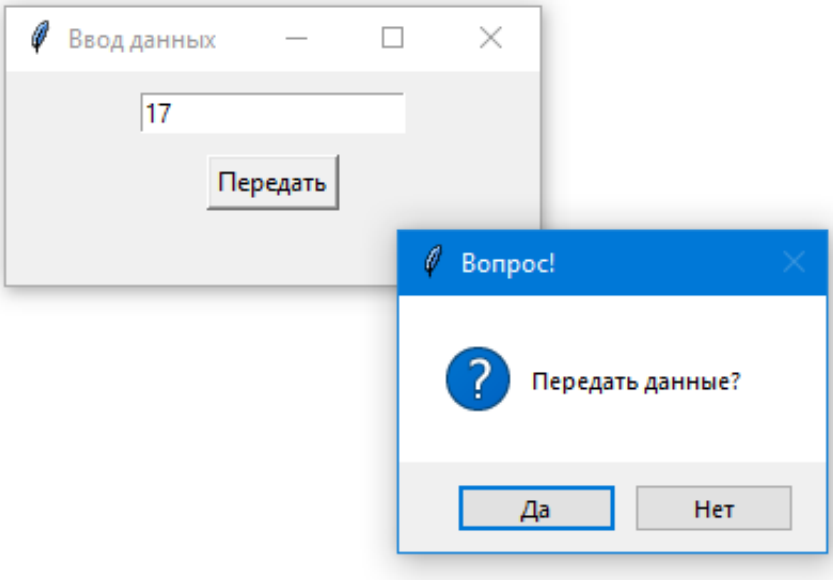
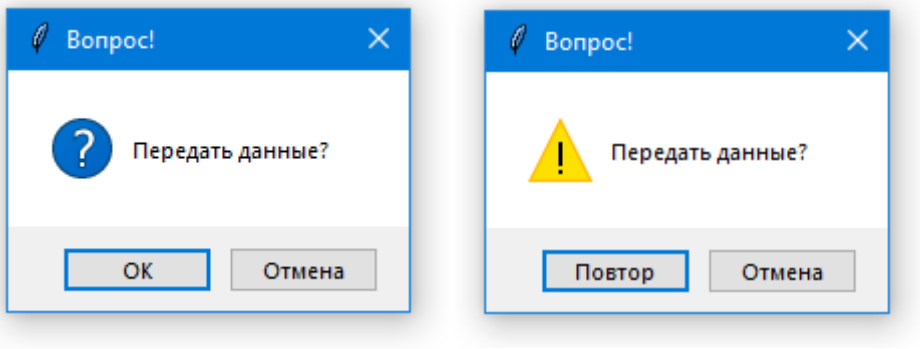


Рис. 1.20. Пример диалогового окна “Да-Нет”

Кроме функции **askyesno()**, модуль **messagebox()** содержит и другие функции окон выбора или уведомлений. Так, наполненная параметрами функция **messagebox.askokcancel(title=”Вопрос!”, massage=”Передать данные?”)** формирует окно, как на рис. 1.21, *а*.

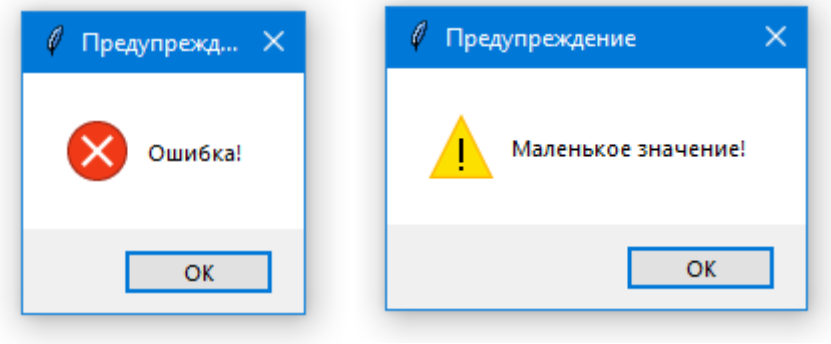
Функция **messagebox.askretrycancel(title=”Вопрос!”, massage=”Передать данные?”)** уже формирует другое окно (рис. 1.21, *б*).



а) *б*)

Рис. 1.21. Пример диалогового окна “OK-Отмена” и “Повтор-Отмена”

Окно с предупреждением “Ошибка!” (рис. 1.22, *а*) формируется вызовом функции **`messagebox.showerror(title=”Предупреждение”, message=”Ошибка!”)`**.



а) *б*)

Рис. 1.22. Пример диалогового окна “Ошибка!” и “Предупреждение”

Можно сформировать и окно с предупреждением (рис. 1.22, *б*). Такое предупреждение реализуется функцией **`messagebox.showwarning(title=”Предупреждение”, message=”Маленькое значение!”)`**.

1.3.10. Модуль filedialog для работы с файлами

Неотъемлемой частью любого приложения является работа с каталогами внешней памяти компьютера для открытия или сохранения файла. Кажущаяся сложность организации подобного диалога в **`Python`** решается очень просто. Достаточно воспользоваться модулем **`filedialog`** из пакета **`tkinter`**. Для того чтобы указать, какой файл необходимо открыть в модуле **`trinter.filedialog`**, существует функция, понятная уже из аббревиатуры (ее название – **`askopenfilename`**). И для сохранения файла – **`askclosefilename`**. Каждая из этих функций возвращает имя файла, который будет открыт или сохранен. Тут следует сделать пояснение. Сами эти две функции не открывают и не сохраняют файлы, а лишь предоставляет стандартные диалоговые окна ОС для выбора файла, с которым необ-

ходимо работать. А процесс открытия и сохранения файла осуществляется стандартными средствами языка **Python**.

Рассмотрим пример, как на практике можно воспользоваться функциями **askopenfilename** и **askclosefilename** модуля **filedialog.** Создадим окно приложения, где в верхней части будем размещать виджет **Text** шириной 50 и высотой 25 символов. В это пространство мы сможем вводить свои данные с клавиатуры или выводить данные открытого файла. Более того, из виджета **Text** мы будем все данные записывать в выбранный файл. Внизу окна будут две кнопки: "**Открыть**" и "**Сохранить**". Если нажать на кнопку "**Открыть**", то появится стандартное диалоговое окно операционной системы для поиска файла. При нажатии на кнопку "**Сохранить**" сформируется стандартное диалоговое окно для записи файла. В поле "**Тип фала**" появятся три возможных вариантов: `txt`, `html` или `htm` и `*.*`.

```
from tkinter import *
from tkinter import filedialog as fd

def insert_text():
    file_name = fd.askopenfilename()
    f = open(file_name)
    s = f.read()
    text.insert(1.0, s)
    f.close()

def extract_text():
    file_name = fd.asksaveasfilename(
        filetypes=(("TXT files", "*.txt"),
                   ("HTML files", "*.html;*.htm"),
                   ("All files", "*.*")))
    f = open(file_name, 'w')
    s = text.get(1.0, END)
    f.write(s)
    f.close()

root = Tk()
text = Text(width=50, height=25)
text.grid(columnspan=2)
b1 = Button(text="Открыть", command=insert_text)
b1.grid(row=1, sticky=E)
b2 = Button(text="Сохранить", command=extract_text)
b2.grid(row=1, column=1, sticky=W)

root.mainloop()
```

За кнопкой **b1** закрепляется функция **insert_text**, где предоставляется диалоговое окно функцией **askopenfilename()**. Имя выбранного файла записывается в переменную **file_name**.

Запись файла реализована по нажатию кнопки **b2**, где вызывается функция **extract_text**. Вызов **askopenfilename()** определяет имя файла и где его записать. Для записи открывается файл строкой **f = open(file_name,'w')**. Вся информация из **text** заносится в объект **s** и записывается в файл.

На рисунке 1.23 представлено окно с открытием файла.

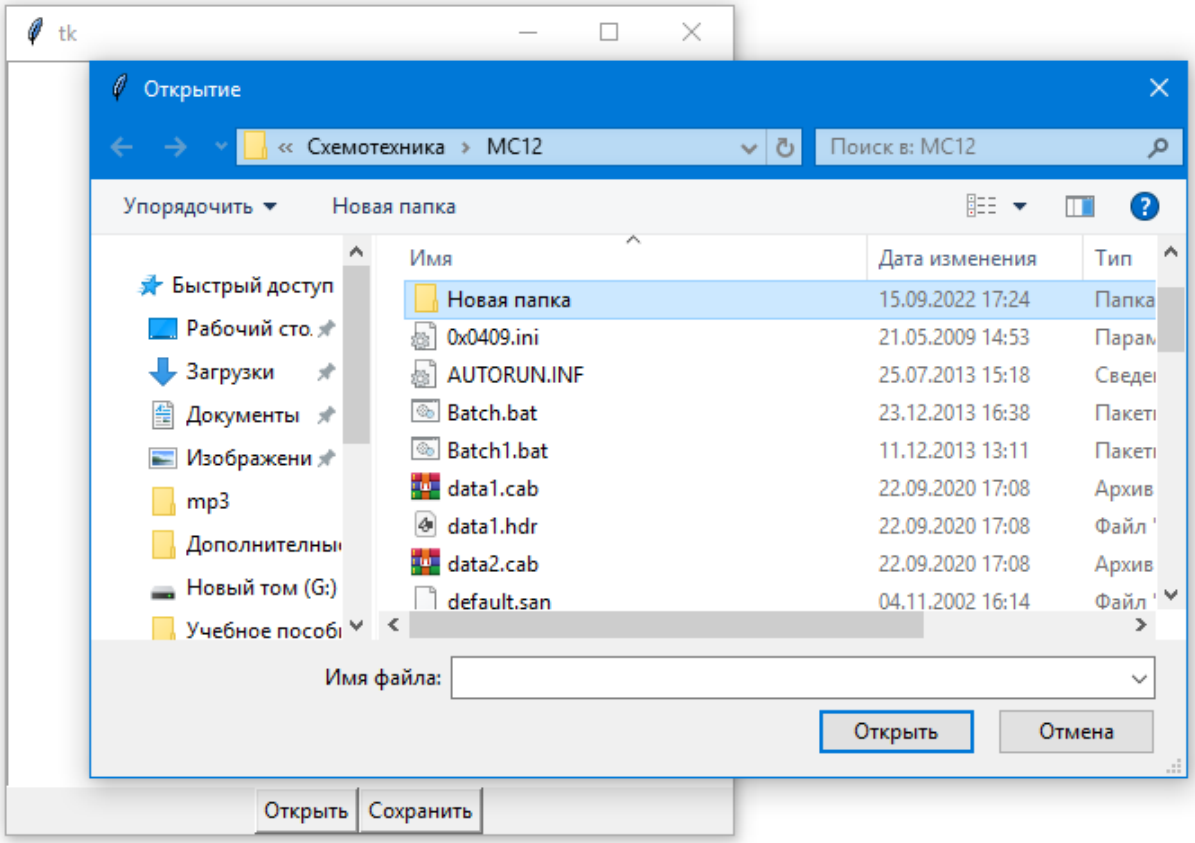


Рис. 1.23. Пример окна “Открытие файла”

1.3.11. Модуль Canvas для работы с графикой

В пакете **tkinter** имеется модуль **Canvas**, благодаря которому можно формировать компьютерную графику: отрезки прямых, кривые, дуги, эллипсы, прямоугольники и т.п. Для работы с графикой в программе из класса **Canvas** пакета **tkinter** создается объект – “холст”, на котором и будет формироваться изображение графических примитивов. Например, так: **c = Canvas(root, width=200, height=200, bg='white')**. Интуитивно понятны параметры класса **Canvas**: **width** – ширина “холста” в пикселях, **height** – высота

“холста” в пикселях и **bg** – цвет фона. После размещения данного виджета в окне приложения можно приступить к формированию изображения из графических примитивов.

Рассмотрим небольшой пример, где на “холст” выводятся несколько отрезков прямых с разными атрибутами визуализации. Для построения отрезков прямых в **Canvas** имеется метод **create_line(*args, **kw)**. Здесь **args** – это две или несколько координат точек линии. Можно задавать как **(x1, y1, x2, y2,…, xn, yn)** или **((x1, y1), (x2, y2),…, (xn, yn))**. Параметры **kw** – видовые характеристики линии (ий). Для более полного ознакомления с параметрами **create_line()** и других графических примитивов можно воспользоваться документацией на методы **Canvas()** – https://tkinter-docs.readthedocs.io/en/latest/widgets/canvas.html.

```
from tkinter import *

root = Tk()

c = Canvas(root, width=200, height=200, bg='white')
c.pack()

c.create_line(10, 10, 180, 50)
c.create_line((10, 15), (10, 50), (100, 50),(10, 15),
              width=2, fill='green')
c.create_line(100, 20, 180, 20, fill='red',
              width=5, dash=(10, 2),
              activefill='blue')
c.create_rectangle(10, 70, 80, 120, outline="#0f0")
c.create_oval(100,70, 150, 120, outline="#f00", fill="#f00")

root.mainloop()
```

В данном примере специально предложены разнообразные параметры графических примитивов, указывающие на гибкость их использования на практике. Например, если не указан цвет графического объекта, то по умолчанию возьмется черный цвет. Значение цвета можно записывать в виде названия на английском языке либо в шестнадцатеричном виде. Если использовать всего трехзначное шестнадцатеричное значение цвета, как в данном примере, то каждый элемент базового цвета (**R**, **G** и **B**) может быть выбран из 16 возможных значений. В случае шестизначного значения – **256**. Например, максимальное значение красного цвета можно задать как **“red”**, **“#f00”** или **“#ff0000”**. В последнем случае имеется возможность более плавно изменять цвет графического примитива.

В замкнутых фигурах имеется возможность указывать цвет контура **outline** и цвет внутренней области **fill**.

Если требуется изменять цвет графического объекта при наведении на него мыши, то можно воспользоваться параметром **activefill**. В приведенном примере в строке 13 параметр **activefill** равен **"blue"**, что позволит пунктирной линией красного цвета изменить цвет на синий при наведении на нее указателя мыши.

Для задания координат прямоугольника **canvas.create_rectangle(x1,y1, x2, y2)**: (**x1, y1**) – верхняя левая точка прямоугольника, а (**x2, y2**) – нижняя правая. Аналогично задаются координаты для прорисовки овала. Ведь в овал, согласно значениям координат в параметрах **canvas.create_oval()**, вписан в оболочку прямоугольника. Для моделирования окружности необходимо указать координаты квадрата, куда будет вписан овал.

Скриншот результата работы программы представлен на рис. 1.24.

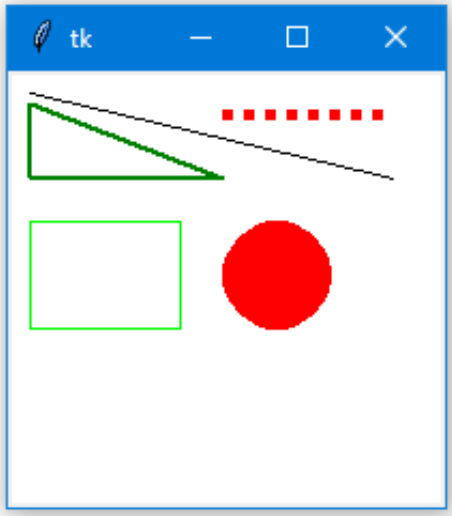


Рис. 1.24. Пример Canvas и графических примитивов

1.4. ПОЗИЦИОНИРОВАНИЕ ГРАФИЧЕСКИХ ИНТЕРФЕЙСОВ

Для позиционирования виджетов в окне приложения существуют три внутренних метода: **pack()**, **place()** и **grid()**. Такое многообразие методов размещения элементов необходимо для разной степени детализации в организации топологии графического интерфейса в окнах приложений. Рассмотрим каждый метод в отдельности.

1.4.1. Метод pack()

Это самый простой метод размещения виджетов. В таблице 1.10 представлены параметры и их значения для метода **pack()**.

Таблица 1.10

Параметр	Функционал
expand	Отвечает за возможность заполнения виджетом всего пространства контейнера (окна). По умолчанию данный параметр равен **FALSE**. В случае **TRUE** – виджет может заполнить все пространство контейнера
fill	Настраивает растяжение виджета для заполнения свободного пространства вокруг. Возможны значения: **NONE** (по умолчанию) – элемент не растягивается; **X** – элемент растягивается по оси X; **Y** – элемент растягивается по оси Y; **BOTH** – элемент растягивается по оси X и Y
side	Указывает, к какой стороне контейнера “прижаться” с центрированием. Возможны значения: **TOP** (по умолчанию) – выравнивается по верху; **BOTTOM** – выравнивается по низу; **LEFT** – выравнивается по левой стороне; **RIGHT** – выравнивается по правой стороне

Попробуем вывести в окно четыре кнопки **Botton** методом **pack()** без указания каких-либо параметров для размещения (рис. 1.25).

```
from tkinter import *

root = Tk()
root.title("Кнопки")
root.geometry("220x140")

btn1 = Button(root,text="Кнопка 1").pack()
btn2 = Button(root,text="Кнопка 2").pack()
btn3 = Button(root,text="Кнопка 3").pack()
btn4 = Button(root,text="Кнопка 4").pack()

root.mainloop()
```

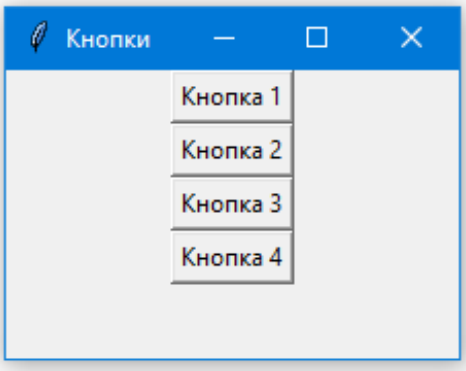


Рис. 1.25. Пример размещения четырех кнопок Button

Обратите внимание, что кнопки размещаются в той последовательности, как было предписано в коде программы. И каждая кнопка была отцентрирована и “прижата” к верху свободного пространства за предыдущей кнопкой.

А теперь давайте для каждой кнопки методом **pack()** укажем, к какой стороне выравниваться.

```
…
btn1 = Button(root,text="Кнопка 1").pack(side=TOP)
btn2 = Button(root,text="Кнопка 2").pack(side=LEFT)
btn3 = Button(root,text="Кнопка 3").pack(side=RIGHT)
btn4 = Button(root,text="Кнопка 4").pack(side=BOTTOM)
…
```

На рисунке 1.26 видим новое расположение четырех кнопок.

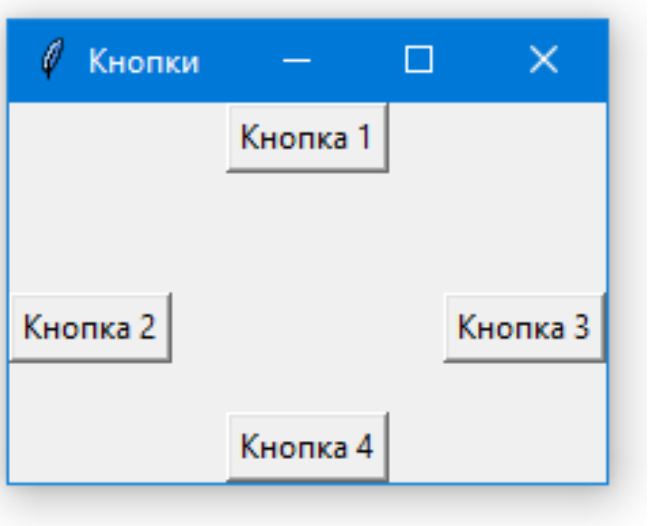


Рис. 1.26. Размещение четырех кнопок `Button` по четырем сторонам

Если посмотреть внимательно на скриншот, то можно заметить, что выравнивание кнопок происходит не по отношению к окну приложения, а свободного пространства окна за последней выведенной кнопкой. Это отчетливо видно, как разместились кнопки 2 и 3. Их выравнивание происходит относительно нижнего пространства окна ниже кнопки 1. В подтверждение этого рассмотрим еще один пример размещения методом **pack()**. Но попробуем все кнопки выровнить к левой стороне контейнера.

```
…
btn1 = Button(root,text="Кнопка 1").pack(side=LEFT)
btn2 = Button(root,text="Кнопка 2").pack(side=LEFT)
btn3 = Button(root,text="Кнопка 3").pack(side=LEFT)
btn4 = Button(root,text="Кнопка 4").pack(side=LEFT)
…
```

И как результат – рис. 1.27.

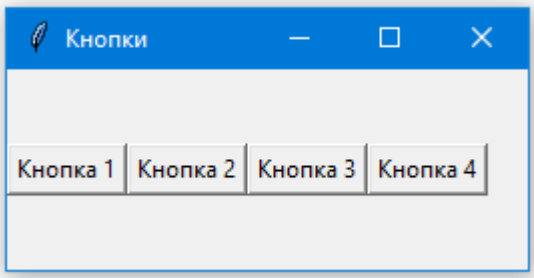


Рис. 1.27. Пример размещения четырех кнопок `Button` с выравниванием в левую сторону

1.4.2. Метод place()

Организация более сложных графических интерфейсов методом pack() – задача не тривиальная, а порой и не разрешимая. Для точного размещения виджетов в контейнере существует метод **`place()`**.

Параметры метода **`place()`** и их назначение приведены в табл. 1.11.

Таблица 1.11

Параметр	Функционал
`height`	Высота элемента в пикселях
`width`	Ширина элемента в пикселях
`relheight`	Высота элемента в значении числа **`float`** (0.0-1.0), которое указывает на долю от высоты родительского контейнера
`relwidht`	Ширина элемента в значении числа **`float`** (0.0-1.0), которое указывает на долю от ширины родительского контейнера
`x`	Координата верхнего левого угла элемента по горизонтали в пикселях относительно верхнего левого угла родительского контейнера
`y`	Координата верхнего левого угла элемента по вертикали в пикселях относительно верхнего левого угла родительского контейнера
`relx`	Координата верхнего левого угла элемента по горизонтали в значениях **`float`** (0.0-1.0) относительно верхнего левого угла родительского контейнера. Значение указывает на долю относительно ширины родительского контейнера
`rely`	Координата верхнего левого угла элемента по вертикали в значениях **`float`** (0.0-1.0) относительно верхнего левого угла родительского контейнера. Значение указывает на долю относительно высоты родительского контейнера
`bordermode`	Формат границы элемента. Возможные значения: **`INSIDE`** (по умолчанию); **`OUTSIDE`**
`anchor`	Устанавливает опции растяжения элемента. Может принимать значения n, e, s, w, ne, nw, se, sw, c, которые являются сокращениями от Noth (север – вверх), South (юг – низ), East (восток – правая сторона), West (запад – левая сторона) и Center (по центру). Например, значение nw указывает на верхний левый угол

Воспользуемся некоторыми параметрами метода **place()** и выведем, для примера (рис. 1.28), в окне 200x100 кнопку высотой 40 pix, шириной 120 pix, при горизонтальном смещении 20 pix и горизонтальном 30 pix.

```
from tkinter import *

root = Tk()
root.title("Кнопки")
root.geometry("200x100")

btn = Button(root,text="x = 20, y = 30",
             background="#555", foreground="#fff",
             padx="20", pady="8", font="18")
btn.place(height=40, width=120, x=20, y=30)

root.mainloop()
```

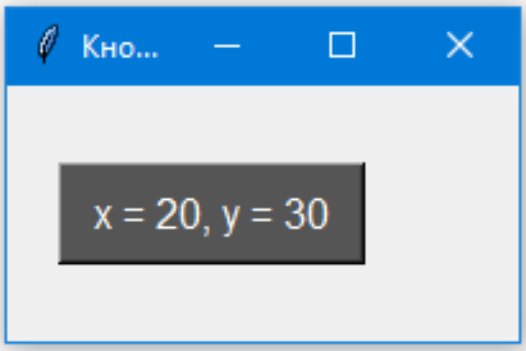


Рис. 1.28. Пример размещения кнопки Button методом place

В отличие от метода **pack()** методом **place()** можно размещать элементы графического интерфейса в любое место контейнера. Например (рис. 1.29), рассмотрим размещение двух простых кнопок **Button**, которое невозможно организовать методом **pack()**.

```
from tkinter import *

root = Tk()
root.title("Кнопки")
root.geometry("170x100")

btn1 = Button(root,text="Кнопка 1")
btn1.place(height=16, width=70, x=10, y=30)

btn2 = Button(root,text="Кнопка 2")
btn2.place(height=16, width=70, x=70, y=60)

root.mainloop()
```

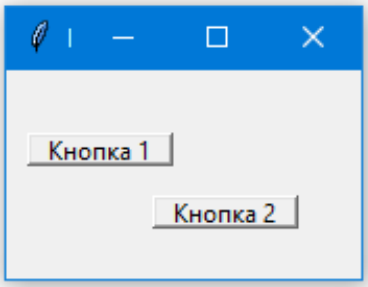


Рис. 1.29. Пример размещения двух кнопок `Button` методом `place`

Параметры с вещественными числами удобно применять в том случае, когда в процессе эксплуатации приложения будут масштабироваться диалоговые окна. В этом случае изменение размеров окна будет пропорционально отражаться на размерах графического интерфейса. Рассмотрим пример (рис. 1.30).

```
from tkinter import *

root = Tk()
root.title("Кнопки")
root.geometry("250x200")

btn1 = Button(root,text="Кнопка 1",
              background="#555", foreground="#fff",
              padx="20", pady="8", font="12")
btn1.place(relheight=0.2, relwidth=0.5,
           relx=0.25, rely=0.25)

btn2 = Button(root,text="Кнопка 2",
              background="#555", foreground="#fff",
              padx="20", pady="8", font="12")
btn2.place(relheight=0.2, relwidth=0.5,
           relx=0.25, rely=0.5)

root.mainloop()
```

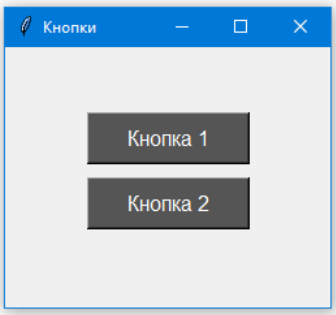


Рис. 1.30. Пример размещения двух кнопок `Button` методом place с использованием `float` параметров. Масштаб 1:1

1.4.3. Метод grid()

Данный метод размещения элементов графического интерфейса в родительском контейнере, в данном случае – окне приложения, основывается на “сетке”, которая накладывается на поле контейнера. Если быть точнее, то контейнер представляется в виде таблицы, у которой имеются строки и столбцы. И каждый необходимый виджет графического интерфейса вкладывается в соответствующую ячейку такой таблицы. Но следует заметить, что размеры таблицы не задаются заранее, а определяются в процессе ее заполнения. Для детального изучения метода **`grid()`** рассмотрим ее основные параметры (табл. 1.12).

Таблица 1.12

Параметр	Функционал
`column`	Номер столбца, с которого размещается графический элемент. Нумерация начинается с 0
`row`	Номер строки, с которой размещается графический элемент. Нумерация начинается с 0
`columnspan`	Количество столбцов, занимаемых графическим элементом. По умолчанию – 1
`rowspan`	Количество строк, занимаемых графическим элементом. По умолчанию – 1
`ipadx`	Расстояние в пикселях между вертикальной границей графического элемента и его внутренним содержанием. По умолчанию – 0
`ipady`	Расстояние в пикселях между горизонтальной границей графического элемента и его внутренним содержанием. По умолчанию – 0
`padx`	Расстояние в пикселях между вертикальной границей графического элемента и до границы соседнего элемента. Если заданы два значения в виде списка, то первое число – дистанция слева, а второе – справа. По умолчанию – 0
`pady`	Расстояние в пикселях между горизонтальной границей графического элемента и до границы соседнего элемента. Если заданы два значения в виде списка, то первое число – дистанция слева, а второе – справа. По умолчанию – 0
`sticky`	Параметр управляет выравниванием элемента в отведенном месте. Значения задаются в виде символов: “**`w`**” – левая сторона элемента прижимается к левой стороне свободного пространства; “**`n`**” – верхняя сторона элемента прижимается к верхней стороне свободного пространства; “**`e`**” – правая сторона элемента прижимается к правой стороне свободного пространства; “**`s`**” – нижняя сторона элемента прижимается к нижней стороне свободного пространства. Допускается комбинация этих параметров. Например: – “n,s” соответствует среднему положению по вертикали

Рассмотрим пример:

```
from tkinter import *

root = Tk()

Label(text="Имя:").grid(row=0, column=0)
Entry(width=30).grid(row=0, column=1, columnspan=3)

Label(text="Столбцов:").grid(row=1, column=0)
Spinbox(width=7, from_=1, to=50).grid(row=1, column=1)

Label(text="Строк:").grid(row=1, column=2)
Spinbox(width=7, from_=1, to=100).grid(row=1, column=3)

Button(text="Справка").grid(row=2,column=0)
Button(text="Вставить").grid(row=2,column=2)
Button(text="Отменить").grid(row=2,column=3)

root.mainloop()
```

В данном примере контейнер (окно) для размещения графических элементов можно представить в виде таблицы с индексацией ячеек:

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)

По этим индексам мы и расставляли виджеты в окне приложения. Например, элемент **`Entry`** (поле ввода) занимает три ячейки (0,1), (0,2) и (0,3) в нулевой строке. Для этого в программе было указано **`grid(row=0, column=1, columspan=3)`**. Аналогично были распределены остальные виджеты. Если посмотреть внимательно на готовое окно приложения, то можно заметить одно незанятое поле – (2,1). Это поле нигде в явном виде не указывается. Оно получается автоматически, как не задействованное для размещения. И в итоге мы получаем следующее диалоговое окно.

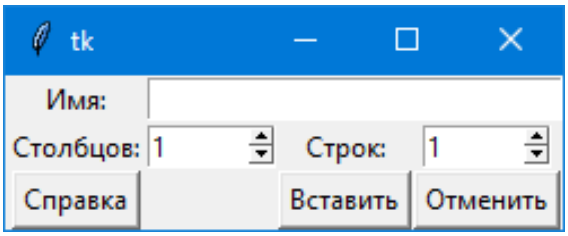


Рис. 1.31. Пример размещения девяти виджетов

1.5. ПРОГРАММИРОВАНИЕ СОБЫТИЙ В Tkinter

Организация интерактивного приложения с виджетами не обходится без обработки событий. События могут поступать из нескольких источников: клавиатура, мышь и оконные манипуляции. Событие в Python – это класс, и когда событие совершается, то создается объект события **event**. Отработка событий пишется пользователем, а факт выявления события реализован функцией привязки события **bind**.

Синтаксис функции **bind**: **object.bind(тип события, функция)**.

Рассмотрим некоторые используемые события клавиатуры и мыши (табл. 1.13).

Таблица 1.13

Событие	Описание
<Button-1>	Нажатие левой клавиши мыши
<Button-2>	Нажатие средней клавиши (колесико) мыши
<Button-3>	Нажатие правой клавиши мыши
<ButtonRelease-1>	Отпускание левой клавиши мыши
<ButtonRelease-2>	Отпускание средней клавиши мыши
<ButtonRelease-3>	Отпускание правой клавиши мыши
<B1-Motion>	Удержание левой клавиши мыши для перемещения
<Duble-Button-1>	Двойной щелчок левой клавиши мыши
<Enter>	Указатель мыши появился в области окна
<Leave>	Указатель мыши покинул области окна
<MouseWheel>	Крутится колесо мыши
<Motion>	Указатель мыши перемещается по окну программы компонента
<Return>	Нажата клавиша Enter клавиатуры
<space>	Нажата клавиша пробел
<KeyPresse-A> <A>	Нажата клавиша “A” (можно отследить нажатие любого символа)
<Alt-KeyPresse-A>	Одновременное нажатие Alt и A (можно отследить любую клавишу)
<Shift-KeyPresse-A>	Одновременное нажатие Shift и A (можно отследить любую клавишу)
<Ctrl-KeyPresse-A>	Одновременное нажатие Ctrl и A (можно отследить любую клавишу)
<Duble-KeyPresse-A>	Дважды быстро нажата клавиша “A” (можно отследить любую клавишу)
<Lock-KeyPresse-A>	Нажатие “A” в верхнем регистре (можно отследить любую клавишу)
<Key>	Нажата любая клавиша клавиатуры

Рассмотрим пример, где отрабатывается событие **<Button-1>**. Если указатель мыши находится в поле окна приложения и нажата левая клавиша мыши, то в терминальном окне выводим координаты указателя. Координаты выводятся в пикселях относительно верхнего левого угла окна приложения.

```
from tkinter import *

window = Tk()
window.title("Пример отработки событий")

# Щелкните левой кнопкой мыши, чтобы вывести координаты точки щелчка
def callback(event):
    print("Координаты нажатия:", event.x, event.y)

frame = Frame(window,width=300, height=300)
frame.bind("<Button-1>", callback)
frame.pack()

window.mainloop()
```

В данном примере объект события **event** передается в качестве параметра в функцию **callback**. Благодаря этому, координаты указателя мыши выбираются как **event.x** и **event.y**. Если необходимо считывать координаты указателя относительно всего экрана компьютера, то следовало бы записать **event.x_root** и **event.y_root**.

Аналогичным образом можно отследить и отработать событие, поступающее от клавиатуры. Сами события от клавиатуры представлены в табл. 1.13. А при отработке реакции на событие от клавиатуры можно использовать, например, при одиночном нажатии на клавишу, **event.char** – символ нажатой клавиши клавиатуры.

1.6. РАБОТА С ГРАФИЧЕСКИМИ ФАЙЛАМИ

Рассмотрим небольшой пример, где на Canvas будем выводить три графических объекта: синий отрезок прямой, изображение файла с расширением png и красный треугольник.

```
from tkinter import *

root = Tk()
root.title("Графика")

c=Canvas(root, width=200, height=200, bg="white")
c.pack()

c.create_line(10, 10, 180, 50, width=2, fill='blue')

img = PhotoImage(file='g:/tree.png')
```

```
image = c.create_image(0, 0, anchor='nw', image=img)

c.create_line((10, 15), (10, 50), (100, 50), (10,15),
              width=2, fill="red")

root.mainloop()
```

Обратите внимание, что на итоговом изображении **Canvas** отсутствует синий отрезок прямой, хотя этот отрезок был представлен в программе как **c.creat_line(10, 10, 180,50, width=2, fill="blue")**. Дело в том, что этот отрезок был сформирован на "холсте", но последующий вывод графического файла заслонил данный отрезок. А ломаная линия красного цвета выводилась после **png**-файла и поэтому мы можем наблюдать ее на изображении (рис. 1.32).

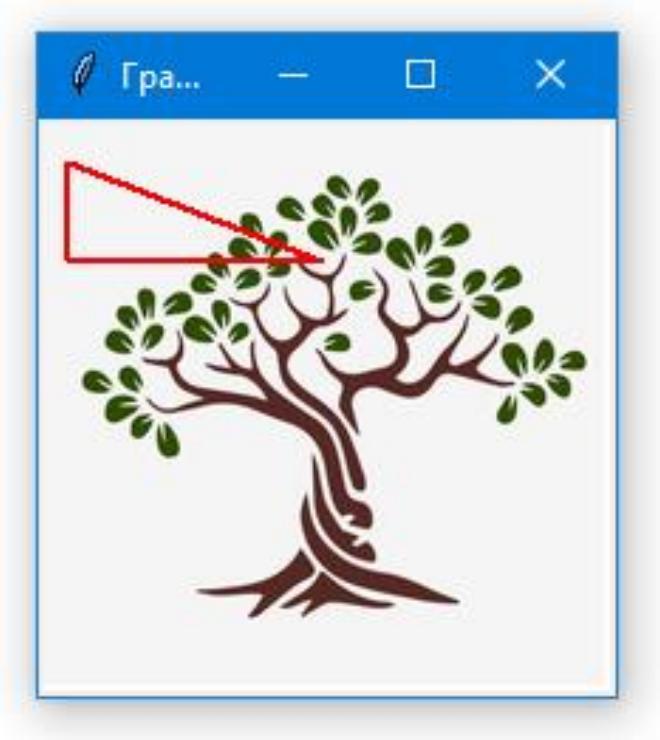


Рис. 1.32. Пример вывода png-файла на Canvas

Немного усложним задачу, выведем в окно приложения текст "**Вывод изображения из файла**", кнопку с внутренним текстом "**Показать**" и само изображение, взятое из файла с расширением **png**. Изображение из файла будет показываться после нажатия на кнопку. Пример подобной программы приведен ниже. Здесь мы использовали позиционирование виджетов **place()** для более точного размещения графических интерфейсов задачи.

```
from tkinter import *

def click():
    image = c.create_image(1, 1, anchor="nw", image=img)
```

```
root = Tk()
root.title("Графика")
root.geometry("350x280")

img = PhotoImage(file='g:/tree.png')

l=Label(root, text="Вывод изображения из файла", font=8)
l.place(height=20,width=240,x=5,y=10)

b=Button(root, text="Показать", command=click, font=8)
b.place(height=20,width=100,x=241,y=10)

c=Canvas(root, width=200, height=200, bg="white")
c.place(height=200,width=200,x=20,y=45)

root.mainloop()
```

Функция **click()** вставляет в объект **c** (“холст”) содержимое объекта изображения **img** и вызывается при нажатии кнопки **b**. Для этого при создании объекта кнопкой **b** была сформирована запись **command=click** в строке параметров. На рисунке 1.33 мы видим результат работы программы после ее запуска и после нажатия на кнопку “**Показать**”.

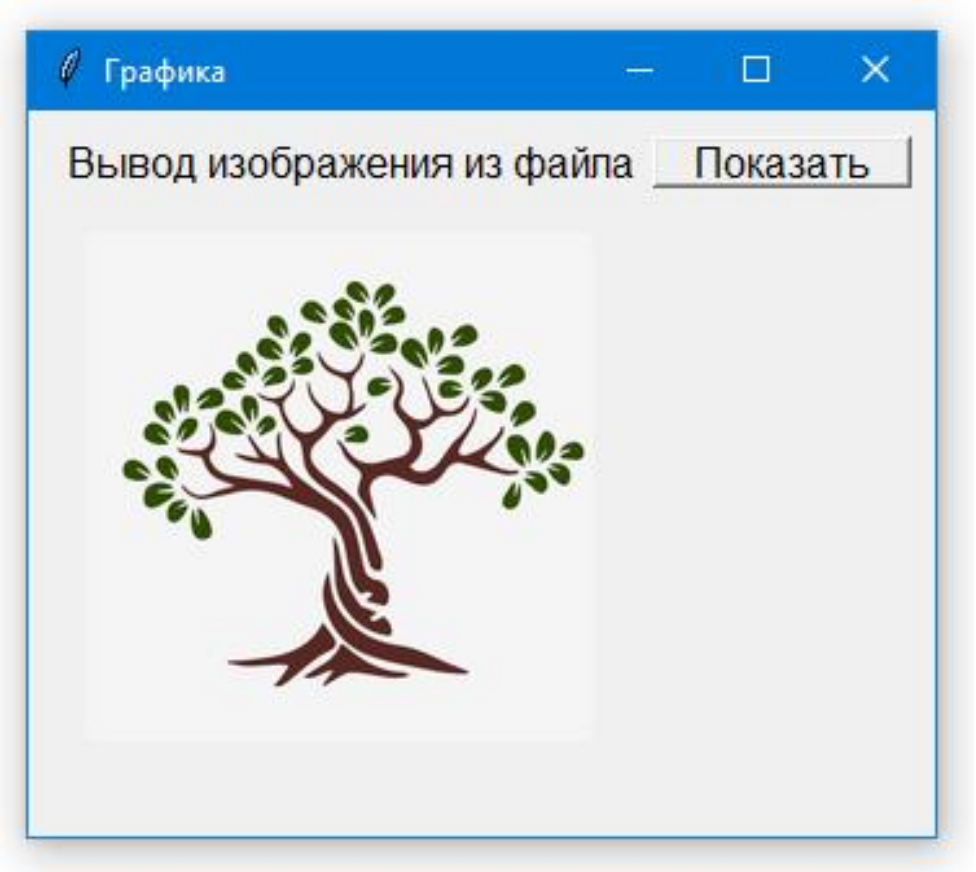


Рис. 1.33. Пример вывода png-файла на Canvas после нажатия на кнопку “Показать”

В языке Python существуют много способов вывести графический файл в окно приложения. В вышеприведенных примерах мы коснулись лишь некоторой части библиотеки **tkinter** для работы с графическими файлами. Более специализированной библиотекой для работы с графическими файлами является, например, **PIL** (Python Image Library). В последнее время данная библиотека была улучшена и дополнена новыми возможностями обработки разнообразных графических форматов. Развитие **PIL** получило название пакета – **Pillow**.

Для использования **Pillow** в программах сперва необходимо установить **Pillow**. Если Вы работаете в среде **PyCharm**, то достаточно в терминальном окне отработать следующую команду: **“pip install Pillow”** или через встроенные средства: **“File > Settings > Project: *project_name* > Project Interpreter”**. В последнем варианте нужно найти в списке **Pillow** и нажать на кнопку “**Install Package**”.

Рассмотрим предыдущую задачу с новым пакетом **PIL**, из которого возьмем два модуля: **Image** и **ImageTk**.

```
from tkinter import *
from PIL import Image, ImageTk

def click():
    image = c.create_image(1, 1, anchor="nw", image=photo)

root = Tk()
root.title("Графика")
root.geometry("350x280")

c=Canvas(root, width=200, height=200, bg="white")
c.place(height=200,width=200,x=20,y=45)

image = Image.open('g:/tree.png')
photo = ImageTk.PhotoImage(image)

l=Label(root, text="Вывод изображения из файла", font=8)
l.place(height=20,width=240,x=5,y=5)

b=Button(root, text="Показать", command=click, font=8)
b.place(height=20,width=100,x=241,y=10)

root.mainloop()
```

Результат работы данной программы аналогичен, как на рис. 1.33.

Основные функции **Pillow** находятся в модуле **Image**. **Pillow** считывает больше 40 форматов изображений в режиме чтения. Полная поддержка есть у 21 формата, среди которых — BMP, JPEG, GIF, TIFF, PNG и PSD. Если файл открывается функцией **open**, то библиотека определяет его формат автоматически, что очень удобно.

Более детально познакомиться с возможностями **Pillow** можно, например, здесь: https://pillow.readthedocs.io/en/stable/.

Рассмотрим некоторые функции модуля **Image**.

Для того чтобы обрезать готовое изображение, взятое из графического файла, достаточно воспользоваться функцией **crop()**. Чтобы воспользоваться данной функцией, достаточно открыть интересующий нас графический файл и вызвать метод **crop()**. Приведем пример практического использования функции **crop()**:

```
image = Image.open('g:/tree.png')
cropped = image.crop((50, 50, 150, 150))
cropped.save('g:/tree_new.png')
```

Здесь открывается файл **tree.png** методом **open** модуля **Image**, т.е. создается объект **image** файла **tree.png**. И теперь по данному объекту отрабатывается метод **crop((50,50,150,150))**, который создает новый объект **cropped** с обрезанным изображением. В данном случае вырезается прямоугольная область с координатой (x1=50, y1=50) верхней левой вершины области и координатой (x2=150, y2=150) нижней правой вершины области. Координаты записываются в пикселях. Обратите внимание, что в качестве параметров необходимо записывать кортеж из значений, поэтому мы видим двойные скобки в параметрах **crop()**. И в завершение объект обрезанного изображения сохраняется методом **save()** с новым именем. Метод **save()** интересен тем, что может сохранять изображение в любом доступном графическом формате пакета **PIL**. Если указать **cropped.save("g:\tree_new.jpg")**, то и создастся новый графический файл с расширением **jpg**. Это очень удобный способ оперативной конвертации графического файла в другой формат.

Если создать специальный скрипт для конвертирования графического формата в другой, то можно обойтись всего лишь двумя действиями – открыть файл и сохранить файл с новым расширением. Например, так:

```
image = Image.open('g:/tree.png')
image.save('g:/tree_new.png')
```

Кроме изменения графического формата у сохраняемого изображения, в **PIL** имеется метод **convert()** для преобразования самого изображения. В таблице 1.14 представлены возможные параметры для данного метода.

Таблица 1.14

Параметр	Описание
1	1-битные пиксели, черно-белое изображение
L	8-битные пиксели, черно-белое изображение
P	8-битные пиксели, используется индексный режим цвета (палитра)
RGB	3*8-битные пиксели (True Color)
RGBA	4*8-битные пиксели (True Color + альфа-канал (прозрачность))
CMYK	4*8-битные пиксели (Cyan, Magenta, Yellow, Key или Black)
YCbCr	3*8-битные пиксели (Y-яркость, Cb-цветность синего, Cr-цветность красного)
I	32-разрядные целые пиксели со знаком
F	32-битные пиксели с плавающей точкой

Итак, для трансформации цветного изображения в черно-белое достаточно выполнить следующие операции:

```
image = Image.open('g:/tree.png')
grayscale = image.convert("L")
grayscale.save('g:/tree_new.png')
```

Если теперь мы посмотрим файл **tree_new.jpg** любым просмотрщиком графических файлов операционной системы компьютера, то увидим черно-белое изображение. А если нет желания выходить из среды программирования **Python** и необходимо быстро посмотреть содержимое нашего нового файла, то можно воспользоваться методом **show()**. Эта функция подгружает программу операционной среды для просмотра данного изображения – **Paint**. Код фрагмента программы представлен ниже.

```
image = Image.open('g:/tree.png')
grayscale = image.convert("L")
grayscale.save('g:/tree_new.png')
grayscale.show()
```

Для поворота изображения используется метод **rotate()**, где в скобках указывается значение поворота в градусах. Повороты изображения осуществляются относительно центра картинки и против часовой стрелки, если значение угла поворота задано положительным значением. Если угол поворота задан отрицательным числом, то поворот осуществится по часовой стрелке. Рассмотрим пример, где наше тестовое изображение поворачивается на 10 градусов.

```
image = Image.open('g:/tree.png')
grayscale = image.rotate(10)
grayscale.save('g:/tree_new.png')
```

И на рисунке 1.34 мы видим результат от применения **rotate(10)**.

Рис. 1.34. Поворот изображения на 10 градусов

Обратите внимание, что непоместившееся изображение после поворота отсекается, что не всегда удобно. Для сохранения всего изображения после поворота достаточно добавить параметр **`expand=True`** в метод **`rotate()`**. Немного скорректируем предыдущую программу:

```
image = Image.open('g:/tree.png')
grayscale = image.rotate(10, expand=True)
grayscale.save('g:/tree_new.png')
```

Теперь рисунок (рис. 1.35) полностью показывается после поворота, и такое изображение сохраняется в новом файле.

Рис. 1.35. Поворот изображения на 10 градусов при `expand=True`

Центр поворота изображения можно задавать любой через параметр **`center(x,y)`**. Точка, относительно которой происходит поворот, не обязательно должна находиться в рамках нашего исходного изображения. Значения x и y могут быть и положительными, и отри-

цательными. Ниже приводим пример поворота изображения на 10 градусов против часовой стрелки относительно точки (0,0) (рис. 1.36).

```
image = Image.open('g:/tree.png')
grayscale = image.rotate(10, center=(0, 0))
grayscale.save('g:/tree_new.png')
```

Рис. 1.36. Поворот изображения на 10 градусов относительно точки (0,0)

Методом **rotate()** можно и перемещать изображение. Достаточно воспользоваться параметром **translate=(x,y)**, где **x** – смещение в пикселях по горизонтали, **y** – по вертикали. Смещение осуществляется относительно верхней левой точки исходного изображения. Например, нам необходимо сместить изображение в рамках размера исходного изображения на 50 пикселей по горизонтали и 20 по вертикали. При этом значение градуса поворота прировняем нулю (рис. 1.37).

```
image = Image.open('g:/tree.png')
grayscale = image.rotate(0, translate=(50, 20))
grayscale.save('g:/tree_new.png')
```

Рис. 1.37. Перемещение изображения

2. ЯЗЫК JAVA

2.1. ГРАФИЧЕСКИЙ ИНТЕРФЕЙС ПОЛЬЗОВАТЕЛЯ В JAVA

Разработка приложений с графическим пользовательским интерфейсом (Graphical User Interface, GUI) в Java построена на использовании библиотеки классов базовой подсистемы GUI Abstract Window Toolkit (`java.awt`) и основанной на ней альтернативной библиотеки графических элементов Swing (`javax.swing`).

Внешний вид (а в некоторых случаях и поведение) визуальных компонентов (виджетов) графической подсистемы AWT зависит от операционной системы, под управлением которой запускается приложение. Базовый набор элементов управления, окон и диалогов AWT использует на каждой платформе нативный набор ресурсов, за что компоненты AWT называют «тяжеловесными» компонентами. Это приводит к тому, что одно и то же приложение будет выглядеть по-разному на разных платформах. При этом отсутствует возможность управлять дополнительными характеристиками отображения элементов (такими, например, как форма и прозрачность), так как внешний вид каждого конкретного компонента определяется операционной системой. Такое поведение AWT не поддерживает декларируемый Java принцип кросс-платформенности «Пишите один раз, запускайте где угодно» («Write once, run everywhere», WORE).

Для того чтобы исключить кросс-платформенные ограничения AWT, была разработана библиотека классов Swing, компоненты которой являются «легковесными» («облегченными»), так как написаны полностью на Java. Внешний вид каждого компонента определяется классами Swing, а не операционной системой. Таким образом, каждый компонент выглядит и функционирует одинаково на всех платформах. Современный графический пользовательский интерфейс Java построен на основе Swing.

Библиотека классов Swing поддерживает также принцип подключаемого внешнего вида. Код, отвечающий за то, как выглядит графический элемент, отделен от кода, отвечающего за поведение данного элемента. Преимуществом данного подхода является легкость изменения способа визуализации каждого компонента без затрагивания логики его работы. Такой подход также позволяет заранее определить целые наборы различных способов визуализации, которые задают разные стили GUI (в том числе и динамически во время работы приложения).

2.2. СХЕМА «МОДЕЛЬ–ПРЕДСТАВЛЕНИЕ–КОНТРОЛЛЕР»

В общем случае визуальный компонент определяется тремя отдельными аспектами: как компонент выглядит во время его визуализации на экране, как компонент взаимодействует с пользователем и информацией о состоянии компонента (текущем значении его свойств). Независимо от того, какая архитектура используется для реализации компонента, она должна неявно включать эти три аспекта. Главной архитектурой в разработке компонентов с графическим интерфейсом является архитектура «Модель–Представление–Контроллер» («Model–View–Controller», MVC).

Удобство использования (а вследствие чего и успешность) архитектуры MVC объясняется тем, что каждый ее элемент соответствует некоторому аспекту компонента. **Модель** соответствует информации о состоянии, связанной с компонентом. Например, в случае флажка (компонент JCheckbox в Swing) его модель содержит поле, которое показывает, отмечен ли флажок. **Представление** определяет, как компонент отображается на экране (или будет отображаться) в зависимости от текущего состояния модели. **Контроллер** определяет, как компонент будет реагировать на действия пользователя. Например, когда пользователь щелкает на флажке, контроллер выполняет действия по изменению модели, чтобы отразить выбор пользователя (отметка флажка или снятие отметки). Затем это также приводит и к обновлению представления. Разделяя компонент на модель, представление и контроллер, можно изменять конкретную реализацию любого из этих аспектов, не затрагивая остальные. Например, набор различных представлений может визуализировать один и тот же компонент разными способами, однако это не будет влиять на модель или контроллер.

Swing использует модифицированную версию MVC, которая объединяет представление и контроллер в один логический объект, называемый делегатом пользовательского интерфейса (UI delegate). В связи с этим подход, применяемый в Swing, называется или архитектурой «Модель–Делегат» («Model–Delegate»), или архитектурой «Разделяемая модель» («Separable Model»). Реализация принципа подключаемого внешнего вида в Swing возможна именно благодаря архитектуре «Модель–Делегат». Поскольку представление и контроллер отделены от модели, внешний вид можно изменять, не влияя на способ использования компонента внутри программы. Для поддержания архитектуры «Модель–Делегат» большинство компонентов Swing содержат два объекта. Один из них представляет модель, а другой – делегата пользовательского интерфейса.

2.3. КОНТЕЙНЕРЫ

В основе GUI с использованием классов Swing находятся две группы элементов: компоненты и контейнеры. **Компонент** представляет собой отдельный визуальный элемент управления, например текстовое поле или кнопка. **Контейнер** является особым типом компонента и предназначен для размещения, объединения и управления другими компонентами. Более того, для отображения компонента необходимым условием является его нахождение внутри конкретного контейнера.

В любом приложении Java, построенном на классах Swing, есть как минимум один контейнер. Так как контейнеры также являются и компонентами, то любой контейнер может содержать другие контейнеры. И для построения сложных приложений определяется иерархия вместимости, в которой выделяются контейнеры верхнего уровня. Таким образом, в Swing определены два типа контейнеров: контейнеры верхнего уровня и «легковесные» контейнеры.

К контейнерам верхнего уровня относятся `JFrame`, `JApplet`, `JWindow` и `JDialog`. Эти классы контейнеров не являются производными от класса `JComponent`. Базовыми для них классами являются AWT-классы `Component` и `Container`, поэтому контейнеры верхнего уровня являются «тяжеловесными» компонентами (в библиотеке Swing это единственные компоненты подобного рода). Контейнеры верхнего уровня также называют окнами, которые являются основой графического пользовательского интерфейса любой операционной системы. Окна визуально разделяют выполняемые в операционной среде приложения. Внешний вид окон будет зависеть от конкретной операционной системы. Контейнер верхнего уровня не может находиться в другом контейнере. Каждая иерархия вместимости для любого приложения с GUI должна начинаться с контейнера верхнего уровня.

Родителем всех окон Swing является `JWindow` – окно без рамки и без элементов управления. Класс `JWindow` определяет базовые возможности оконных приложений, например закрытие или перемещение. Окно `JFrame` наследует свойства класса `JWindow`, и является самым часто используемым контейнером верхнего уровня. У `JFrame`, в отличие от `JWindow`, есть рамка, которая позволяет изменять размер окна, заголовок с названием приложения (может быть пустым), кнопки управления для закрытия и свертывания окна. В окне `JFrame` также есть возможность использовать системное меню, позволяющее выполнять стандартные действия над окном и приложением.

Вторым типом контейнеров в Swing являются «легковесные» контейнеры, производные от класса `JComponent`. Примером «легковесного» контейнера является `JPanel`. Данные контейнеры часто используют для объединения компонентов в группы по структурному или функциональному признаку, упрощая управление таких компонентов.

Каждый контейнер верхнего уровня включает в себя набор панелей. Панель `JRootPane` представляет собой корневую панель, находящуюся наверху иерархии вмести-

мости панелей и предназначенную для управления остальными панелями. `JRootPane` содержит «стеклянную» панель (glass pane), панель содержимого (content pane) и многослойную панель (layered pane).

Многослойная панель является экземпляром `JLayeredPane` и предназначена для управления глубиной расположения компонентов. Значение, соответствующее этой глубине, определяет слой, в котором будет отображен компонент (то есть в каком порядке будут отрисованы графические элементы). Многослойная панель вмещает в себя панель содержимого и линейку меню приложения.

Панель содержимого предназначена для размещения визуальных компонентов приложения и по умолчанию является непрозрачным экземпляром `JPanel`. Любой компонент, который добавляется в контейнер верхнего уровня, попадает в панель содержимого.

«Стеклянная» панель находится над всеми остальными панелями и по умолчанию является прозрачным экземпляром `JPanel`. Она может перехватывать и управлять событиями мыши или рисовать поверх любого другого компонента.

Например, оконное приложение Java на основе класса `JFrame`:

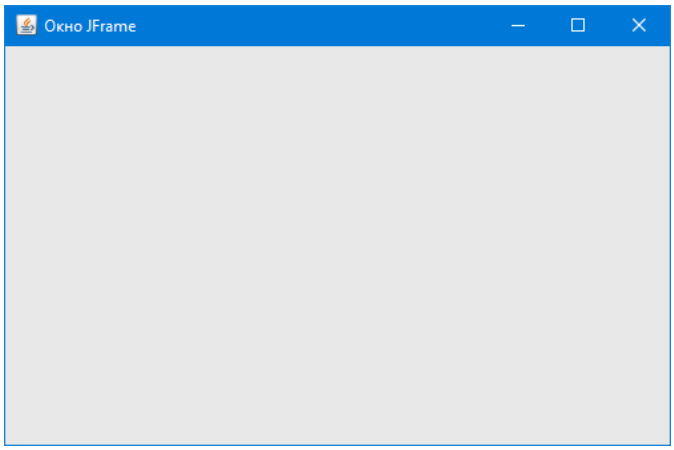


Рис. 2.1. Окно JFrame

Код программы, внешний вид которой представлен на рис. 2.1:

```
import javax.swing.*;
import java.awt.*;

public class SimpleGUIApp extends JFrame {

  SimpleGUIApp() {
    initGUI();
  }

  private void initGUI() {
    setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    getContentPane().setPreferredSize(new Dimension(500, 300));
```

```
        getContentPane().setBackground(new Color(100, 100, 100, 10));
        setTitle("Окно JFrame");
        pack();
    }

    public static void main(String args[]) {
        EventQueue.invokeLater(() -> {
            new SimpleGUIApp().setVisible(true);
        });
    }

}
```

Очевидно, что для создания собственного окна достаточно унаследовать имеющийся в библиотеке Swing контейнер верхнего уровня `JFrame`. В данном коде наследником является класс `SimpleGUIApp`. В его конструкторе вызывается закрытый метод `initGUI()`, в котором и формируется внешний вид приложения.

Метод `setDefaultCloseOperation()` устанавливает операцию, которая будет выполняться по умолчанию, когда пользователь инициирует «закрытие» окна. В данном случае (как и в большинстве случаев для контейнеров верхнего уровня) в метод передается значение константы `EXIT_ON_CLOSE`, что предполагает использовать для «закрытия» окна метод `System.exit()`. Метод `getContentPane()` предоставляет доступ к панели содержимого окна. В данном примере у полученного объекта вызываются методы `setPreferredSize()` и `setBackground()` для управления размерами отображаемой области окна и фоновым цветом соответственно. Метод `setTitle()` задает строку заголовка окна. Метод `pack()` изменяет размер окна в соответствии с предпочтительными размерами и расположением его подкомпонентов.

В методе `main()` создается объект класса `SimpleGUIApp`, для которого вызывается метод `setVisible()`, отображающий окно приложения. Окно в данном случае сразу получает фокус, то есть становится активным окном в операционной среде. Стоит отметить, что объекты графического пользовательского интерфейса Swing должны создаваться и управляться только в потоке диспетчеризации событий. Для этого создание и отрисовка окна «упаковываются» в отдельный поток исполнения и помещаются в конец очереди событий потока диспетчеризации. В противном случае, если этого не сделать и приложение будет занято вычислениями, то возможна задержка реакции на действия пользователя с элементами GUI.

Далее все приведенные примеры создания и размещения основных визуальных компонентов для сокращения и компактности описания не будут содержать полного кода. Для фрагментов кода из этих примеров предполагается, что они будут находиться в методе `initGUI()`.

2.4. КОМПОНОВЩИКИ

Каким образом в панели содержимого будут располагаться компоненты, определяет **менеджер расположения** или **компоновщик** (layout manager). Независимо от операционной системы, версии виртуальной машины Java, разрешения и размеров экрана менеджер расположения гарантирует, что компоненты будут иметь предпочтительный или близкий к нему размер и располагаться в том порядке, который был указан разработчиком при создании программы.

Поддержка компоновщиков определена в базовом классе контейнеров `Container`. Для любого компонента Swing можно определить, какой менеджер размещения используется им в данный момент или установить нужный менеджер. Для этого используются методы `getLayout()` и `setLayout()`. Основными менеджерами расположения являются `BorderLayout`, `FlowLayout`, `GridLayout`, `BoxLayout` и `GroupLayout`.

`BorderLayout` – менеджер для полярного расположения. Он предназначен для обычных и диалоговых окон. Данный компоновщик позволяет просто и быстро разместить наиболее часто используемые элементы любого окна: панель инструментов, строку состояния и основное содержимое. Для этого `BorderLayout` разбивает окно на четыре области, а все оставшееся место заполняется компонентом, выполняющим основную функцию приложения. Чтобы добавить с помощью менеджера расположения `BorderLayout` компонент, в его методе `add()` необходимо использовать дополнительный параметр, который определяет область контейнера для размещения компонента. Значениями этого параметра могут быть:

- `BorderLayout.NORTH` – в этом случае компонент располагается вдоль верхней границы окна и растягивается на всю его ширину. Обычно в этом месте размещается панель инструментов;
- `BorderLayout.SOUTH` – компонент располагается вдоль нижней границы и растягивается на всю ширину окна. Такое положение характерно для строки состояния;
- `BorderLayout.WEST` – компонент располагается вдоль левой границы окна и растягивается на всю его высоту. При этом учитываются размеры северных и южных компонентов, имеющих приоритет;
- `BorderLayout.EAST` – компонент располагается вдоль правой границы окна;
- `BorderLayout.CENTER` – компонент помещается в центр окна, занимая максимально возможное пространство.

`FlowLayout` – менеджер для последовательного расположения. Он размещает компоненты в контейнере слева направо, сверху вниз. При полном заполнении компонентами строки контейнера `FlowLayout` переходит на следующую строку вниз. Данное расположение устанавливается по умолчанию в панелях `JPanel` (в том числе и для панели содержимого окна). Основным свойством `FlowLayout` является определение предпочтительного размера компонентов.

`GridLayout` – менеджер для табличного расположения. Он представляет контейнер в виде таблицы с ячейками одинакового размера. Количество строк и столбцов можно указать в конструкторе. Имеется возможность задать произвольное количество либо строк, либо столбцов (но не одновременно). Все ячейки таблицы имеют одинаковый размер, равный размеру самого большого компонента, находящегося в таблице.

Менеджер расположения `BoxLayout` позволяет управлять размещением компонентов в вертикальном или в горизонтальном направлении, пространством между компонентами, используя вставки. Для размещения компонентов в вертикальной плоскости конструктору передается константа `BoxLayout.Y_AXIS`, а для размещения в горизонтальной – `BoxLayout.X_AXIS`.

Менеджер расположения компонентов `GroupLayout` раскладывает компоненты по группам. Группы имеют горизонтальное и вертикальное направления и могут быть параллельными и последовательными. В последовательной группе у каждого следующего компонента координата вдоль оси на единицу больше (имеется в виду координата в сетке), в параллельной группе компоненты имеют одну и ту же координату.

2.5. КОМПОНЕНТЫ

В общем случае компоненты из библиотеки Swing происходят от класса `JComponent`, который предлагает функциональные возможности, общие для всех компонентов. Например, `JComponent` поддерживает подключаемый внешний вид. JComponent является наследником классов `Container` и `Component`, следовательно, совместим с компонентами AWT. Все компоненты Swing представлены классами, определенными в пакете `javax.swing` и начинаются с буквы «J».

В современных средах разработки, используемых для Java, поддерживаются редакторы форм. Редактор форм может использоваться для визуального формирования внешнего вида приложения. В этом случае код на Java, соответствующий манипуляциям разработчика (добавление компонента, изменение его расположения, размеров, используемых цветов и т.п.) в редакторе форм, создается автоматически. Компоненты в редакторе форм обычно доступны разработчику в отдельной панели. Пример такой панели (палитры компонентов среды разработки NetBeans IDE) представлен на рис. 2.2.

Основными классами элементов управления Swing являются `JLabel`, `JButton`, `JToggleButton`, `JCheckBox`, `JRadioButton`, `JComboBox`, `JList`, `JTextField`, `JPasswordField`, `JTextArea`, `JScrollBar`, `JSlider`, `JProgressBar`, `JSpinner`, `JSeparator`.

`JLabel` используется для вывода статического текста (так называемой «метки»). Возможен вывод текста с изображением (иконкой) и применение html-разметки для форматированного вывода.

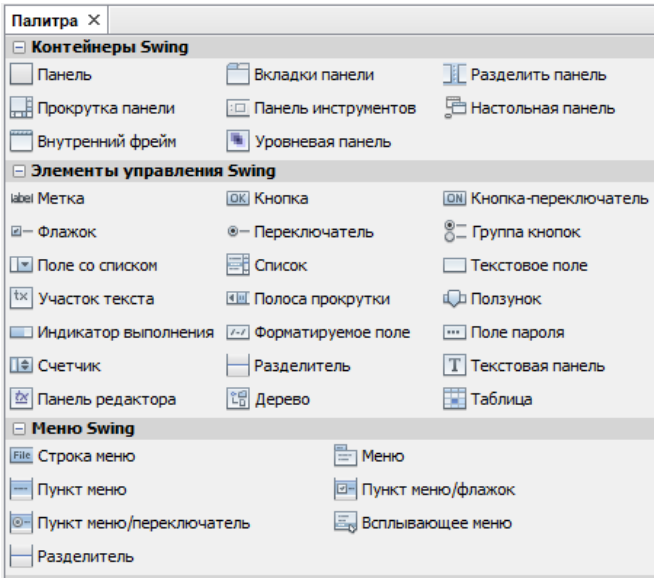


Рис. 2.2. Палитра компонентов NetBeans IDE

`JButton` используется для взаимодействия с пользователем, представляет собой кнопку (прямоугольник с текстом и(или) изображением). Обычно для кнопки регистрируется слушатель события действия (для обработки события нажатия кнопки).

`JToggleButton` представляет собой кнопку, которая может находиться и в обычном состоянии, и в нажатом. Состояние кнопки при этом фиксируется. Когда пользователь взаимодействует с такой кнопкой, она меняет свое состояние. Управление внешним видом компонента осуществляется так же, как и у `JButton`.

Пример отображения компонентов `JLabel`, `JButton` и `JToggleButton` представлен на рис. 2.3.

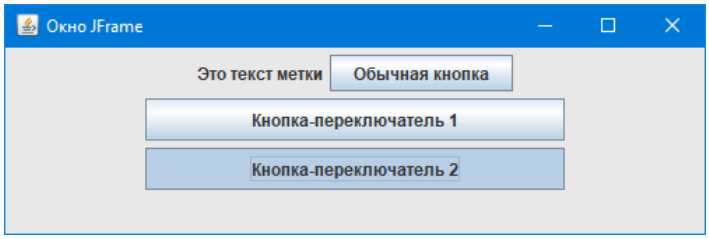


Рис. 2.3. Компоненты JLabel, JButton и JToggleButton

Фрагмент кода для приложения на рис. 2.3:

```
setLayout(new FlowLayout());
JLabel jlb = new JLabel();
jlb.setText("Это текст метки");
add(jlb);

add(new JButton("Обычная кнопка"));

JToggleButton jtb = new JToggleButton("Кнопка-переключатель 1");
```

```
jtb.setPreferredSize(new Dimension(300, 30));
add(jtb);
jtb = new JToggleButton("Кнопка-переключатель 2");
jtb.setPreferredSize(new Dimension(300, 30));
add(jtb);
```

Для окна установлен диспетчер компоновки `FlowLayout`. Каждый новый созданный компонент размещается в окне методом `add()`. Так же стоит отметить, что конструктор практически каждого виджета может принимать в качестве параметра строку, текст которой будет размещен на компоненте.

`JCheckBox` представляет собой элемент для выбора («флажок»). Класс `JCheckBox` является производным от `JToggleButton`, причем отличается от последнего только внешним видом: квадрат, в который можно поставить или убрать «галочку». `JCheckBox` обычно используется в меню настроек для множественного выбора из группы элементов.

`JRadioButton` представляет собой элемент для выбора (переключатель). Класс `JRadioButton` также является наследником класса `JToggleButton`. Внешний вид элемента: круг, в который можно поставить или убрать точку. Однако в группе `JRadioButton` ведет себя как переключатель. Если пользователь выбирает конкретный элемент («устанавливает точку»), то с ранее выбранного элемента выбор снимается («убирается точка»). `JRadioButton` также используется в меню настроек, но для выбора одного элемента из группы элементов.

Пример отображения компонентов `JCheckBox` и `JRadioButton` представлен на рис. 2.4.

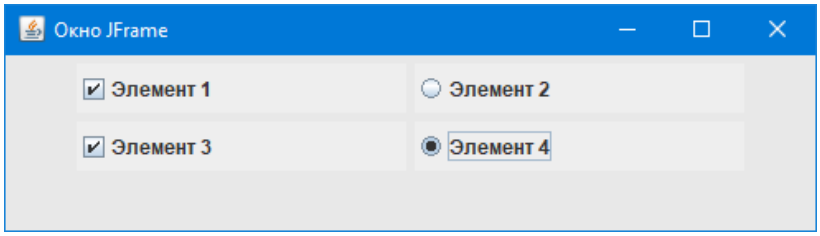


Рис. 2.4. Компоненты JCheckBox, JRadioButton

Фрагмент кода для приложения на рис. 2.4:

```
ButtonGroup bg = new ButtonGroup();
Dimension d = new Dimension(200, 30);

JCheckBox jchb = new JCheckBox("Элемент 1");
jchb.setPreferredSize(d);
JRadioButton jrb = new JRadioButton("Элемент 2");
jrb.setPreferredSize(d);
add(jchb);
add(jrb);
bg.add(jrb);
```

```
jchb = new JCheckBox("Элемент 3");
jchb.setPreferredSize(d);
jrb = new JRadioButton("Элемент 4");
jrb.setPreferredSize(d);
add(jchb);
add(jrb);
bg.add(jrb);
```

Для объединения элементов `JRadioButton` в группу используется объект типа `ButtonGroup`. В таком случае при нажатии (выборе) на неактивный (невыбранный) элемент из группы он будет выделен (выбран). При этом с активного на тот момент элемента будет снято выделение. Элементы `JCheckBox` не могут быть объединены в группу и являются независимыми элементами для выбора.

`JComboBox` используется для выбора одного элемента из нескольких вариантов. Этот виджет представляет собой раскрывающийся список, который в нормальном состоянии отображает только один выбранный элемент. Остальные элементы появляются в специальном всплывающем окне при раскрытии списка. `JComboBox` допускает редактирование текущего элемента, то есть пользователь может не только выбрать определенный элемент, но и ввести свое собственное значение. Список `JComboBox` обладает возможностью поиска элементов с клавиатуры, что значительно упрощает работу с большими наборами данных.

`JList` используется для отображения группы элементов (списка). В отличие от компонента `JComboBox` данный виджет отображает одновременно сразу несколько элементов списка (весь список или его часть). Кроме того, пользователь может выбрать в списке также несколько элементов одновременно (если установлен соответствующий режим выделения).

Пример отображения компонентов `JComboBox` и `JList` представлен на рис. 2.5.

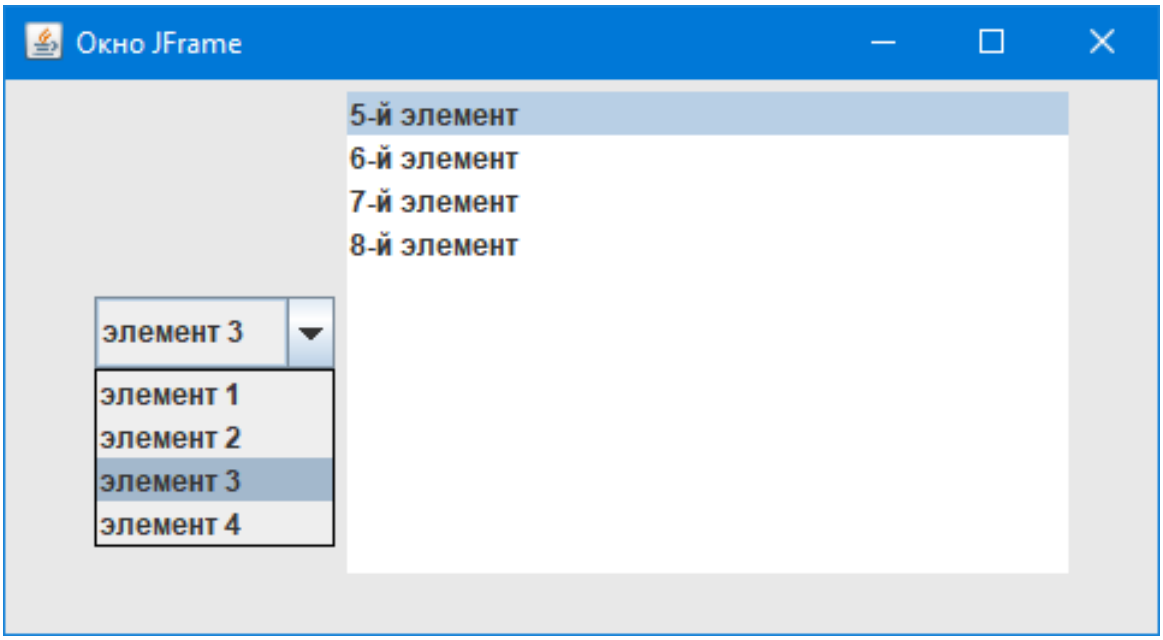


Рис. 2.5. Компоненты JComboBox, JList

Фрагмент кода для приложения на рис. 2.5:

```
String e1[] = {"элемент 1", "элемент 2", "элемент 3", "элемент 4"};
String e2[] = {"5-й элемент", "6-й элемент", "7-й элемент",
                "8-й элемент"};

JComboBox jcb = new JComboBox(e1);
jcb.setPreferredSize(new Dimension(100, 30));
add(jcb);

JList jl = new JList(e2);
jl.setPreferredSize(new Dimension(300, 200));
add(jl);
```

В данном примере стоит обратить внимание на возможность передать в конструкторы «списочных» виджетов ссылки на массивы данных.

`JTextField` используется для ввода и редактирования однострочного текста. Для обработки текстовых данных, введенных пользователем, обычно используют слушатель события действия, который получает событие после нажатия пользователем клавиши Enter.

`JPasswordField` представляет собой однострочный текстовый редактор. Работа с компонентом производится аналогично, как и с `JTextField`. Единственным отличием данного компонента от `JTextField` является то, что весь введенный в него текст заменяется специальными символами (звездочками или другими заданными символами). Обычно используется для ввода паролей.

`JTextArea` представляет собой многострочный текстовый редактор. Класс `JTextArea` является производным от `JTextField` и наследует все его методы, дополняя их методами для управления вводом нескольких строк. При вводе пользователь разделяет строки обычным способом, нажимая клавишу Enter на клавиатуре. Если текст формируется динамически в программе, то для разделения строк используется специальный управляющий символ «\n». `JTextArea` обычно помещается в панель (контейнер) с полосами прокрутки `JScrollPane`.

Фрагмент кода для приложения на рис. 2.6:

```
Dimension d = new Dimension(400, 20);
JLabel lb = new JLabel("Текстовое поле:");
lb.setPreferredSize(d);
add(lb);
JTextField jtf = new JTextField("Текст");
jtf.setPreferredSize(d);
add(jtf);
lb = new JLabel("Поле для ввода пароля:");
lb.setPreferredSize(d);
add(lb);
JPasswordField jpf = new JPasswordField("Пароль");
```

```
jpf.setPreferredSize(d);
add(jpf);
lb = new JLabel("Многострочный текстовый редактор:");
lb.setPreferredSize(d);
add(lb);
JScrollPane jsp = new JScrollPane();
JTextArea ja = new JTextArea();
ja.setColumns(20);
ja.setRows(5);
ja.setText("H\ne\n1\n1\no\nw\no\nr\n1\nd\n!");
jsp.setPreferredSize(new Dimension(400, 150));
jsp.setViewportView(ja);
add(jsp);
```

Пример отображения компонентов `JTextField`, `JPasswordField`, `JTextArea` и `JScrollPane` представлен на рис. 2.6.

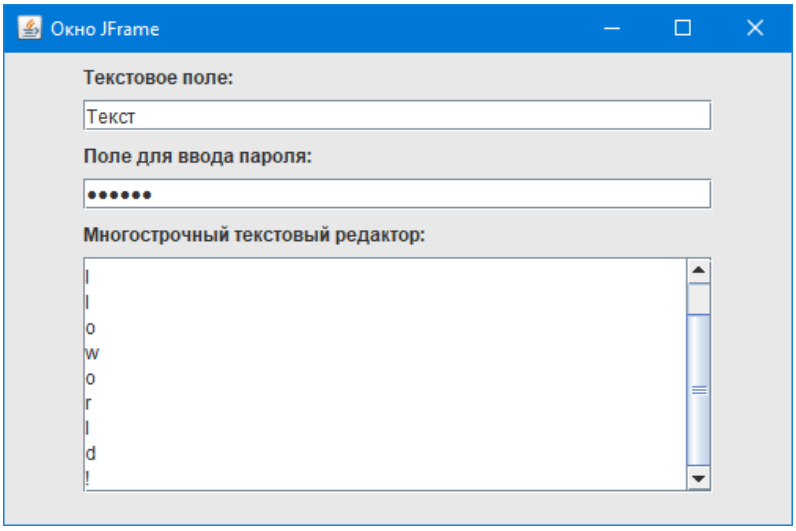


Рис. 2.6. Компоненты JTextField, JPasswordField, JTextArea и JScrollPane

Перед размещением объекта `JTextArea` на панель с полосами прокрутки `JScrollPane` у него последовательно вызываются методы `setColumns()` и `setRows()` для указания начального значения количества отображаемых столбцов и строк. Далее вызовом `setText()` устанавливается начальное значение отображаемого текста. Метод `setViewportView()` объекта `JScrollPane` используется для указания элемента, который будет отображаться в области прокрутки.

`JScrollBar` представляет собой отдельную полосу прокрутки (ползунок, область его движения и управляющие движением ползунка кнопки). Используется дополнительно для некоторых компонентов, если нет возможности использовать `JScrollPane`.

`JSlider` представляет собой элемент для изменения значения в заданном диапазоне. Внешний вид компонента: шкала значений заданного диапазона (числа, изображения «засе-

чек» или другие пиктограммы) и ползунок. Для управления ползунком используется перетаскивание мышью или клик на шкале ползунка. Для работы ползунка используется информация о минимальном значении, максимальном значении, текущем значении и внутреннем диапазоне.

`JProgressBar` представляет собой индикатор процесса (например, некоторого вычисления), который позволяет наглядно отображать его выполнение. Пользователь может оценить время исполнения некоторого процесса, прогнозировать его завершение и т.д. Внешний вид компонента: непрерывно меняющаяся (заполняющаяся) полоса со значением (в процентах), насколько выполнен процесс. В отличие от `JSlider` индикатор процесса `JProgressBar` использует лишь три значения: минимальное и максимальное значения задаются перед выполнением процесса, а текущее значение должно обновляться по мере выполнения процесса.

`JSpinner` представляет собой счетчик, который позволяет изменять текущее значение путем нажатия на соответствующие кнопки (увеличения и уменьшения значения). Внешне компонент похож на список `JComboBox` с одним выбранным элементом списка, но нажатие не раскрывает список, а изменяет значение в области отображения. Данное свойство счетчика позволяет использовать неограниченное количество элементов. Для счетчика `JSpinner` можно узнать его текущее, предыдущее и следующее значения, а также заменить текущее значение новым.

`JSeparator` представляет собой вертикальную или горизонтальную черту (разделитель), используется для графического разделения различных по структуре или функциям областей программы (например, пунктов меню).

Фрагмент кода для приложения на рис. 2.7:

```
add(new JLabel("JSlider:"));
JSlider jsl = new JSlider();
jsl.setPreferredSize(new Dimension(400, 40));
jsl.setValue(33);
add(jsl);

JSeparator js = new JSeparator();
js.setPreferredSize(new Dimension(500, 10));
add(js);

add(new JLabel("JSpinner:"));
JSpinner jsp = new JSpinner();
jsp.setPreferredSize(new Dimension(100, 40));
jsp.setValue(60);
add(jsp);

js = new JSeparator();
```

```
        js.setPreferredSize(new Dimension(500, 10));
        add(js);

        add(new JLabel("JProgressBar:"));
        JProgressBar jpb = new JProgressBar();
        jpb.setPreferredSize(new Dimension(400, 40));
        jpb.setValue(81);
        add(jpb);

        js = new JSeparator();
        js.setPreferredSize(new Dimension(500, 10));
        add(js);

        add(new JLabel("JScrollBar:"));
        JScrollBar jsb = new JScrollBar();
        jsb.setOrientation(JScrollBar.HORIZONTAL);
        jsb.setPreferredSize(new Dimension(400, 20));
        jsb.setValue(50);
        add(jsb);
```

Пример отображения компонентов `JSlider`, `JSpinner`, `JProgressBar`, `JScrollBar` и `JSeparator` представлен на рис. 2.7.

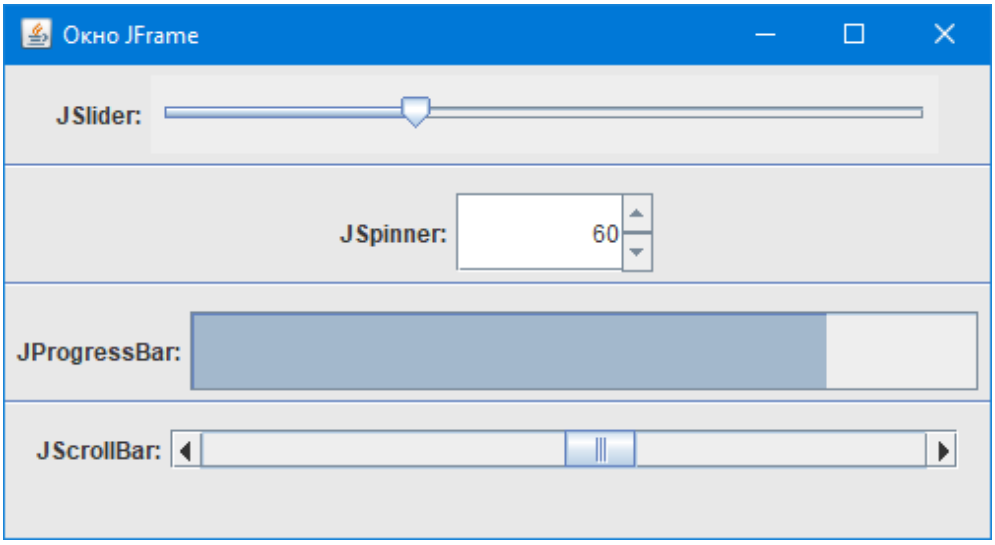


Рис. 2.7. Компоненты JSlider, JSpinner, JProgressBar, JScrollBar и JSeparator

Метод `setValue()` для использованных в примере компонентов меняет их текущее значение (внутреннее состояние), что также приводит к обновлению внешнего вида виджета (положение ползунка, шкалы или текста). Метод `setOrientation()` для объекта `JScrollBar` задает внешний вид компонента – направление движения (горизонтальное или вертикальное) полосы прокрутки.

2.6. МОДЕЛЬ ДЕЛЕГИРОВАНИЯ СОБЫТИЙ

Подход к обработке событий в Java основан на модели делегирования событий, определяющей стандартные и согласованные механизмы для генерации и обработки событий (рис. 2.8). При этом логика обработки события отделяется от логики пользовательского интерфейса, генерирующего эти события. Каждый элемент пользовательского интерфейса может «делегировать» обработку события отдельному фрагменту кода.

Концепция модели делегирования событий включает совместную работу трех объектов: **источника**, **слушателя** и **события**.

Источник генерирует событие и посылает его одному или более слушателям. Слушатель ожидает до тех пор, пока не получит событие. Как только событие получено, слушатель обрабатывает его и возвращает управление.

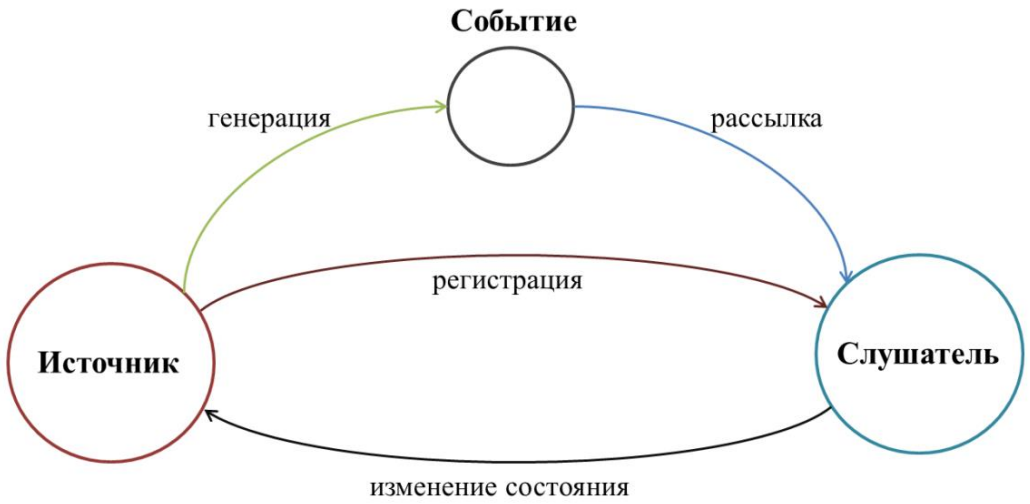


Рис. 2.8. Модель делегирования событий

Под событием в данной модели понимается объект, инкапсулирующий изменение состояния источника. Событие генерируется в результате взаимодействия пользователя с элементом в графическом интерфейсе (например, клик на экранной кнопке, ввод символа с клавиатуры, выбор элемента в списке или клик кнопкой мыши).

Под источником в данной модели понимается объект, генерирующий событие. Генерация события происходит при изменении внутреннего состояния объекта. Источники могут генерировать более одного типа событий. Для обработки событий источник должен регистрировать слушателей. Каждый тип события имеет собственный метод регистрации. Общая форма регистрации слушателя:

```
public void addTypeListener(TypeListener el)
```

где *Type* – это имя объекта события, *el* – ссылка на слушателя события. Например, `addKeyListener()` – регистрация слушателя событий клавиатуры, `addMouseMotionListener()` – регистрация слушателя движения мыши. Когда событие

сгенерировано, все зарегистрированные слушатели получают копию объекта события. Это называется групповой рассылкой события.

Источник также может отменить регистрацию слушателя для определенного типа событий. Общая форма отмены регистрации слушателя:

```
public void removeTypeListener(TypeListener el)
```

Под слушателем понимается объект, уведомляемый о возникновении события и выполняющий определенный набор действий (обработку события). Слушатель должен быть зарегистрирован одним или более источниками событий, чтобы получать уведомления о событиях определенного рода. А также должен реализовать методы для получения и обработки данных событий.

Основные типы объектов событий: `MouseEvent`, который генерируется при перетаскивании, перемещении, щелчках, нажатии и отпускании кнопок мыши, а также возникает, когда курсор мыши входит на компонент либо покидает его; `KeyEvent`, который генерируется при получении ввода с клавиатуры; `ActionEvent`, который генерируется при нажатии кнопки, двойном щелчке на элементе списка либо выборе пункта меню; `ItemEvent`, который генерируется при щелчке на флажке или элементе списка, при выборе элемента списка, отметке или снятии отметки пункта меню; `TextEvent`, который генерируется при изменении значения текстовой области или текстового поля.

Основными интерфейсами слушателей являются:

- `MouseListener` – содержит метод для обработки клика мыши (нажатия и отпускания кнопки мыши в одной и той же точке) `mouseClicked()`, метод для обработки события перехода мыши в область отображения компонента `mouseEntered()`, метод для обработки события перехода мыши из области отображения компонента `mouseExited()`, метод для обработки нажатия кнопки мыши `mousePressed()` и метод для обработки отпускания кнопки мыши `mouseReleased()`. Каждый метод этого слушателя принимает событие `MouseEvent`;
- `KeyListener` – содержит метод для обработки нажатия клавиши клавиатуры `keyPressed()`, метод для обработки отпускания клавиши клавиатуры `keyReleased()` и метод для обработки печати (ввода) символа (только для печатных символов) `keyTyped()`. Каждый метод данного слушателя принимает событие `KeyEvent`;
- `ActionListener` – содержит один метод для обработки события действия `actionPerformed()`, который принимает событие `ActionEvent`;
- `ItemListener` – содержит один метод для обработки события изменения состояния элемента `itemStateChanged()`, который принимает событие `ItemEvent`;
- `MouseMotionListener` – содержит метод для обработки перемещения курсора мыши `mouseMoved()` и метод для обработки перетаскивания курсора мыши `mouseDragged()`. Каждый метод этого слушателя принимает событие `MouseEvent`;

- `TextListener` – содержит один метод для обработки события изменения содержимого текстового поля или текстовой области `textChanged()`, который принимает событие `TextEvent`.

Использование модели делегирования событий заключается в реализации соответствующего интерфейса в слушателе и его последующей регистрации источником как получателя уведомлений о событии. Так как источник может генерировать несколько типов событий, он может зарегистрировать отдельно для каждого события собственного слушателя. К тому же слушатель может подписаться на получение нескольких типов событий и должен реализовать все интерфейсы, необходимые для получения этих событий.

Для того чтобы реализовывать только нужные методы перечисленных интерфейсов слушателей, можно использовать классы-адаптеры, к которым уже подключены соответствующие интерфейсы и есть пустая реализация их методов. В таком случае в метод регистрации слушателя передается ссылка на объект наследника класса-адаптера с необходимыми переопределенными методами или анонимный вложенный класс-адаптер с определением нужных методов.

Например, приложение, регистрирующее изменение координат курсора мыши при его движении и позволяющее перетаскивать компонент `JButton` в области отображения панели `JPanel`:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JavaGUIEvnt extends JFrame {

  private JPanel pane;
  private JButton btn;
  private JTextField field1, field2;
  private int btX, btY;

  JavaGUIEvnt() {
    initGUI();
  }

  private void initGUI() {
    field1 = new JTextField();
    btn = new JButton("D&D");
    pane = new JPanel();
    field2 = new JTextField();

    setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    getContentPane().setPreferredSize(new Dimension(500, 500));
    getContentPane().setBackground(new Color(100, 100, 100, 10));
```

```
        setTitle("Mouse Events");
        setLayout(new FlowLayout());

        btn.setPreferredSize(new Dimension(60, 60));
        btn.addMouseListener(new MouseAdapter() {
          @Override
          public void mousePressed(MouseEvent me) {
            btX = me.getX();
            btY = me.getY();
          }
        });
        btn.addMouseMotionListener(new MouseAdapter() {
          @Override
          public void mouseDragged(MouseEvent me) {
            btn.setLocation(btn.getX() + (me.getX() - btX),
                            btn.getY() + (me.getY() - btY));
          }
        });

        pane.setPreferredSize(new Dimension(498, 400));
        pane.setBackground(new Color(255, 255, 255));
        pane.add(btn);
        pane.addMouseMotionListener(new MouseAdapter() {
          @Override
          public void mouseMoved(MouseEvent me) {
            field1.setText(String.valueOf(me.getX()));
            field2.setText(String.valueOf(me.getY()));
          }
        });
        add(pane);

        add(new JLabel("X: "));
        field1.setPreferredSize(new Dimension(100, 30));
        add(field1);
        add(new JLabel("Y: "));
        field2.setPreferredSize(new Dimension(100, 30));
        add(field2);

        pack();
      }

      public static void main(String args[]) {
        EventQueue.invokeLater(() -> {
          new JavaGUIEvnt().setVisible(true);
        });
      }
    }
```

Внешний вид данного приложения приведен на рис. 2.9.

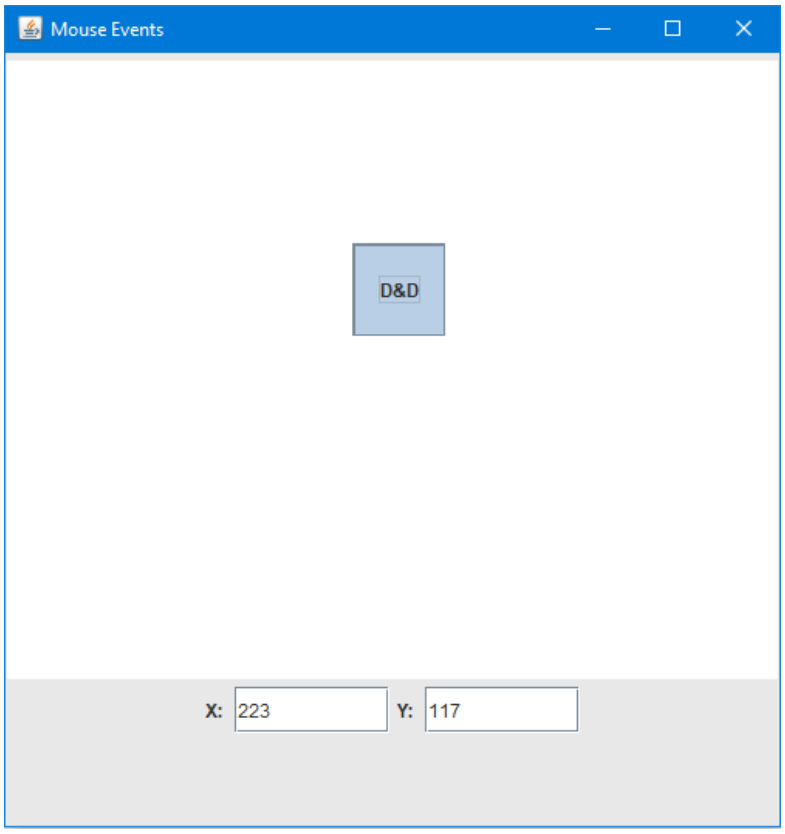


Рис. 2.9. Обработка событий курсора мыши

Для объекта `pane` типа `JPanel` с помощью метода `addMouseMotionListener()` в качестве слушателя событий курсора мыши зарегистрирован объект анонимного вложенного класса-адаптера `MouseAdapter`. В классе слушателя переопределен метод `mouseMoved()`, который получает ссылку на объект события `MouseEvent` при движении курсора мыши. Методы объекта события `getX()` и `getY()` возвращают текущие координаты позиции курсора мыши. Значения координат преобразуются в текст и помещаются в соответствующие текстовые поля.

Для объекта `btn` типа `JButton` зарегистрированы два типа слушателей `MouseListener` и `MouseMotionListener`, однако для каждого из них используется один тип адаптера `MouseAdapter` (что лишний раз демонстрирует удобство их использования). В методе `mousePressed()`, который получает объект события `MouseEvent` при нажатии кнопки мыши, текущие значения координат курсора мыши сохраняются в полях `btX` и `btY`. В методе `mouseDragged()`, который получает объект события `MouseEvent` при перетаскивании (перемещении с зажатой левой кнопкой) курсора мыши, обновляется положение объекта `btn` относительно своего контейнера `pane`. Устанавливается новое положение вызовом у `btn` метода `setLocation()`, который получает новые координаты левого верхнего угла виджета.

2.7. СРЕДСТВА Java 2D API

В основе Java 2D API находится класс `java.awt.Graphics`. Это абстрактный базовый класс для всех графических контекстов, которые позволяют приложению рисовать на визуальных компонентах, а также формировать графические изображения.

Поскольку `Graphics` является абстрактным классом, приложения не могут напрямую вызывать его конструктор. Графические контексты получают из других графических контекстов или создают путем вызова метода `getGraphics()` для компонента. Полученный объект может использоваться разработчиками для изменения внешнего вида компонентов в соответствии с используемым дизайном графического интерфейса приложения.

Например, код для получения графического контекста компонента `comp`:

```
java.awt.Graphics g = comp.getGraphics();
```

Когда запускается приложение Java с графическим пользовательским интерфейсом, за короткий промежуток времени может быть создано большое количество объектов `Graphics`. Хотя процесс финализации сборщика мусора освобождает системные ресурсы, используемые графическими контекстами, предпочтительнее освобождать связанные ресурсы вручную, вызывая у объекта метод `dispose()`. После вызова этого метода объект `Graphics` использовать нельзя. Так же стоит отметить, что для повышения эффективности приложения вызывать `dispose()` после завершения использования объекта нужно только в том случае, если он был получен непосредственно из компонента или создан из другого объекта `Graphics`. В случае, когда графические объекты передаются в качестве аргументов методам рисования и обновления компонентов, они освобождаются системой автоматически при возврате этих методов.

Например, освобождение связанных ресурсов с графическим объектом из предыдущего фрагмента:

```
g.dispose();
```

Для вывода пользовательской графики обычно создается отдельный элемент управления, который затем размещается на главное окно. В нем перегружается метод `paintComponent()` для графических элементов библиотеки Swing, или методы `paint()` и `update()` для элементов из библиотеки AWT.

Например, использование в качестве вывода пользовательской графики компонента `JPanel` из библиотеки Swing:

```
public class MyGraphic extends javax.swing.JPanel {
    @Override
    protected void paintComponent(java.awt.Graphics g) {
        // работа с методами графического контекста g
    }
}
```

Для отрисовки исходного внешнего вида компонента перед его изменением можно сделать вызов соответствующего метода отрисовки базового класса, передав тому текущий графический контекст:

```
super.paintComponent(g);
```

Большинство методов класса `Graphics` можно разделить на две основные группы:

1) методы настройки атрибутов, которые влияют на то, как отображается этот рисунок и заливка (например, `setFont()` или `setColor()`);

2) методы рисования и заливки, позволяющие отображать основные фигуры, текст и изображения (например, `drawString()`, `drawImage()` или `drawLine()`).

Пример использования основных методов рисования класса `Graphics` приведен на рис. 2.10.

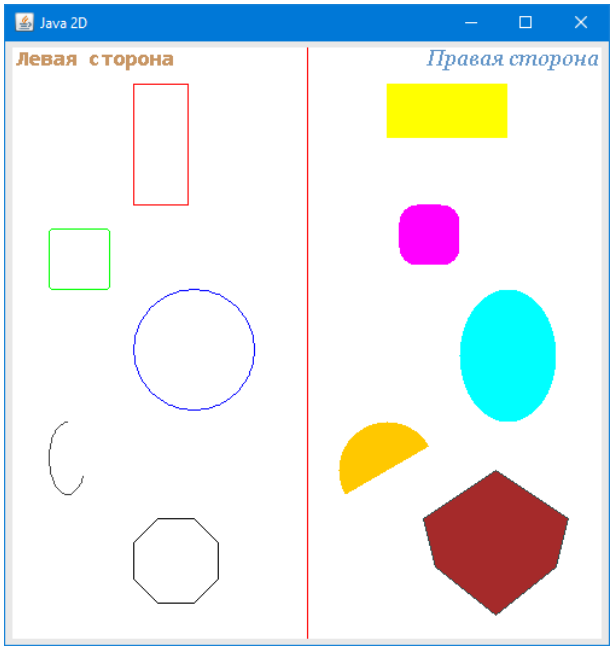


Рис. 2.10. Использование основных методов отрисовки класса Graphics

Код из метода `paintComponent()`, переопределенный для элемента JPanel в данном приложении:

```
super.paintComponent(g);
g.setColor(Color.red);
g.drawLine(244, 0, 244, 490);

g.setColor(new Color(200, 150, 100));
g.setFont(new Font("Consolas", Font.BOLD, 18));
g.drawString("Левая сторона", 3, 15);
g.setColor(Color.red);
g.drawRect(100, 30, 45, 100);
g.setColor(Color.green);
g.drawRoundRect(30, 150, 50, 50, 5, 5);
g.setColor(Color.blue);
```

```
g.drawOval(100, 200, 100, 100);
g.setColor(Color.darkGray);
g.drawArc(30, 310, 30, 60, 90, 240);
int x1[] = {100, 120, 150, 170, 170, 150, 120, 100};
int y1[] = {410, 390, 390, 410, 440, 460, 460, 440};
g.setColor(Color.black);
g.drawPolygon(x1, y1, 8);
g.setColor(new Color(100, 150, 200));
g.setFont(new Font("Georgia", Font.ITALIC, 18));
g.drawString("Правая сторона", 343, 15);
g.setColor(Color.yellow);
g.fillRect(310, 30, 100, 45);
g.setColor(Color.magenta);
g.fillRoundRect(320, 130, 50, 50, 30, 30);
g.setColor(Color.cyan);
g.fillOval(370, 200, 80, 110);
g.setColor(Color.orange);
g.fillArc(270, 310, 80, 80, 30, 180);
int x2[] = {340, 400, 460, 450, 400, 350};
int y2[] = {390, 350, 390, 430, 470, 430};
g.setColor(new Color(165, 42, 42));
g.fillPolygon(x2, y2, 6);
g.setColor(new Color(47, 79, 79));
g.drawPolygon(x2, y2, 6);
```

Метод `void setColor(Color c)` устанавливает текущий цвет графического контекста в цвет `c`. Все последующие графические операции в данном контексте будут использовать указанный цвет. Для задания цвета можно использовать статические поля класса `Color` этого же типа (например, `Color.red`) или создать объект, используя один из конструкторов. Например, конструктор `Color(int r, int g, int b)` создает непрозрачный цвет sRGB с указанными значениями красного `r`, зеленого `g` и синего `b` в диапазоне от 0 до 255.

Метод `void drawLine(int x1, int y1, int x2, int y2)` рисует линию, используя текущий цвет, между точками (`x1`, `y1`) и (`x2`, `y2`) в системе координат графического контекста (точка отсчета (0, 0) – левый верхний угол графического контекста по умолчанию для каждого компонента).

Метод `void setFont(Font font)` устанавливает шрифт графического контекста в шрифт `font`. Все последующие текстовые операции с использованием этого графического контекста используют этот шрифт. Создать объект шрифта можно, используя один из конструкторов класса `Font`. В данном примере используется конструктор `Font(String name, int style, int size)`, который создает новый шрифт с указанным именем, стилем и размером. Имя шрифта `name` может быть именем начертания шрифта (например, «Times

New Roman») или названием семейства шрифтов (например, «Arial»). Аргумент `style` представляет собой целочисленную битовую маску, которая может быть равна `Font.PLAIN` (по умолчанию, обычный стиль), `Font.BOLD` (полужирное начертание) и(или) `Font.ITALIC` (курсивное начертание). Указанный стиль объединяется со стилем указанного в `name` шрифта, а не заменяется или вычитается. Аргумент `size` задает размер шрифта в пунктах.

Метод `void drawString(String str, int x, int y)` рисует текст, заданный строкой `str`, используя текущий шрифт и цвет этого графического контекста. Базовая линия самого левого символа находится в позиции с координатами (`x`, `y`).

Метод `void drawRect(int x, int y, int width, int height)` рисует контур прямоугольника, у которого левая верхняя точка задана координатами (`x`, `y`), ширина – аргументом `width`, а высота – аргументом `height`. Прямоугольник рисуется с использованием текущего цвета графического контекста.

Метод `void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)` рисует контур прямоугольника с закругленными углами, используя текущий цвет графического контекста. Левая верхняя точка прямоугольника задана координатами (`x`, `y`), ширина – аргументом `width`, высота – аргументом `height`. Аргумент `arcWidth` задает горизонтальный диаметр дуги в углах прямоугольника, а `arcHeight` – вертикальный.

Метод `void drawOval(int x, int y, int width, int height)` рисует контур овала. Результатом является круг или эллипс, который помещается в прямоугольник, заданный аргументами `x`, `y`, `width` и `height`.

Метод `void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)` рисует контур дуги окружности или эллипса. Центр дуги – это центр прямоугольника, левая верхняя точка которого задана координатами (`x`, `y`), а размер определяется аргументами ширины `width` и высоты `height`. Таким образом, дуга – это видимая часть того овала, для построения которого используются первые четыре параметра. Дуга начинается с угла `startAngle` и имеет угловой размер `arcAngle`. Значения углов задаются в градусах. Начальный угол (`startAngle = 0`) соответствует направлению часовой стрелки, указывающей на «3 часа». Положительное значение указывает на вращение против часовой стрелки, а отрицательное значение указывает на вращение по часовой стрелке. Например, при значениях `startAngle = 0` и `arcAngle = 90` контур дуги будет занимать верхнюю правую четверть овала.

Метод `void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)` рисует замкнутый многоугольник, точки которого определены значениями массивов координат `xPoints` и `yPoints`. Каждая пара элементов массивов с одинаковыми индексами (`xPoints[i]`, `yPoints[i]`) определяет точку. Количество точек, используемых для отри-

совки полигона, определяется значением аргумента nPoints. Фигура автоматически закрывается путем рисования линии, соединяющей конечную точку с первой точкой, если эти точки различны.

Методы fillRect(), fillRoundRect(), fillOval(), fillArc() и fillPolygon() предназначены для рисования тех же графических примитивов, но контур фигуры при этом не отрисовывается, а выполняется ее заливка текущим цветом графического контекста.

Для использования в рисовании уже готовых изображений, хранящихся в виде графических файлов, можно использовать метод drawImage(). Например, фрагмент кода, в котором на JPanel выводятся изображения из трех файлов:

```
BufferedImage img;
try {
  img = ImageIO.read(new File("tree1.png"));
  g.drawImage(img, 5, 100, this);
  img =.ImageIO.read(new File("tree2.png"));
  g.drawImage(img, 175, 100, this);
  img = ImageIO.read(new File("tree3.png"));
  g.drawImage(img, 320, 100, this);
} catch(IOException e) {
    e.printStackTrace();
  }
```

Используемая форма метода boolean drawImage(Image img, int x, int y, ImageObserver observer) рисует изображение img в исходных размерах. Левый верхний угол изображения задан координатами точки (x, y) в координатном пространстве используемого графического контекста. Прозрачные пиксели в изображении не влияют на уже существующие пиксели. Если изображение полностью загружено и его пиксели больше не изменяются, то drawImage() возвращает значение true.

Для инкапсуляции изображения используется производный от Image класс BufferedImage, который имеет буфер для обработки и управления данными изображения. Объект типа BufferedImage возвращает статический метод read() класса ImageIO. Данный метод формирует объект изображения из графического файла, путь к которому получает в качестве параметра. Путь к файлу инкапсулирует объект типа File. Класс BufferedImage находится в библиотеке классов java.awt.image, класс ImageIO – в javax.imageio, класс File – в java.io. Так как метод ImageIO.read() может генерировать исключение IOException, его вызов необходимо осуществлять в блоке try-catch.

Внешний вид окна для приведенного выше фрагмента кода приведен на рис. 2.11.

Динамическую работу с графическим контекстом можно организовать, используя модель делегирования событий, так как каждый виджет способен генерировать основные типы событий.

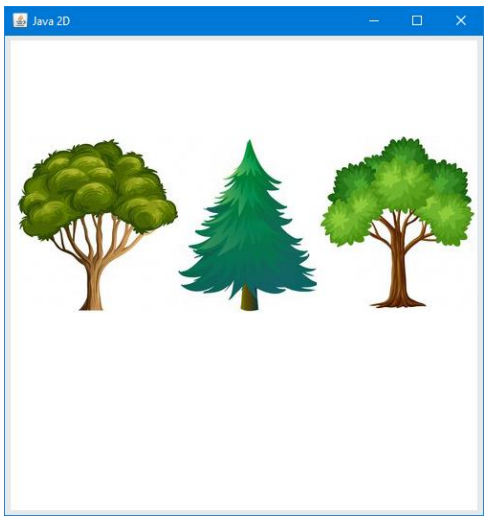


Рис. 2.11. Использование метода drawImage() класса Graphics

Пример приложения для динамического формирования полигона:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class UserJPanel extends JPanel {

  private int[] xPoints, yPoints;
  private int numPoints;
  boolean hidePoints;

  UserJPanel(int num) {
    xPoints = new int[num];
    yPoints = new int[num];
    numPoints = 0;
    hidePoints = false;
  }

  public void setPoint(int x, int y) {
    xPoints[numPoints] = x;
    yPoints[numPoints] = y;
    numPoints++;
  }

  public void resetPoint() {
    numPoints--;
  }

  public void hsPoints() {
    hidePoints = !hidePoints;
  }
```

```
    @Override
    public void paintComponent(Graphics g) {
      super.paintComponent(g);
      if(numPoints > 0) {
        g.setColor(new Color(176, 196, 222));
        g.fillPolygon(xPoints, yPoints, numPoints);
        g.setColor(new Color(0, 180, 0));
        if(!hidePoints)
          for(int i = 0; i < numPoints; ++i) {
            g.fillOval(xPoints[i] - 3, yPoints[i] - 3, 7, 7);
          }
      }
    }
  }

  public class Java2D extends JFrame {

    private UserJPanel pane;

    Java2D() {
      initGUI();
    }

    private void initGUI() {
      pane = new UserJPanel(100);

      setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
      getContentPane().setPreferredSize(new Dimension(500, 500));
      getContentPane().setBackground(new Color(100,100,100,10));
      setTitle("Java 2D");
      setLayout(new FlowLayout());
      pane.setPreferredSize(new Dimension(488,490));
      pane.setBackground(new Color(255, 255, 255));
      pane.addMouseListener(new MouseAdapter() {
        @Override
        public void mousePressed(MouseEvent me) {
          pane.setPoint(me.getX(), me.getY());
          pane.updateUI();
        }
      });
     pane.addKeyListener(new KeyAdapter() {
        @Override
        public void keyPressed(KeyEvent ke) {
          if(ke.getKeyCode() == KeyEvent.VK_BACK_SPACE) {
            pane.resetPoint();
          }
```

```
            if(ke.getKeyCode() == KeyEvent.VK_H) {
              pane.hsPoints();
            }
            pane.updateUI();
          }
        });
        pane.setFocusable(true);
        add(pane);
        pack();
      }

      public static void main(String args[]) {
        EventQueue.invokeLater(() -> {
          new Java2D().setVisible(true);
        });
      }

    }
```

Для объекта `pane` типа `UserJPanel` зарегистрированы слушатели событий мыши и клавиатуры. В методе `mousePressed()` координаты курсора мыши добавляются как координаты новой точки полигона с помощью метода `setPoint()`. В методе `keyPressed()` используется информация о коде нажатой клавиши клавиатуры. При нажатии клавиши «Backspace» (код `KeyEvent.VK_BACK_SPACE`) удаляется последняя добавленная точка полигона. При нажатии клавиши «H» (код `KeyEvent.VK_H`) включается или отключается отображение точек полигона (меняется значение внутреннего поля `hidePoints` объекта `pane`). В каждом из перечисленных обработчиков также осуществляется перерисовка используемого графического контекста вызовом метода `updateUI()` для объекта `pane`.

Внешний вид приложения для динамического формирования полигона приведен на рис. 2.12.

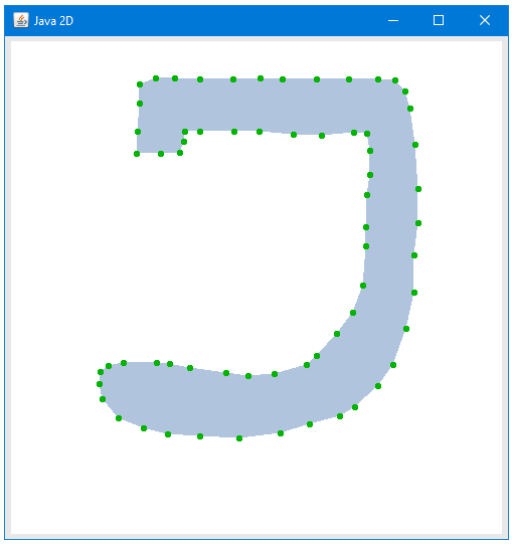


Рис. 2.12. Динамическое формирование полигона

ЗАКЛЮЧЕНИЕ

В учебном пособии изложены материалы, раскрывающие важные инструментарии – виджеты при организации интерактивных приложений, обрабатывающих графическую информацию и не только. Кроме этого, на простых примерах показаны все этапы в реализации компьютерной графики. Представленные два языка программирования Python и JAVA в качестве примера использования методов графического интерфейса и вывода компьютерной графики в приложениях пользователей выводят данное учебное пособие на востребованность в свете реалий последнего времени. Следует учесть, что рассматриваемые языки программирования являются диаметрально противоположными с точки зрения организации и реализации, но имеют схожие подходы к построению графического интерфейса пользователя написанных на них приложений.

Материалы, изложенные в учебном пособии, позволяют не только понять сущность виджетов и принципов формирования компьютерной графики, но и использовать их на практике при организации рабочих приложений научного и коммерческого характера.

СПИСОК ЛИТЕРАТУРЫ

1. **Лутц, М.** Изучаем Python / М. Лутц. – СПб. : Символ–Плюс, 2011. – 1280 с.

2. **Обухов, А. Д.** Алгоритмы обработки данных в автоматических системах управления на основе компьютерного зрения / А. Д. Обухов, К. И. Патутин, А. О. Назарова // Вестник Тамбовского государственного технического университета. – 2022. – Т. 28, № 4. – С. 573 – 585.

3. **Златопольский, Д.** Основы программирования на языке Python / Д. Златопольский. – Litres, 2019.

4. **Van Rossum, G.** PEP 8: Style Guide for Python Code / G. Van Rossum, B. Warsaw, N. Coghlan // Python. org. – 2001. – Т. 1565.

5. **Using** Tkinter of Python to Create Graphical User Interface (GUI) for Scripts in LNLS / Beniz D. et al. // WEPOPRPO25. – 2016. – Т. 9. – P. 25 – 28.

6. **Shipman, J. W.** Tkinter 8.4 Reference: a GUI for Python / J. W. Shipman // New Mexico Tech Computer Center. – 2013. – Т. 54.

7. **Dey, S.** Hands-On Image Processing with Python: Expert Techniques for Advanced Image Analysis and Effective Interpretation of Image Data / S. Dey. – Packt Publishing Ltd, 2018.

8. **Chityala, R.** Image Processing and Acquisition Using Python / R. Chityala, S. Pudipeddi. – CRC Press, 2020.

ОГЛАВЛЕНИЕ

Учебное электронное издание

ВАСИЛЬЕВ Сергей Александрович
ЕВДОКИМОВ Александр Александрович
ОБУХОВ Артем Дмитриевич

ПРОГРАММНЫЙ ИНСТРУМЕНТАРИЙ АНАЛИЗА И ОБРАБОТКИ ГРАФИЧЕСКОЙ ИНФОРМАЦИИ

Учебное пособие

Редактор Л. В. Комбарова
Графический и мультимедийный дизайнер Н. И. Кужильная
Обложка, упаковка, тиражирование Л. В. Комбаровой

ISBN 978-5-8265-2612-5

Подписано к использованию 03.07.2023.
Тираж 50 шт. Заказ № 72

Издательский центр ФГБОУ ВО «ТГТУ»
392000, г. Тамбов, ул. Советская, д. 106, к. 14
Тел./факс (4752) 63-81-08.
E-mail: izdatelstvo@tstu.ru