ПРОГРАММЫ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ

Особенности современных прикладных программ

Большинство современных программ, предназначенных для пользователей, управляются с помощью графического интерфейса. Вы, несомненно, знакомы с понятиями «окно программы», «кнопка», «выключатель», «поле ввода», «полоса прокрутки» и т.п. Такие оконные системы чаще всего построены на принципах объектно-ориентированного программирования, то есть все элементы окон – это объекты, которые обмениваются данными, посылая друг другу сообщения.

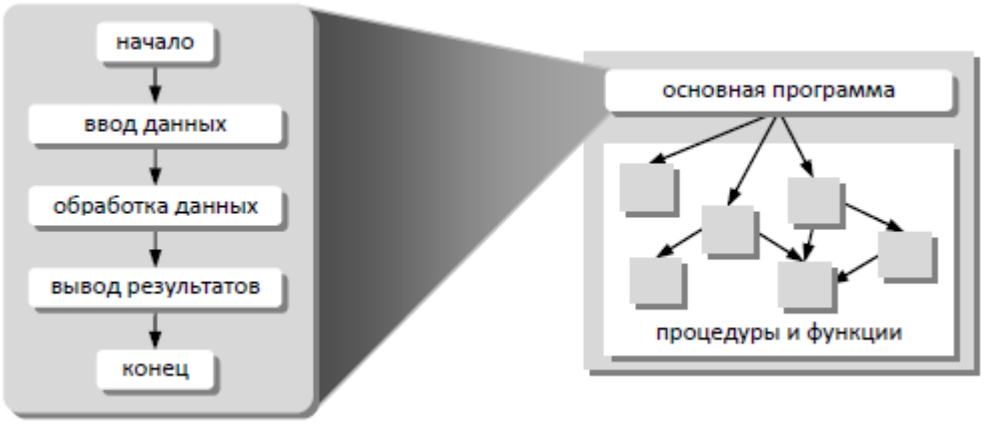
Сообщение – это блок данных определённой структуры, который используется для обмена информацией между объектами.

В сообщении указывается

- адресат (объект, которому посылается сообщение);
- числовой код (тип) сообщения;
- параметры (дополнительные данные), например, координаты щелчка мыши или код нажатой клавиши.

Сообщение может быть *широковещательным*, в этом случае вместо адресата указывается особый код и сообщение поступает всем объектам определенного типа (например, всем главным окнам программ).

В программах, которые мы писали раньше (см. рисунок), последовательность действия заранее определена — основная программа выполняется строчка за строчкой, вызывая процедуры и функции, все ветвления выполняются с помощью условных операторов.



В современных программах порядок действий определяется пользователем, другими программами или поступлением новых данных из внешнего источника (например, из сети), поэтому классическая схема не подходит. Пользователь текстового редактора может щелкать по любым кнопкам и выбирать любые пункты меню в произвольном порядке. Программа-сервер, передающая данные с Web-сайта на компьютер пользователя, начинает действовать только при поступлении очередного запроса. При программировании сетевых игр нужно учитывать взаимодействие многих объектов, информация о которых передается по сети в случайные моменты времени.

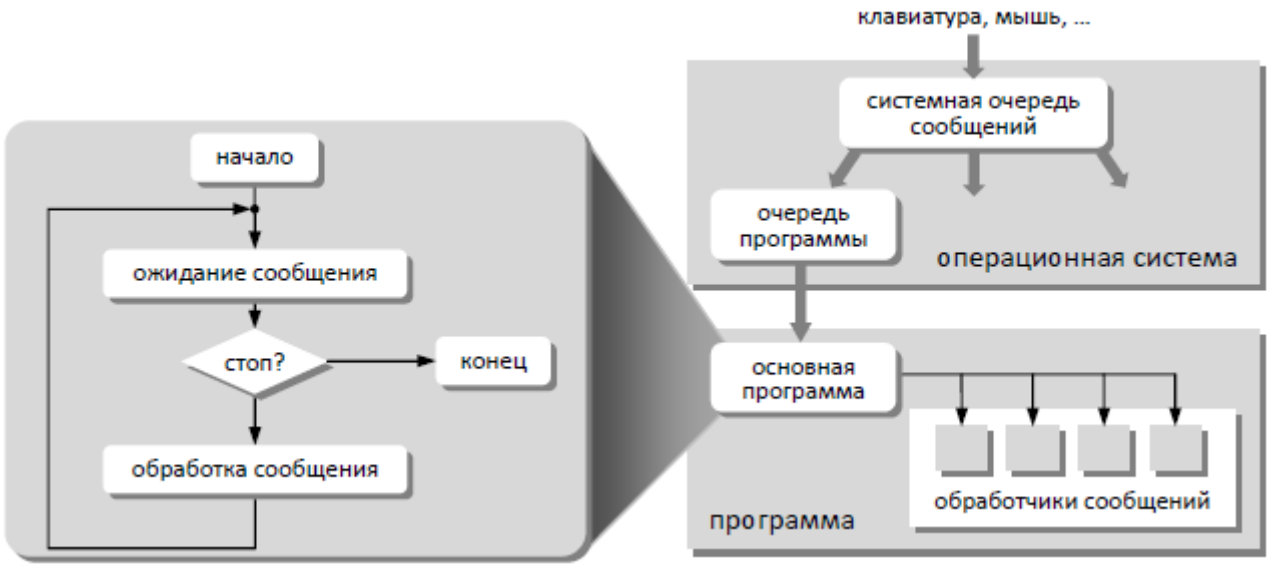
Во всех этих примерах программа должна «включаться в работу» только тогда, когда получит условный сигнал, то есть произойдет некоторое событие (изменение состояния).

Событие – это переход какого-либо объекта из одного состояния в другое.

События могут быть вызваны действиями пользователя (управление клавиатурой и мышью), сигналами от внешних устройств (переход принтера в состояние готовности, получение данных из сети), получением сообщения от другой программы. При наступлении событий объекты посылают друг другу сообщения.

Таким образом, весь ход выполнения современной программы определяется происходящими событиями, а не жестко заданным алгоритмом. Поэтому говорят, что программы управляются событиями, а соответствующий стиль программирования называют событийно-ориентированным, то есть основанным на обработке событий.

Нормальный режим работы событийно-ориентированной программы – цикл обработки сообщений. Все сообщения (от мыши, клавиатуры и драйверов устройств ввода и вывода и т.п.) сначала поступают в единую очередь сообщений операционной системы. Кроме того, для каждой программы операционная система создает отдельную очередь сообщений, и помещает в нее все сообщения, предназначенные именно этой программе.



Программа выбирает очередное сообщение из очереди и вызывает специальную процедуру –обработчик этого сообщения (если он есть). Когда пользователь закрывает окно программы, ей посылается специальное сообщение, при получении которого цикл (и работа всей программы) завершается.

Таким образом, главная задача программиста – написать содержание обработчиков всех нужных сообщений. Еще раз подчеркнем, что последовательность их вызовов точно не определена, она может быть любой в зависимости от действий пользователя и сигналов, поступающих с внешних устройств.

RAD-среды для разработки программ

Разработка программ для оконных операционных систем до середины 1990-х годов была довольно сложным и утомительным делом. Очень много усилий уходило на то, чтобы написать команды для создания интерфейса с пользователем: разместить элементы в окне программы, написать и правильно оформить обработчики сообщений. Значительную часть своего времени программист занимался трудоёмкой работой, которая почти никак не связана с решением главной задачи. Поэтому возникла естествен-

ная мысль — автоматизировать описание окон и их элементов так, чтобы весь интерфейс программы можно было построить без ручного программирования (чаще всего с помощью мыши), а человек думал бы о сути задачи, то есть об алгоритмах обработки данных.

Такие системы программирования получили название *RAD-сред* (от англ. *Rapid Application Development* — быстрая разработка приложений). Разработка программы в RAD-системе состоит из следующих этапов:

- создание формы (так называют шаблон, по которому строится окно программы или диалога); при этом минимальный код добавляется автоматически и сразу получается работоспособная программа;
- расстановка на форме элементов интерфейса (полей ввода, кнопок, списков) с помощью мыши и настройка их свойств;
- создание обработчиков событий;
- написание алгоритмов обработки данных, которые выполняются при вызове обработчиков событий.

Обратите внимание, что при программировании в RAD-средах обычно говорят не об обработчиках сообщений, а об обработчиках событий. Событием в программе может быть не только нажатие клавиши или щелчок мышью, но и перемещение окна, изменение его размеров, начало и окончание выполнения расчетов и т.д. Некоторые сообщения, полученные от операционной системы, библиотека RAD-среды «транслирует» (переводит) в соответствующие события, а некоторые – нет. Более того, программист может вводить свои события и определять процедуры для их обработки.

Одной из первых сред быстрой разработки стала среда *Delphi*, разработанная фирмой *Borland* в 1994 году. Самая известная современная профессиональная RAD-система — *Microsoft Visual Studio* – поддерживает несколько языков программирования. Существует свободная RAD-среду *Lazarus* (lazarus.freepascal.org), которая во многом аналогична *Delphi*, но позволяет создавать кроссплатформенные программы (для операционных систем *Windows*, *Linux*, *Mac OS X* и др.).

Среды RAD позволили существенно сократить время разработки программ. Однако нужно помнить, что любой инструмент — это только инструмент, который можно использовать грамотно и безграмотно. Использование среды RAD само по себе не гарантирует, что у вас автоматически получится хорошая программа с хорошим пользовательским интерфейсом.

ГРАФИЧЕСКИЙ ИНТЕРФЕЙС: ОСНОВЫ

В этом разделе мы продемонстрируем основные принципы построения программ с графическим интерфейсом на языке Python. К сожалению, для этого языка не создано сколько-нибудь надёжных средств визуальной разработки приложений (RAD).

Мы будем использовать графическую библиотеку **`tkinter`**, которая входит в стандартную библиотеку Python. Существуют также и другие библиотеки для разработки интерфейсов на Python: wxPython, PyGTK, PyQt и некоторые другие. Их нужно устанавливать отдельно.

В программе с графическим интерфейсом может быть несколько окон, которые обычно называют формами. На форме размещаются элементы графического интерфейса: поля ввода, флажки, кнопки и др., которые называются виджетами (англ. *widget*

– украшение, элемент) или (как в большинстве аналогичных сред) компонентами. Каждый компонент – это объект определённого класса, у которого есть свойства и методы.

Для событий, которые происходят с компонентом, можно задать функции-обработчики. Они будут выполняться тогда, когда происходит соответствующее событие. Примеры таких событий – это изменение состояния флажка, изменение размеров формы, щелчок мышью на объекте, нажатие клавиши на клавиатуре.

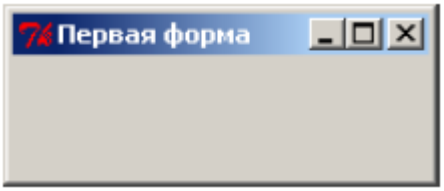
Далее в этой главе мы сосредоточимся на принципах проектирования программ с графическим интерфейсом, а не на особенностях библиотеки **tkinter**. Для этого мы будем применять так называемую «обёртку» **simpletk** – оболочку, которая скрывает сложности использования исходной библиотеки.

Простейшая программа

Простейшая программа с графическим интерфейсом состоит из трёх строк:

```
from simpletk import *                  # (1)
app = TApplication("Первая форма")      # (2)
app.Run()                               # (3)
```

Вероятно, вам понятна первая строка: из модуля **simpletk** импортируются все функции (для того, чтобы не нужно было каждый раз указывать название модуля). В строке 2 создаётся объект класса **TApplication** – это приложение (программа, от англ. *application* – приложение). Конструктору передаётся заголовок главного окна программы.



В последней строке с помощью метода **Run** (от англ. *run* – запуск) запускается цикл обработки сообщений, который скрыт внутри этого метода, так что здесь тоже использован принцип инкапсуляции (скрытия внутреннего устройства). Приведённую выше программу можно запустить, и вы должны увидеть окно, показанное на рисунке.

Свойства формы

Объект класса **TApplication** (наследник класса **Tk** библиотеки **tkinter**) имеет методы, позволяющие управлять свойствами окна. Например, можно изменить начальное положение окна на экране с помощью свойства **position**:

```
app.position = (100, 300)
```

Позиция окна – это *кортеж* (неизменяемый набор данных), состоящий из x-координаты и y-координаты левого верхнего угла окна.

Аналогично изменяются и размеры окна:

```
app.size = (500, 200)
```

Первый элемент кортежа – ширина, а второй – высота окна.

Свойство **resizable** (от англ. *изменяемый размер*) позволяет запретить изменение размеров окна по одному или обоим направлениям. Например, вызов

```
app.resizable = (True, False)
```

разрешает только изменение ширины, а изменить высоту окна будет невозможно.

С помощью свойств **minsize** и **maxsize** можно установить минимальные и максимальные размеры формы, например:

```
app.minsize = (100, 200)
```

Теперь ширина формы не может быть меньше 100 пикселей, а высота – меньше 200 пикселей.

Обработчик событий

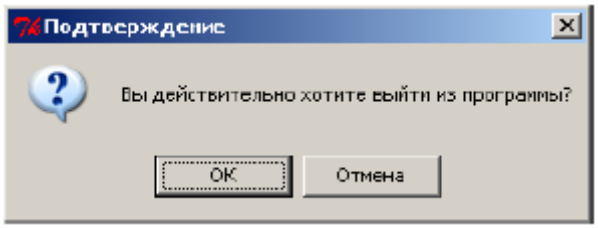
Рассмотрим простой пример. Вы знаете, что многие программы запрашивают подтверждение, когда пользователь завершает их работу. Для этого можно использовать обработчик события «удаление окна». Он устанавливается так:

```
app.onCloseQuery = AskOnExit
```

Здесь **onCloseQuery** – название обработчика события (от англ. on close query – «при запросе на закрытие»). Справа записано название функции **AskOnExit**, которая вызывается при этом событии. Эта функция определяет действия в том случае, когда пользователь закрывает окно программы, например, так:

```
def AskOnExit():
  app.destroy()
```

Всё действие этого обработчика сводится к вызову метода **destroy** (англ. разрушить): окно удаляется из памяти, и работа программы завершается. Нам нужно спросить пользователя, не ошибся ли он, то есть выдать ему сообщение в диалоговом окне:



и завершить работу программы только в том случае, когда он нажмёт на кнопку «ОК».

Для этого используем готовую функцию **askokcancel** (англ. *ask* – спросить, *OK* – кнопка «ОК», *cancel* – кнопка «Отмена») из модуля **messagebox** пакета **tkinter**. Импортировать эту функцию можно так:

```
from tkinter.messagebox import askokcancel
```

Теперь функция-обработчик принимает следующий вид:

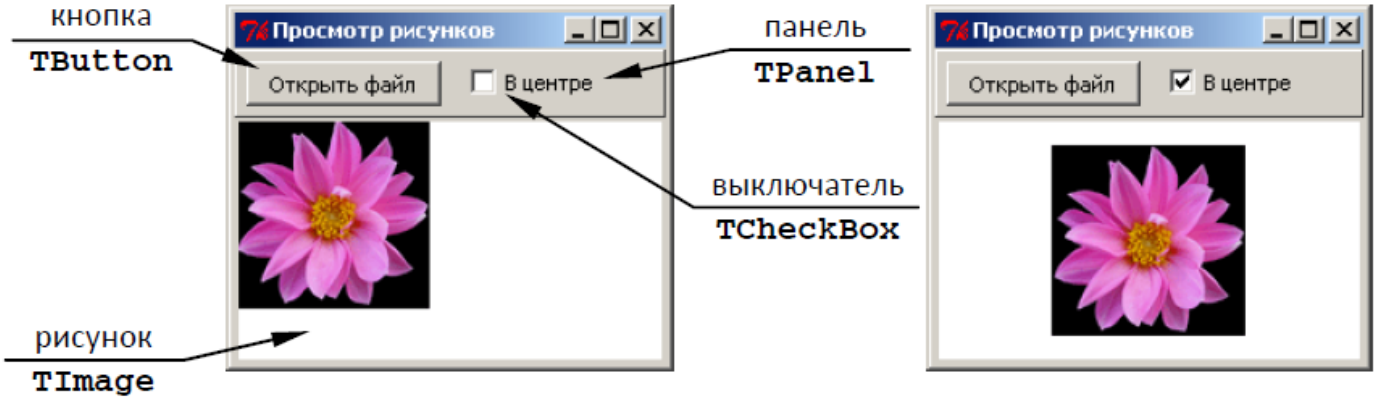
```
def AskOnExit():
  if askokcancel ( "Подтверждение",
     "Вы действительно хотите выйти из программы?" ):
    app.destroy()
```

Функция **askokcancel** принимает два аргумента: заголовок окна и сообщения для пользователя. Она возвращает ненулевое значение, если пользователь нажат кнопку «ОК». В этом случае срабатывает условный оператор и вызывается метод **destroy**, завершающий работу программы.

ИСПОЛЬЗОВАНИЕ КОМПОНЕНТОВ (ВИДЖЕТОВ)

Программа с компонентами

Теперь построим программу для просмотра файлов, которая умеет открывать файлы с диска и показывать их в нижней части окна в левом верхнем углу или по центру свободной области:



К сожалению, наша программа будет работать только с файлами формата GIF, поскольку с остальными распространёнными форматами (JPEG, PNG, BMP) библиотека **tkinter** работать не умеет.

На форме размещены четыре компонента:

- кнопка с надписью «Открыть файл» (компонент **TButton**, наследник класса **Button** библиотеки **tkinter**);
- выключатель с текстом «В центре» (компонент **TCheckBox**, наследник класса **Checkbox** библиотеки **tkinter**);
- панель, на которой лежат кнопка и выключатель (компонент **TPanel**, наследник класса **Frame** библиотеки **tkinter**);
- поле для рисунка, заполняющее все остальное место на форме (компонент **TImage**, наследник класса **Canvas** библиотеки **tkinter**).

Начнем с простой программы, которая устанавливает свойства главного окна:

```
from simpletk import *
app = TApplication ( "Просмотр рисунков" )
app.position = (200, 200)
app.size = (300, 300)
app.Run()
```

Теперь построим верхнюю панель:

```
panel = TPanel ( app, relief = "raised", height = 35, bd = 1 )
```

Для создания объекта-панели вызывается конструктор класса **TPanel**. Ему передаётся несколько параметров:

- **app** – ссылка на родительский объект;
- **relief** – объёмный вид, значение **"raised"** означает «приподнятый»; этот параметр может принимать также значения **"flat"** (плоский, по умолчанию), **"sunken"** («утопленный») и др.;
- **height** – высота в пикселях;
- **bd** (от англ. *border* - граница) – толщина границы в пикселях.

Родительский объект – это объект, отвечающий за перемещение и вывод на экран нашей панели,
которая становится «дочерним» объектом для главного окна.
Расположим панель на форме, прижав её к верхней границе с помощью свойства align:

```
panel.align = "top"
```

Это свойство задаёт расположение панели внутри родительского окна, "**top**" означает «прижать вверх» ("**bottom**" – вниз, "**left**" – влево, "**right**" – вправо).

Теперь нужно разместить на панели компоненты **TButton** (кнопка) и **TCheckBox** (выключатель). Для них «родителем» уже будет не главное окно, а панель **panel**. Начнём с кнопки:

```
openBtn = TButton ( panel, width = 15, text = "Открыть файл" )
openBtn.position = (5, 5)
```

Для создания кнопки вызван конструктор класса **TButton**. Первый параметр – это родительский
объект (панель), параметр **width** задаёт ширину кнопки в символах (не в пикселях!), а параметр
text – надпись на кнопке. В следующей строке с помощью свойства **position** определяются
координаты кнопки на панели.

Используя тот же подход, создаём и размещаем выключатель:

```
centerCb = TCheckBox ( panel, text = "В центре" )
centerCb.position = (115, 5)
```

Можно запустить программу и убедиться, что все компоненты появляются на форме, но пока не
работают (не реагируют на щелчки мыши).

В нижнюю часть формы нужно «упаковать» поле для рисунка – объект класса **TImage** – так,
чтобы он занимал все оставшееся место. Создаём такой объект:

```
image = TImage ( app, bg = "white" )
```

Его «родителем» будет главное окно **app**, параметр **bg** (от англ. **background** - фон) задаёт цвет
«холста» (англ. **white** - белый). Затем «упаковываем» его в окно так, чтобы он занимал все остав-
шееся место, то есть заполнял оставшуюся область по вертикали и горизонтали:

```
image.align = "client"
```

Теперь нужно задать обработчики событий. Нас интересуют два события:

- *щелчок по кнопке* (после него должен появиться диалог выбора файла);
- *переключение флажка-выключателя* (нужно изменить режим показа рисунка).

Сначала определим, какие действия нужно выполнить в случае щелчка мышью по кнопке. Псевдокод выглядит так:

```
выбрать файл с рисунком
if файл выбран:
  загрузить рисунок в компонент image
```

Выбрать файл можно с помощью стандартного диалога операционной системы, который вызывает функция **askopenfilename** из модуля **filedialog** библиотеки **tkinter**. Сначала этот модуль нужно импортировать:

```
from tkinter import filedialog
```

Теперь вызываем функцию **askopenfilename** и передаём ей расширения, который будет показан в выпадающем списке «Тип файла»:

```
fname = filedialog.askopenfilename (
              filetypes = [ ("Файлы GIF", "*.gif"),
                            ("Все файлы", "*.*")] )
```

Параметр **filetypes** – это список, составленный из двух кортежей. Каждый кортеж содержит два элемента: текстовое объяснение и расширение имени файлов. Результат работы функции –имя выбранного файла или пустая строка, если пользователь отказался от выбора (нажал на кнопку «Отмена»). Если файл всё-таки выбран (строка не пустая), записываем имя файла в свойство **picture** объекта **TImage**:

```
if fname:
  image.picture = fname
```

При изменении этого свойства файл будет автоматически загружен и выведен на экран (это выполняет класс **TImage**). Обратите внимание, что пустая строка в Python воспринимается как ложное значение. Теперь можно составить всю функцию:

```
def selectFile ( sender ):
  fname = filedialog.askopenfilename(
              filetypes = [ ("Файлы GIF", "*.gif"),
                            ("Все файлы", "*.*")] )
  if fname:
    image.picture = fname
```

Функция-обработчик должна принимает единственный параметр **sender** – ссылку на объект, с которым произошло событие. Но в нашем случае эту ссылка не нужна, и мы её использовать не будем.

Теперь нужно установить обработчик события: сделать так, чтобы функция **selectFile** вызывалась в случае щелчка по кнопке. Свойство, определяющее обработчик события «щелчок мышью», называется **onClick** (от англ. click – щёлкнуть):

```
openBtn.onClick = selectFile
```

Теперь можно запустить программу и проверить загрузку файлов. Остаётся добавить еще один обработчик, который при изменении состояния выключателя **centerCb** переключает режим вывода рисунка (в левом углу «холста» или по центру»). Напишем код такого обработчика:

```
def cbChanged ( sender ):
  image.center = sender.checked
  image.redrawImage()
```

У объекта **TImage** есть логическое свойство **center**, которое должно быть равно **True**, если нужно вывести рисунок по центру и **False**, если рисунок выводится в левом верхнем углу «холста». С другой стороны, у выключателя **TCheckBox** есть логическое свойство **checked**, которое равно **True**, если выключатель включен (флажок отмечен). В первой строке функции **cbChanged** мы установили свойство **center** в соответствии с состоянием выключателя. Затем вызывается метод **redrawImage** (от англ. *redraw image* – перерисовать рисунок), который выводит рисунок в новой позиции. Этот обработчик нужно подключить к событию «изменение состояния выключателя», записав его адрес в свойство **onChange**:

```
centerCb.onChange = cbChanged
```

Если теперь запустить программу, мы должны увидеть правильную работу всех элементов управления. Более того, при изменении размеров окна перерисовка выполняется автоматически (за это также «отвечает» компонент **TImage**).

Отметим следующие важные особенности:

- программа целиком состоит из объектов и основана на идеях ООП;
- использование готовых компонентов скрывает от нас сложность выполняемых операций, поэтому скорость разработки программ значительно повышается.

НОВЫЙ КЛАСС: ВСЁ В ОДНОМ

Доведём объектный подход до логического завершения, собрав все элементы интерфейса в новый класс **TImageViewer** – оконное приложение для просмотра рисунков. При этом основная программа будет состоять всего из двух строчек: создание главного окна и запуск программы:

```
class TImageViewer ( TApplication ):
  ...
app = TImageViewer()
app.Run()
```

Вместо многоточия нужно добавить описание класса: конструктор, который создает и размещает на форме все компоненты, и обработчики событий. Начнём с конструктора:

```
class TImageViewer ( TApplication ):
  def __init__(self):
    TApplication.__init__ ( self, "Просмотр рисунков" )
    self.position = (200, 200)
    self.size = (300, 300)
    self.panel = TPanel(self, relief = "raised",
                              height = 35, bd = 1)
    self.panel.align = "top"
    self.image = TImage ( self, bg = "white" )
    self.image.align = "client"
    self.openBtn = TButton ( self.panel,
                             width = 15, text = "Открыть файл" )
    self.openBtn.position = (5, 5)
    self.openBtn.onClick = self.selectFile
    self.centerCb = TCheckBox ( self.panel, text = "В центре" )
    self.centerCb.position = (115, 5)
    self.centerCb.onChange = self.cbChanged
```

Сначала вызываем конструктор базового класса **TApplication** и настраиваем свойства окна. Вместо имени объекта **app** в предыдущей программе используем **self** – ссылку на текущий объект.

Затем строятся и размещаются компоненты, причём ссылки на них становятся полями объекта (см. «приставку» **self.** перед именами). Иначе мы не сможем обратиться к компонентам в обработчиках событий.

Обработчики событий – функции **selectFile** и **cbChanged** – тоже должны быть членами класса **TImageViewer**, поэтому их первым параметром будет **self**, а вторым – ссылка **sender** на объект, в котором произошло событие:

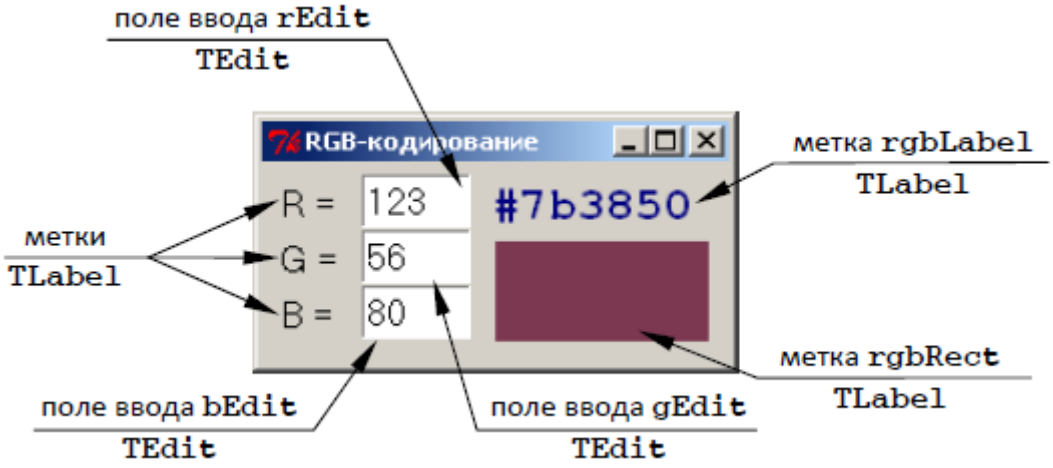
```
class TImageViewer ( TApplication ):
  ... # здесь должен быть конструктор __init__
  def selectFile ( self, sender ):
    fname = filedialog.askopenfilename(
       filetypes = [ ("Файлы GIF", "*.gif"),
                     ("Все файлы", "*.*")] )
    if fname:
      self.image.picture = fname
  def cbChanged ( self, sender ):
    self.image.center = sender.checked
    self.image.redrawImage()
```

Поскольку при создании объекта в конструкторе мы запомнили адреса всех элементов в полях объекта, теперь можно к ним обращаться для изменения свойств.

В итоге получилась программа, полностью построенная на принципах объектно-ориентированного программирования: все данные и методы работы с ними объединены в классе **`TImageViewer`**.

Ввод и вывод данных

Во многих программах нужно, чтобы пользователь вводил текстовую или числовую информацию. Чаще всего для этого применяют поле ввода – компонент **`TEdit`** (наследник класса **`Entry`** библиотеки **`tkinter`**). Для доступа к введённой строке используют его свойство **`text`** (англ. текст). Построим программу для перевода RGB-составляющих цвета в соответствующий шестнадцатеричный код, который используется для задания цвета в языке HTML.



На форме расположены

- три поля ввода (в них пользователь может задать значения красной, зелёной и синей составляющих цвета в модели RGB);
- прямоугольник (компонент **`TLabel`**), цвет которого изменяется согласно введенным значениям;
- несколько меток (компонентов **`TLabel`**)

Метки – это надписи, которые пользователь не может редактировать, однако их содержание можно изменять из программы через свойство **`text`**.

Во время работы программы будут использоваться поля ввода **`rEdit`**, **`gEdit`** и **`bEdit`**, метка **`rgbLabel`**, с помощью которой будет выводиться результат – код цвета, и метка без текста **`rgbRect`** – прямоугольник, закрашенный нужным цветом. В

качестве начальных значений полей ввода можно ввести любые целые числа от 0 до 255 (свойство **text**).

При изменении содержимого одного из трёх полей ввода нужно обработать введённые данные и вывести результат в свойство **text** метки **rgbLabel**, а также изменить цвет фона для метки **rgbRect**. Обработчик события, которое происходит при изменении текста в поле ввода, называется **onChange**. Так как при изменении любого из трех полей нужно выполнить одинаковые действия, для этих компонентов можно установить один и тот же обработчик.

Обработчик события **onChange** для поля ввода может выглядеть так:

```
def onChange ( sender ):
  r = int ( rEdit.text )              # (1)
  g = int ( gEdit.text )              # (2)
  b = int ( bEdit.text )              # (3)
  s = "#{:02x}{:02x}{:02x}".format(r, g, b)    # (4)
  rgbRect.background = s              # (5)
  rgbLabel.text = s                   # (6)
```

Содержимое всех полей ввода преобразуется в числа с помощью функции **int** (строки 1-3). Затем в строке 4 строится символьная строка – шестнадцатеричная запись цвета в HTML-формате. Значения красной, зелёной и синей составляющих (переменные **r**, **g** и **b**) выводятся по формату «**02x**», то есть в шестнадцатеричной записи, состоящей из двух цифр с заполнением пустых позиций нулями. В строке 5 изменяется фон метки-прямоугольника **rgbRect**, а в строке 6 код цвета заносится в метку **rgbLabel**.

В основной программе создаём объект-приложение с главным окном:

```
app = TApplication ( "RGB-кодирование" )
app.size = (210, 90)
app.position = (200, 200)
```

и метки слева от полей ввода:

```
f = ( "MS Sans Serif", 12 )
lblR = TLabel ( app, text = "R = ", font = f )
lblR.position = (5, 5)
lblG = TLabel ( app, text = "G = ", font = f )
lblG.position = (5, 30)
lblB = TLabel ( app, text = "B = ", font = f )
lblB.position = (5, 55)
```

При создании меток мы изменили шрифт с помощью параметра **font**. Значение этого параметра – кортеж **f**, который в данном случае состоит из двух элементов: названия шрифта и его размера в пунктах. Создаём еще две метки для вывода результата:

```
fc = ( "Courier New", 16, "bold" )
rgbLabel = TLabel ( app, text = "#000000", font = fc, fg = "navy" )
rgbLabel.position = (100, 5)
rgbRect = TLabel ( app, text = "", width = 15, height = 3 )
rgbRect.position = (105, 35)
```

Размеры меток – ширина (англ. *width*) и высота (англ. h*eigth*) указываются не в пикселях, а в текстовых единицах, то есть в размерах символа. Шрифт для метки **rgbLabel** задаётся в виде кортежа из трёх элементов: название шрифта, размер и свойство «жирный» (англ. *bold*). Параметр **fg** (от англ. *foreground* – цвет переднего

плана) при вызове конструктора метки определяет цвет символов («*navy*» – тёмно-синий, от англ. *navy* - военно-морской).
Затем добавляем на форму три поля ввода:

```
rEdit = TEdit ( app, font = f, width = 5 )
rEdit.position = (45, 5)
rEdit.text = "123"
gEdit = TEdit ( app, font = f, width = 5 )
gEdit.position = (45, 30)
gEdit.text = "56"
bEdit = TEdit ( app, font = f, width = 5 )
bEdit.text = "80"
bEdit.position = (45, 55)
```

для каждого из них устанавливаем обработчик **onChange**, который вызывается при любом изменении текста в поле ввода:

```
rEdit.onChange = onChange
gEdit.onChange = onChange
bEdit.onChange = onChange
```

и запускаем программу:

```
app.Run()
```

Обратите внимание, что назначение обработчика событий нужно делать тогда, когда все объекты, используемые в обработчике **onChange** (поля ввода **rEdit**, **gEdit**, **bEdit** и метки **rgbLabel** и **rgbRect**) уже созданы.

При желании можно переписать программу в стиле ООП, как это мы сделали в конце предыдущего примера. Это вы уже можете сделать самостоятельно, используя образец.

Обработка ошибок

Если в предыдущей программе пользователь введет не числа, а что-то другое (или пустую строку), программа выдаст сообщение о необработанной ошибке на английском языке, и программа завершит работу. Хорошая программа никогда не должна завершаться аварий, но для этого все ошибки, которые можно предусмотреть, надо обрабатывать.

В современных языках программирования есть так называемый механизм исключений, который позволяет обрабатывать практически все возможные ошибки. Для этого все «опасные» участки кода (на которых может возникнуть ошибка) нужно поместить в блок **try – except**:

```
try:
  # «опасные» команды
except:
  # обработка ошибки
```

Слово **try** по-английски означает «попытаться», **except** — «исключение» (исключительная или ошибочная, непредвиденная ситуация). Программа попадает в блок **except** только тогда, когда между **try** и **except** произошла ошибка. В нашей программе «опасные» команды – это операторы преобразования данных из текста в числа (вызовы функции **int**). В случае ошибки мы выведем вместо кода цвета знак вопроса, а цвет фона метки-прямоугольника **rgbRect** изменим на стандартное значение "**SystemButtonFace**", которое означает «системный цвет кнопки». При этом

прямоугольник становится невидимым, потому что сливается с фоновым цветом окна. Улучшенный обработчик с защитой от неправильного ввода принимает вид:

```
def onChange ( sender ):
  s = "?"
  bkColor = "SystemButtonFace"
  try:
    # получить новый цвет из полей ввода
  except:
    pass
  rgbLabel.text = s
  rgbRect.background = bkColor
```

Здесь переменная **s** обозначает шестнадцатеричный код цвета (в виде символьной строки), а переменная **bkColor** – цвет прямоугольника. Если при попытке получить новый код цвета происходит ошибка, то в этих переменных остаются значения, установленные в первых строках обработчика.

Если значения полей ввода удалось преобразовать в числа, нужно убедиться, что все составляющие цвета не меньше 0 и не больше 255, добавив проверку диапазона **range(256)** для всех значений:

```
def onChange ( sender ):
  s = "?"
  bkColor = "SystemButtonFace"
  try:
    r = int ( rEdit.text )
    g = int ( gEdit.text )
    b = int ( bEdit.text )
    if r in range(256) and \
       g in range(256) and b in range(256):
      s = "#{:02x}{:02x}{:02x}".format(r, g, b)
      bkColor = s
  except:
    pass
  rgbLabel.text = s
  rgbRect.background = bkColor
```