1.8 Interacting with widget methods

Widget methods

Widgets have **methods** – you've met some of them already. Now we're going to show you a few more of them, and we'll start with two which seem to be very specific. We can even say that the sense of their existence is very closely bound to the unique features of **event programming**.

The methods are named (assuming that `Widget` is an existing widget):

```
Widget.after(time_ms, function)

Widget.after_cancel(id)
```

- `after()` – this method expects two arguments: the first is a **time interval specification** (expressed in milliseconds: 1 s = 1000 ms) and the second **points to an existing function**; successful invocation of the method causes the event manager to change its plans; when the number of milliseconds elapses, the manager **will invoke the function** (only once); note: this the only possible way of controlling the **passage of time** when using an event-driven environment.\\Why? Because you can't just invoke the built-in `sleep()` function within any of your callbacks – it would **freeze** your application for the whole nap time; the `after()` method returns a value which is as specific as the method itself – it's a unique **id** of the planned invocation; is it usable? Yes, it is, e.g., when you are going to delete the previously planned invocation from the manager's calendar, which is done with a method named...
- `after_cancel(id)` – the method cancels the planned invocation identified by the **id** argument.

Seems confusing? Not at all. The example will shed more light on it than telling you a long and winding story.

```
import tkinter as tk


def blink():
    global is_white
    if is_white:
        color = 'black'
    else:
        color = 'white'
    is_white = not is_white
    frame.config(bg=color)
    frame.after(500, blink)


is_white = True
window = tk.Tk()
frame = tk.Frame(window, width=200, height=100, bg='white')
frame.after(500, blink)
frame.pack()
window.mainloop()
```

The code we've written to demonstrate how the `after()` method works is rather simple (yes, absolutely; don't you think so, too?). You can see it in the editor window.

There's a function named `blink()`, which changes the f frame's background color from white to black and back depending on the state of the `is_white` variable. Easy.

Note: there is no **explicit invocation** of the function inside the code. Moreover, it isn't assigned as a callback. The question is – who invokes it?

The event managers do, because:

- we initially encourage it to make the invocation before the frame widget is packed into the main window;
- we continue to encourage it every time the `blink()` function is invoked – this gives the application the ability to blink as long as the application is running.

Try to change the delay time (the first method's argument) and check how the application works then.

The destroy() method

The `destroy()` method is very destructive. It removes the widget completely, not only from your sight, but also from the event manager's memory, as the widget's object is deleted and becomes inaccessible.

```
Widget.destroy()
```

Do you remember? We used this method to close the main window and to stop the whole application. You can use the method in a less devastating manner to get rid of unnecessary widgets while keeping the application alive.

Note: if the widget you want to destroy has **children** (when other widgets are embedded inside it, which can happen with frames) the children will be destroyed, too (this rule works **recursively**).

```
import tkinter as tk


def suicide():
    frame.destroy()


window = tk.Tk()
frame = tk.Frame(window, width=200, height=100, bg='green')
button = tk.Button(frame, text="I'm a frame's child")
button.place(x=10, y=10)
frame.after(5000, suicide)
frame.pack()
window.mainloop()
```

The example prepares a window filled with a **frame** that is the parent of one Button. Note – we've ordered the event manager to activate the `suicide()` function during the 5th second of application life.

The function destroys the frame, but first it destroys all the frame's children and its children's children and ... okay, let's end the story here . It's infinite – telling it will take too much time.

Run the code and check how it works.

As you already know, widgets may or may not have the **focus**. At most one widget can have the focus. When you use a keyboard to interact with the application, you can use the *Tab* and *Shift-Tab* keys to move the focus forward and backward, but the focus can be controlled programmatically, too. There are two methods to help you cope with this issue. Assuming that `Widget` is an existing widget, the methods look as follows:

```
wi.focus_get()
```

```
wi.focus_set()
```

- the `focus_get()` method returns a reference to the **currently focused widget**, or None when no widget owns the focus (note: you can invoke the method from any widget, including the main window)
- the `focus_set()` method **focuses** the widget from the method which was invoked, so you have to choose it carefully.

Look at the code we've provided in the editor. This simple sample application shows how the focus can be **moved** between two buttons and uses the `after()` method to propel the process.

```
import tkinter as tk


def flip_focus():
    if window.focus_get() is button_1:
        button_2.focus_set()
    else:
        button_1.focus_set()
    window.after(1000, flip_focus)


window = tk.Tk()
button_1 = tk.Button(window, text="First")
button_1.pack()
button_2 = tk.Button(window, text="Second")
button_2.pack()
window.after(1000, flip_focus)
window.mainloop()
```

Try to add one or more buttons to the window and change the `jumpthefocus()` function to organize a cyclical focus journey around all the buttons.

We'll say "goodbye" to widgets now, as we're going to place the **observable variables** under our magnifying glass.