Countdown Timer

Łukasz Szóstek (@thels6)

December 6, 2013

1 Introduction

This is a literate program. It means it has been written mainly for a human reader but it also contains all the code making the application. If you are reading a PDF document—it is documentation created from the literate program and if you run the application, you have executed application code extracted from that program.

To explain the code as simply as I could it has been split into a number of "chunks" which we will go through.

`"timer.tcl"` 1≡

⟨ *preamble* 10a ⟩
⟨ *updatingDisplay* 2b, ... ⟩
⟨ *theGUI* 3b, ... ⟩
⟨ *customTimeWidgets* 5b, ... ⟩
⟨ *startingTimer* 7b ⟩
⟨ *pauseButton* 8a ⟩
⟨ *reset* 8b ⟩
⟨ *startButton* 9a ⟩
⟨ *fontWarning* 9b ⟩
⟨ *theCountDown* 2a ⟩
⟨ *runTheApp* 10b ⟩
◇

In order to explain the application better I will explain the functionality in order a bit different than defined above.

2 Fundamental decisions: counting time and displaying it

Let's start from the core of this application: the procedure that counts time. Tcl has an event loop built in, so we just need to schedule a counter procedure to be called every second.

We are going to use a few global variables that will bind the program together. It is possible and a good engineering practice to get rid of them but I'm going to leave them for now for the sake of simplicity.

The `howMany` global variable holds a number of seconds remaining until the end of the countdown period. The `timesUp` global variable is not used at the moment but it is meant to be a trigger when the countdown gets down to zero. Just add a `trace` on it and you will be notified when the timer ends.

So, the procedure counting down is simple: if the number of remaining seconds is greater than 0 we update the `howMany` variable and schedule this procedure to be called again in one second. Otherwise we trigger the `timesUp` variable and do nothing.

⟨ *theCountDown* 2a ⟩ ≡

```
proc timerSeconds {} {
    global timesUp
    global howMany

    if {$howMany > 0} {
        incr howMany -1
        after 1000 "timerSeconds"
    } else {
        #toggle the variable
        set timesUp [expr {[incr timesUp]%2}]
    }
    return
}
```

◇

Fragment referenced in 1.

Just counting seconds wouldn't be helpful if we could not update the screen upon that. We are going to trace the `howMany` variable that the countdown procedure is updating to trigger screen update.

⟨ *updatingDisplay* 2b ⟩ ≡

```
trace add variable howMany write updateDisplay
```

◇

Fragment defined by 2b, 3a.
Fragment referenced in 1.

To make the update easy we are, again, going to use some global variables. The `timeLeft` will keep the number of seconds left in the "MM:SS" format and a separate `hoursLeft` variable will count the number of hours. The split is there to later simplify construction of the GUI: we are going to use Tk's self-updating labels to display information on the user interface.

⟨ *updatingDisplay* 3a ⟩ ≡

```
proc updateDisplay {args} {
    global howMany
    global timeLeft
    global hoursLeft

    set timeLeft [clock format $howMany -format %M:%S]
    set hoursLeft [expr $howMany / 3600]
}
```

◇

Fragment defined by 2b, 3a.
Fragment referenced in 1.

3 GUI

Once we have the fundamental decisions taken, let's build the GUI. We are going to build it from top to bottom and from left to right.

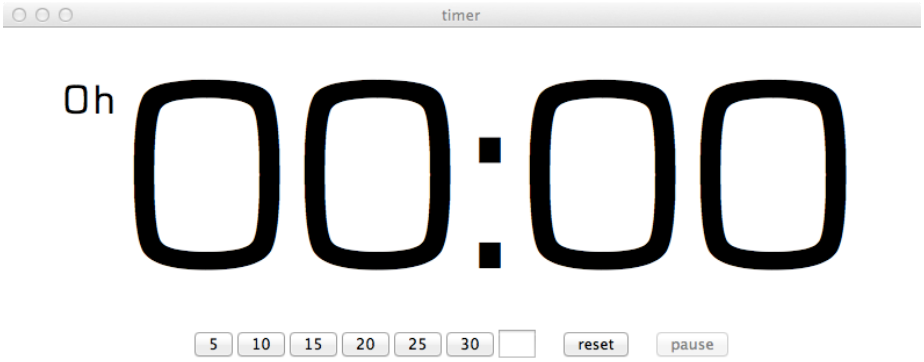

Figure 1: The GUI

The timer works fine with the default OSX font—I haven't checked Windows. The only requirement is that all the number should be of the same width. Otherwise the width of the window will be changing[1]. We will cover choosing a specific font later for now we will choose font sizes.

⟨ *theGUI* 3b ⟩ ≡

```
proc makeGUI {} {
    wm title . {timer}

    font create FCounter -size 240 -family $::fontFamily
    font create FCounterSmall -size 36 -family $::fontFamily
}
```

◇

Fragment defined by 3b, 4ab, 5a.
Fragment referenced in 1.

[1] If you would like to use a font in which all the numbers are not of the same width, you may want to explicitely fix the width of the window.

The top part of the application window will show the time inside a `.f_time` frame. As we have already said, we are going to place there labels bound to global variables defined in Section 2.

⟨ *theGUI* 4a ⟩ ≡

```
pack [frame .f_time -padx 50] -side top -expand 1
pack [label .f_time.l_timeLeftHours -font FCounterSmall\
      -padx 0 -textvariable hoursLeft\
     ] -side left -anchor ne -ipady 40
pack [label .f_time.l_timeLeftHoursMark -font FCounterSmall\
      -padx 0 -text h\
     ] -side left -anchor ne -ipady 40
pack [label .f_time.l_timeLeft -font FCounter\
      -textvariable timeLeft\
     ] -side left
```

◇

Fragment defined by 3b, 4ab, 5a.
Fragment referenced in 1.

At the bottom of the window we will put a frame that holds all the buttons controlling the application.

⟨ *theGUI* 4b ⟩ ≡

```
pack [frame .f_buttons] -side bottom -anchor s
```

◇

Fragment defined by 3b, 4ab, 5a.
Fragment referenced in 1.

If you want different predefined buttons, the following `foreach` is the place to change it:

⟨ *theGUI* 5a ⟩ ≡

```
    foreach time {5 10 15 20 25 30} {
        pack [button .f_buttons.b_$time -text $time\
              -command "startTimer $time"\
             ] -side left
    }
    makeCustomTimeWidgets
    pack [button .f_buttons.b_stop -text reset\
          -command resetTimer\
         ] -padx 20 -side left
    pack [button .f_buttons.b_pause -text pause\
          -command pauseTimer -state disabled\
         ] -side left

    update idletasks
}
```

◇

Fragment defined by 3b, 4ab, 5a.
Fragment referenced in 1.

We have added a few buttons triggering predefined countdowns but we also want the user to be able to run the timer for any period so we need an entry to input custom time and some logic to take care of that input.

As you can see in Figure 1 when the application starts only the entry is displayed. We will display the "start" button triggering the custom countdown once the user has entered correct data, so below we are going to create this button but will not pack it. The entry for the custom time will be validated after each key press inside it. We will also bind pressing the `<Return>` key to starting the custom countdown so it all can be done from the keyboard.

⟨ *customTimeWidgets* 5b ⟩ ≡

```
proc makeCustomTimeWidgets {} {
    pack [frame .f_buttons.f_customTime] -side left
    pack [entry .f_buttons.f_customTime.e_customTime\
          -validate key\
          -validatecommand "validateCustomTime %S %P" -width 3\
         ] -side left
    bind .f_buttons.f_customTime.e_customTime <Return> startCustomTimer
    button .f_buttons.f_customTime.b_start\
           -text start -command startCustomTimer
}
```

◇

Fragment defined by 5b, 6ab, 7a.
Fragment referenced in 1.

The procedure validating the custom time the user has entered is called after each key-press inside the custom time entry. The parameters are the key that has been pressed and the whole string entered until now.

There is a number of things this procedure is checking. First, if the entry is empty we are going to hide the “start” button.

⟨ *customTimeWidgets* 6a ⟩ ≡

```
proc validateCustomTime {char value} {
    if {$value == {}} {
        set value 0
        hideStartButton
        return true
    }
◇
```

Fragment defined by 5b, 6ab, 7a.
Fragment referenced in 1.

Otherwise and if the entered character is a digit we are going to display the “start” button and adjust the length of the entry so the whole number is visible but not smaller than 3 characters. Finally, if the entered character was not a digit we are going to return `false` which will discard this character.

⟨ *customTimeWidgets* 6b ⟩ ≡

```
    if [string is digit $char] {
        showStartButton
        set timeLen [string length $value]
        if {$timeLen > 3} {
            .f_buttons.f_customTime.e_customTime configure -width [expr {$timeLen + 1}]
        } else {
            .f_buttons.f_customTime.e_customTime configure -width 3
        }
        return true
    } else {
        return false
    }
}
◇
```

Fragment defined by 5b, 6ab, 7a.
Fragment referenced in 1.

The procedure of starting the custom timer is a little different from the procedure starting the predefined countdowns—it needs to get the time from the custom time entry. We are not going to check if this value is correct—that has been already taken care of by the validation procedure.

⟨ *customTimeWidgets* 7a ⟩ ≡

```
proc startCustomTimer {} {
    focus .
    set customTime [.f_buttons.f_customTime.e_customTime get]
    startTimer $customTime
}
```

◇

Fragment defined by 5b, 6ab, 7a.
Fragment referenced in 1.

3.1 Functionality for the buttons

3.1.1 Starting the timer

Starting the timer uses the `timerSeconds` procedure we have defined in Section 2. We also cancel any countdown that may be in progress, convert the number of minutes the countdown is going to take to the number of seconds, and enable the "pause" button. We will also add a binding to the `space` key that will allow to pause the timer.

⟨ *startingTimer* 7b ⟩ ≡

```
proc startTimer {time} {
    after cancel timerSeconds
    set ::howMany [expr $time * 60]
    timerSeconds
    enablePauseButton
    resetPauseButton
    bind . <space> {pauseTimer}
    focus .
}
```

◇

Fragment referenced in 1.

3.1.2 Pausing the timer

The functionality of the "pause" button depends on the state of the timer. If the countdown is running pressing it will pause it; otherwise, if the countdown is paused, it will enable resuming it.

The `focus .` command in the `resumeTimer` is there because of the custom time entry field. It is the only widget capable of showing focus highlight and, for esthetical reasons, we want to remove this focus when the timer is running.

⟨ *pauseButton* 8a ⟩ ≡

```
proc pauseTimer {} {
    after cancel timerSeconds
    .f_buttons.b_pause configure -text resume -command resumeTimer
    bind . <space> {resumeTimer}
}

proc resumeTimer {} {
    timerSeconds
    .f_buttons.b_pause configure -text pause -command pauseTimer
    bind . <space> {pauseTimer}
    focus .
}

proc resetPauseButton {} {
    if [string equal [.f_buttons.b_pause cget -text] resume] {
        .f_buttons.b_pause configure -command pauseTimer -text pause
    }
}

proc enablePauseButton {} {
    .f_buttons.b_pause configure -state normal
}

proc disablePauseButton {} {
    .f_buttons.b_pause configure -state disabled
}
```

◇

Fragment referenced in 1.

3.1.3 Reseting the timer

Reseting the timer is straightforward: we try to cancel a running countdown, reset the countdown period and bring the pause button to the initial state. We will also reomove the binding of the `space` key since pausing or resuming the timer makes no sense in this situation.

⟨ *reset* 8b ⟩ ≡

```
proc resetTimer {} {
    after cancel timerSeconds
    set ::howMany 0
    resetPauseButton
    disablePauseButton
    bind . <space> {}
    focus .
}
```

◇

Fragment referenced in 1.

3.2 The “start” button

The start button that triggers custom timeout is not going to be visible all the time. The following two procedures make it show or disappear.

⟨ *startButton* 9a ⟩ ≡

```
proc showStartButton {args} {
    pack .f_buttons.f_customTime.b_start -side left
}

proc hideStartButton {args} {
    pack forget .f_buttons.f_customTime.b_start
}
```

◇

Fragment referenced in 1.

3.3 Handling the missing font

I have spent some time choosing a nice looking font so I wanted people who just use this appliaction—as opposed to those reading this documentation—to be able to have the same experience. Therefore, if this specific font is missing we are going to show a pop-up explaining where it can be found.

⟨ *fontWarning* 9b ⟩ ≡

```
proc checkIfMissingFont {} {
    if ![regexp -- $::fontFamily [font actual FCounter]] {
        tk_messageBox -title {missing font} \
            -message "The look of this application has been optimized\
                for the '$::fontFamily' font. You can download it for\
                free from:\nhttp://www.fontpalace.com"
    }
}
```

◇

Fragment referenced in 1.

4 Initializing variables

Finally, we can set the initial values of our global variables controlling the countdown and select the font to display time.

⟨ *preamble* 10a ⟩ ≡

```
package require Tk

set howMany 0
set timeLeft 0
set hoursLeft 0

set fontFamily Eurostile
```
◇

Fragment referenced in 1.

5 Running the application

Now we are ready to run...

⟨ *runTheApp* 10b ⟩ ≡

```
makeGUI
updateDisplay
checkIfMissingFont
```
◇

Fragment referenced in 1.

6 Bugs and requests

1. Entering custom time greater than 7 minutes with a number starting with `0` causes an error. So, '`007`' works fine but '`008`' does not.