1. Some GUI programs need to perform background activities periodically. For example,

- a blinky widget where we might want to schedule a future event at a regular time interval, or
- a long-running operation (e.g., reading a large file or some method with time.sleep()) which we don't want to have block other GUI activity

If the event loop could be forced to update periodically, then the GUI could remain responsive. Tkinter comes with the tools for both scheduling delayed actions and forcing screen updates:

| Event-Driven Methods | Description |
|---|---|
| `id = widget.after(msecs, function, *args)` | Schedules a function callback after a specified number of milliseconds in the future. This `.after` form does NOT pause the GUI program and returns an id that can be passed to the `.after_cancel` function to cancel the callback event. |
| `widget.after_cancel(id)` | Cancels a pending `after` event before it occurs. |
| `widget.after(msecs)` | Pauses the program for a specified number of milliseconds. |
| `widget.update()` | Forces Tkinter to process all pending events including widget updates and redraws. Long-running callback handlers will cause the screen to appear to freeze unless they periodically perform `.update` functions. |

The following programs use different techiques to animate the simple drawing program from lecture and lab. Assigns a tag to each shape ('circle' or 'top_left_corner') on the canvas. When right-mouse button is pressed, the shapes with tags matching the currently selected menu shape move toward the mouse position.

- CanvasTest_tags.py - When the right-mouse button is pressed,
  1. builds idList for canvas items with the correct tag: idList = self._myCanvas.find_withtag(tag)
  2. for i in xrange(5):

```
for shapeId in idList:
    determine step toward cursor in xDiff and yDiff
    self._myCanvas.move(shapeId, xDiff, yDiff)
self._myCanvas.update()
time.sleep(0.1)
```

- CanvasTest_tags_after.py - uses the `.after` method to schedule a future mo
  1. builds idList for canvas items with the correct tag: idList = self._myCanvas.find_withtag(tag)
  2. for i in xrange(5):

```
for shapeId in idList:
    determine step toward cursor in xDiff and yDiff
    self._myCanvas.after(100*i, self._myCanvas.move,shapeId, xDiff, yDiff)
```

  NOTE: This should be more responsive than the time.sleep() approach which freezes the GUI

- CanvasTest_tags_thread.py - uses a new thread which sleeps without freezing the GUI

```
def _moveShape(self, tag, cursor_x, cursor_y):
    thread.start_new_thread(self._moveThem, (tag, cursor_x, cursor_y))
```

```
def _moveThem(self, tag, cursor_x, cursor_y):
```

  1. builds idList for canvas items with the correct tag: idList = self._myCanvas.find_withtag(tag)
  2. for i in xrange(5):

```
for shapeId in idList:
    determine step toward cursor in xDiff and yDiff
    self._myCanvas.move(shapeId, xDiff, yDiff)
self._myCanvas.update()
time.sleep(0.1)
```

2. A Python program utilizes a single core unless the programmer explicitly uses *multithreading*. Fortuately, the `threading` module has `Thread` class which we can inherit as the starting point for our threads. Each `Thread` object might be scheduled by the PVM to run on separate processor cores. If we run the following program OUTSIDE of IDLE by double-click on the file `incrementingThreads.py`, it increments a single variable a specified number of times using a specified number of threads. Thread methods are:

| `Thread` Method | Description |
|---|---|
| `.__init__(name = None)` | Constructs a thread object with an optional name |
| `.getName()` | Returns the thread's name |
| `.setName(newName)` | Sets the thread's name to `newName` |
| `.run()` | Executed when the thread acquires the CPU. |
| `.start()` | Makes the new thread ready. Raises an exception if run more than once. |
| `.isAlive()` | Returns True if the thread is alive or False otherwise |

```
""" File:  incrementingThreads.py   Illustrates concurrency with multiple threads."""
import random
from threading import Thread
from time import clock

class IncrementingThread(Thread):
    """Represents an incrementing thread"""

    def __init__(self, number, counterListRef, increment):
        """Create a thread with the given name and
        increments the counter by incrementAmount"""
        Thread.__init__(self, name = "Thread "+ str(number))
        self._counterList = counterListRef
        self._incrementAmount = increment

    def run(self):
        """Increments the counter by the incrementAmount by a loop."""
        for count in xrange(self._incrementAmount):
            self._counterList[0] += 1
        print self.getName(), "is done."

def main():
    """Create the user's number of threads.  Then start the threads."""
    maxThreads = input("Enter maximum number of threads: ")
    targetAmount = input("Enter number of times all threads should increment: ")
    for numThreads in xrange(1,maxThreads+1):
        start = clock()
        incrementAmount = targetAmount/numThreads
        incrementAmountLastThread = targetAmount - (numThreads-1)*(incrementAmount)

        threadList = []
        counter=[0]
        for count in xrange(numThreads-1):
            threadList.append(IncrementingThread(count+1, counter, incrementAmount))
        threadList.append(IncrementingThread(numThreads,counter,incrementAmountLastThread))
        for thread in threadList:
            thread.start()
        allThreadsDone = False
        while not allThreadsDone:
            allThreadsDone = True
            for thread in threadList:
                if thread.isAlive():
                    allThreadsDone = False
        end = clock()
        runTime = end - start
        print "Counter should be", targetAmount,"but the counter =",counter, "with a",
        print "time of %.6f sec." % runTime

main()
raw_input("Hit <Enter> to quit")
```

Consider the following screen capture with a maximum of 5 threads and an increment amount of 10,000,000.

```
Enter maximum number of threads: 5
Enter number of times all threads should increment: 10000000
Thread 1 is done.
Counter should be 10000000 but the counter = [10000000] with a time of 18.204649 sec.
Thread 1 is done.
Thread 2 is done.
Counter should be 10000000 but the counter = [7248169] with a time of 14.204239 sec.
Thread 1 is done.
Thread 3 is done.
Thread 2 is done.
Counter should be 10000000 but the counter = [5442444] with a time of 12.019376 sec.
Thread 4 is done.
Thread 2 is done.
Thread 1 is done.
Thread 3 is done.
Counter should be 10000000 but the counter = [4281350] with a time of 11.449488 sec.
Thread 1 is done.
Thread 4 is done.
Thread 2 is done.
Thread 3 is done.
Thread 5 is done.
Counter should be 10000000 but the counter = [4032293] with a time of 10.473196 sec.
Hit <Enter> to quit
```

a) Why do the threads sometimes complete out-of-order?

b) Why is the counter only correct with a single thread?