Всплывающие окна / tkinter 11

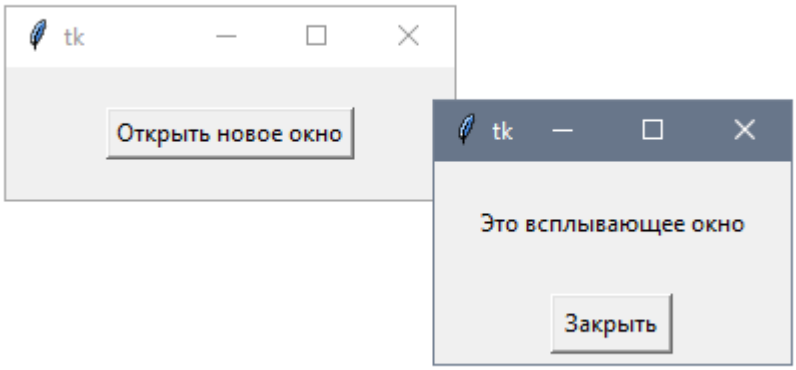
Скачайте код уроков с GitLab: **https://gitlab.com/PythonRu/tkinter-uroki**

Открытие дополнительного окна

Корневой экземпляр `Tk` представляет собой основное окно графического интерфейса. Когда он уничтожается, приложение закрывается, а основной цикл завершается.

Но есть в Tkinter и другой класс, который используется для создания дополнительных окон верхнего уровня. Он называется `Toplevel`. Этот класс можно использовать для отображения любого типа окна: от диалоговых до мастер форм.

Начнем с создания простого окна, которое открывается по нажатию кнопки в основном. Дополнительное окно будет включать кнопку, которая закрывает его и возвращает фокус в основное:



Класс виджета `Toplevel` создает новое окно верхнего уровня, которое выступает в качестве родительского контейнера по аналогии с экземпляром `Tk`. В отличие от

класса Tk можно создавать неограниченное количество окон верхнего уровня:

КОПИРОВАТЬ

```
import tkinter as tk

class About(tk.Toplevel):
    def __init__(self, parent):
        super().__init__(parent)
        self.label = tk.Label(self, text="Это всплывающее окно")
        self.button = tk.Button(self, text="Закрыть", command=self.destroy)

        self.label.pack(padx=20, pady=20)
        self.button.pack(pady=5, ipadx=2, ipady=2)

class App(tk.Tk):
    def __init__(self):
        super().__init__()
        self.btn = tk.Button(self, text="Открыть новое окно",
                             command=self.open_window)
        self.btn.pack(padx=50, pady=20)

    def open_window(self):
        about = About(self)
        about.grab_set()

if __name__ == "__main__":
    app = App()
    app.mainloop()
```

Как работают всплывающие окна

Определяем подкласс `Toplevel`, который будет представлять собой кастомное окно. Его отношение с родительским будет определено в методе `__init__`. Виджеты добавляются в это окно привычным образом:

КОПИРОВАТЬ

```
class Window(tk.Toplevel):
    def __init__(self, parent):
        super().__init__(parent)
```

Окно открывается за счет создания нового экземпляра, но чтобы оно получало все события, нужно вызвать его метод `grab_set`. Благодаря этому пользователи не будут взаимодействовать с основным окном, пока дополнительное не закроется:

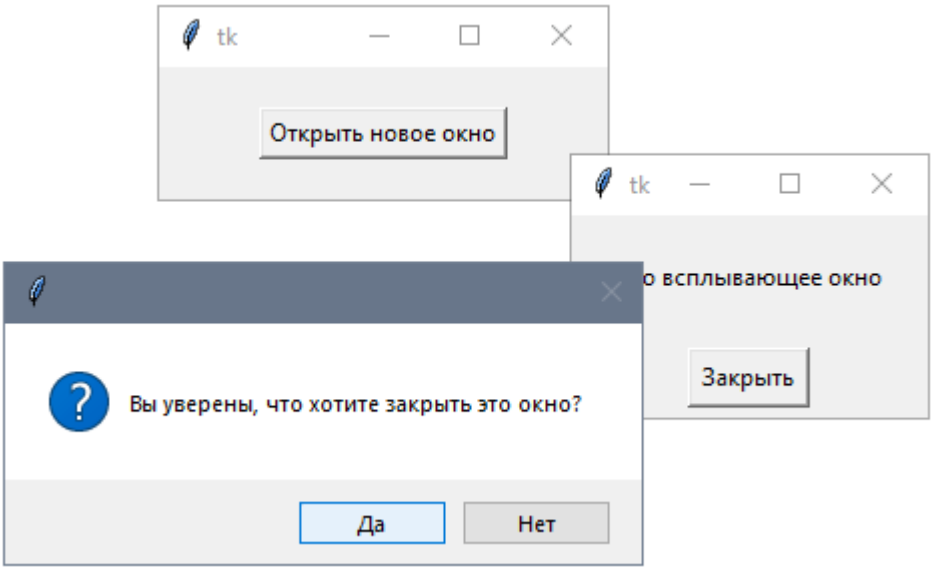
КОПИРОВАТЬ

```
def open_window(self):
    window = Window(self)
    window.grab_set()
```

Обработка закрытия окна

При определенных условиях может потребоваться выполнить определенное действие до закрытия окно верхнего уровня: например, чтобы пользователь не потерял несохраненную работу. Tkinter позволяет перехватывать этот тип событий, что позволяет уничтожать окно только при определенном условии.

Будем заново использовать класс `App` из прошлого примера, но изменим класс `Window`, чтобы он показывал диалоговое окно для подтверждения перед закрытием:



В Tkinter можно определить, когда окно готовится закрыться с помощью функции-обработчика для протокола `WM_DELETE_WINDOW`. Его можно запустить, нажав на иконку «x» в верхней панели в большинстве настольных программ.

КОПИРОВАТЬ

```
import tkinter as tk
import tkinter.messagebox as mb

class Window(tk.Toplevel):
    def __init__(self, parent):
        super().__init__(parent)
        self.protocol("WM_DELETE_WINDOW", self.confirm_delete)
        self.label = tk.Label(self, text="Это всплывающее окно")
        self.button = tk.Button(self, text="Закрыть", command=self.destroy)
        self.label.pack(padx=20, pady=20)
        self.button.pack(pady=5, ipadx=2, ipady=2)

    def confirm_delete(self):
        message = "Вы уверены, что хотите закрыть это окно?"
        if mb.askyesno(message=message, parent=self):
            self.destroy()

class App(tk.Tk):
    def __init__(self):
        super().__init__()
        self.btn = tk.Button(self, text="Открыть новое окно",
                             command=self.open_about)
        self.btn.pack(padx=50, pady=20)
```

```
    def open_about(self):
        window = Window(self)
        window.grab_set()

if __name__ == "__main__":
    app = App()
    app.mainloop()
```


Этот метод-обработчик показывает диалоговое окно для подтверждения удаления окна. В более сложных программах логика обычно включает дополнительную валидацию.

Как работает проверка закрытия окна

Метод `bind()` используется для регистрации обработчиков событий виджетов, а метод `protocol` делает то же самое для протоколов менеджеров окна.

Обработчик `WM_DELETE_WINDOW` вызывается, когда окно верхнего уровня должно уже закрываться, и по умолчанию `Tk` уничтожает окно, для которого оно было получено. Поскольку это поведение перезаписывается с помощью обработчика `confirm_delete`, нужно явно уничтожить окно при подтверждении.

Еще один полезный протокол — `WM_TAKE_FOCUS`. Он вызывается, когда окно получает фокус.

Стоит запомнить, что для сохранения фокуса в дополнительном окне при открытом диалоговом нужно передать ссылку экземпляру верхнего уровня в параметре `parent` диалоговой функции:

```
if mb.askyesno(message=message, parent=self):
    self.destroy()
```

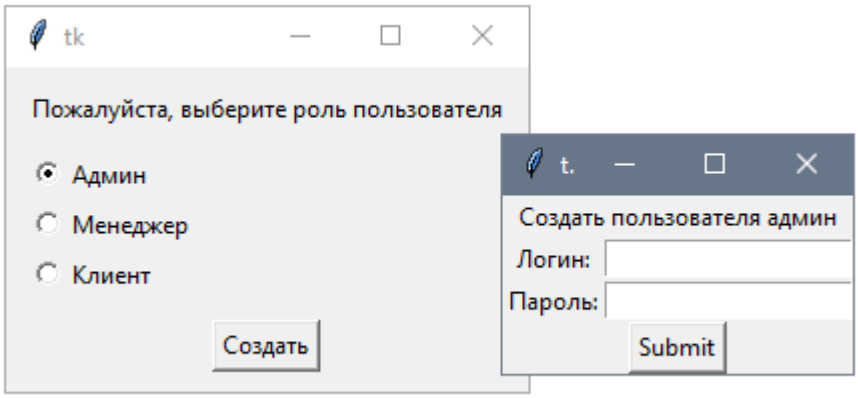
КОПИРОВАТЬ

В противном случае диалоговое окно будет считать родительским корневое, и оно будет возникать над вторым. Это может запутать пользователей, поэтому хорошей практикой считается правильное указание родителя для каждого экземпляра верхнего уровня или диалогового окна.

Передача переменных между окнами

Два разных окна иногда нуждаются в том, чтобы передавать информацию прямо во время работы программы. Ее можно сохранять на диск и читать в нужном окне, но в определенных случаях удобнее обрабатывать ее в памяти и передавать в качестве переменных.

Основное окно будет включать три кнопки-переключателя для выбора типа пользователя, а второе — форму для заполнения его данных:



Для сохранения пользовательских данных создаем `namedtuple` с полями, которые представляют собой экземпляр каждого пользователя. Эта функция из модуля `collections` получает имя типа и последовательность имен полей, а возвращает подкласс кортежа для создания простого объекта с заданными полями:

КОПИРОВАТЬ

```
import tkinter as tk
from collections import namedtuple

User = namedtuple("User", ["username", "password", "user_type"])

class UserForm(tk.Toplevel):
    def __init__(self, parent, user_type):
        super().__init__(parent)
        self.username = tk.StringVar()
        self.password = tk.StringVar()
        self.user_type = user_type

        label = tk.Label(self, text="Создать пользователя " + user_type.lower())
        entry_name = tk.Entry(self, textvariable=self.username)
        entry_pass = tk.Entry(self, textvariable=self.password, show="*")
        btn = tk.Button(self, text="Submit", command=self.destroy)

        label.grid(row=0, columnspan=2)
        tk.Label(self, text="Логин:").grid(row=1, column=0)
        tk.Label(self, text="Пароль:").grid(row=2, column=0)
        entry_name.grid(row=1, column=1)
        entry_pass.grid(row=2, column=1)
        btn.grid(row=3, columnspan=2)
```

```
    def open(self):
        self.grab_set()
        self.wait_window()
        username = self.username.get()
        password = self.password.get()
        return User(username, password, self.user_type)

class App(tk.Tk):
    def __init__(self):
        super().__init__()
        user_types = ("Админ", "Менеджер", "Клиент")
        self.user_type = tk.StringVar()
        self.user_type.set(user_types[0])

        label = tk.Label(self, text="Пожалуйста, выберите роль пользователя")
        radios = [tk.Radiobutton(self, text=t, value=t,
                                 variable=self.user_type) for t in user_types]
        btn = tk.Button(self, text="Создать", command=self.open_window)

        label.pack(padx=10, pady=10)
        for radio in radios:
            radio.pack(padx=10, anchor=tk.W)
        btn.pack(pady=10)

    def open_window(self):
        window = UserForm(self, self.user_type.get())
        user = window.open()
        print(user)

if __name__ == "__main__":
    app = App()
    app.mainloop()
```

Когда поток выполнения возвращается в основное окно, пользовательские данные выводятся в консоль.

Как работает передача данных между окнами

Большая часть кода в этом примере рассматривалась и ранее, а основное отличие — в методе `open()` класса `UserForm` , куда перемещен вызов `grab_set()` . Однако именно

метод `wait_windows()` отвечает за остановку исполнения и гарантирует, что данные не вернутся, пока форма не будет изменена:

КОПИРОВАТЬ

```
def open(self):
    self.grab_set()
    self.wait_window()
    username = self.username.get()
    password = self.password.get()
    return User(username, password, self.user_type)
```

Важно отметить, что `wait_windows()` запускает локальный цикл событий, который завершается после уничтожения окна. Хотя и существует возможность передать виджет, который должен быть удален, этот момент можно пропустить. В таком случае ссылка будет выполнена неявно на экземпляр, который вызвал метод.

Когда экземпляр `UserForm` уничтожается, выполнение метода `open()` продолжается, и он возвращает объект `User` , который теперь может быть использован в классе `App` :

КОПИРОВАТЬ

```
def open_window(self):
    window = UserForm(self, self.user_type.get())
    user = window.open()
    print(user)
```