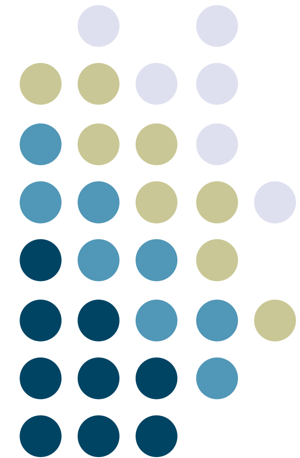
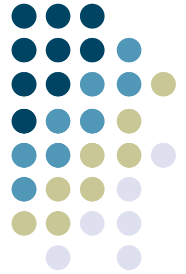


Asynchronous Programming Under the Hood

Georgia Tech

Week 4

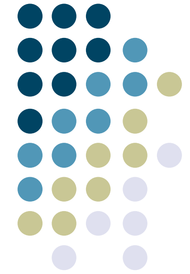
How to Implement Event Dispatchers



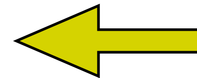
- Recall:
  - Record registered callbacks in a data structure (easy)
  - Wait for an event to happen (easy)
  - Call the callbacks when it happens (easy)
- What's the problem?

Example:

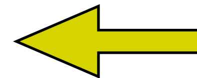
Building an Event Dispatcher to Handle Timer Callbacks



```
import time
import javax.swing as swing

class Timer:
    def __init__(self):
        self.callbacks = []

    def registerCallback(self, callback):
        self.callbacks.append(callback)

    def waitForEvent(self):
        while 1:
            time.sleep(5)
            for cb in self.callbacks:
                cb()


def myCallback():
    now = time.localtime(time.time())
    print "The time is now " + str(now[3]) + ":" + str(now[4]) + ":" + str(now[5])

if __name__ == "__main__":
    disp = Timer()
    disp.registerCallback(myCallback)
    disp.waitForEvent()
    swing.JFrame(”I’m running!").show()
```



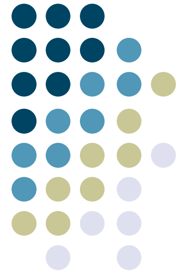
Record callback functions in a list



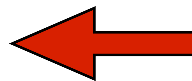
In a loop, sleep for 5 seconds, wake up, then fire all of the callbacks

Example:

Building an Event Dispatcher to Handle Timer Callbacks



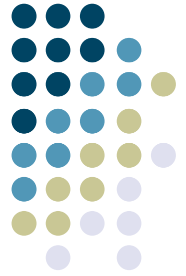
```
import time
import javax.swing as swing

class Timer:
	def __init__(self):
		self.callbacks = []

	def registerCallback(self, callback):
		self.callbacks.append(callback)

	def waitForEvent(self):
		while 1:
			time.sleep(5)
			for cb in self.callbacks:
				cb()


def myCallback():
	now = time.localtime(time.time())
	print "The time is now " + str(now[3]) + ":" + str(now[4]) + ":" + str(now[5])

if __name__ == "__main__":
	disp = Timer()
	disp.registerCallback(myCallback)
	disp.waitForEvent()
	swing.JFrame(”I’m running”).show()
```



The Problem: the waitForEvent() method has to *block* waiting for the timer to expire.

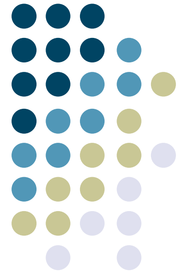
The method never returns control back to the main program.

Building Event Dispatchers



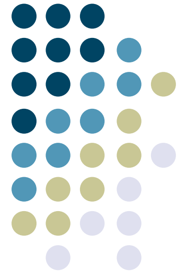
- There are actually two problems with the previous code
    1. Any code that calls waitForEvent() hangs, because the waitForEvent() method *blocks* indefinitely
    2. The way waitForEvent() is implemented, it can only wait for *one* kind of thing (time.sleep()). While the method is blocked waiting on the timer to expire, it wouldn't be able to block waiting for other kinds of events (mouse, etc.)

First Solution: Multi-Way Polling



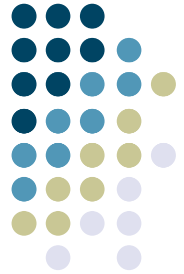
- A technique to let you block waiting for *multiple* sources of activity at the same time
- In Unix: select(), poll()
- Depends on operating system-level support
  - Only recently appeared in Java
  - Not available in current version of Jython
- So we won't talk about it any more!

Second Solution: Threads

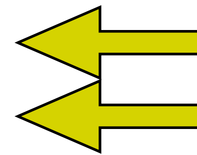


- What you'd like to do is have a way for the main program to keep running while the event dispatcher does its own thing
- *Threads*
  - Separate flow of execution in your program
  - Has its own "position" in the program
- Using threads to implement event dispatchers
  - Your past programs have been "singly threaded": one main thread
  - "Multi threaded": one thread can block waiting on an event to occur without affecting the main thread
  - Need to wait on more than one thing? Use another thread!
- This is basically how the Swing event dispatcher works

Example: A Threaded Event Dispatcher

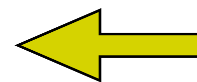


```
import time
import thread
import threading
import javax.swing as swing

class Timer:
	def __init__(self):
		self.callbacks = []
		self.thread = threading.Thread(target=self.waitForEvent)
		self.thread.start()

	def registerCallback(self, callback):
		self.callbacks.append(callback)

	def waitForEvent(self):
		while 1:
			time.sleep(5)
			for cb in self.callbacks:
				cb()


def myCallback():
	now = time.localtime(time.time())
	print "The time is now " + str(now[3]) + ":" + str(now[4]) + ":" + str(now[5])

if __name__ == "__main__":
	disp = Timer()
	disp.registerCallback(myCallback)
	swing.JFrame("hello").show()
```



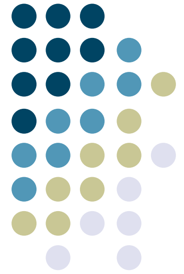
Make a new thread to execute code

Start it running. Make it execute the waitForEvent() method forever.

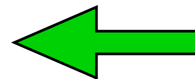


The second thread starts when the Timer is created. No need to call waitForEvents() here.

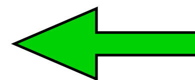
Example: A Threaded Event Dispatcher



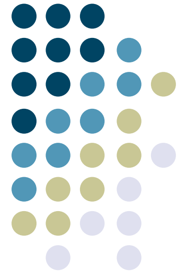
```
import time
import thread
import threading
import javax.swing as swing

class Timer:
	def __init__(self):
		self.callbacks = []
		self.thread = threading.Thread(target=self.waitForEvent)
		self.thread.start()

	def registerCallback(self, callback):
		self.callbacks.append(callback)

	def waitForEvent(self):
		while 1:
			time.sleep(5)
			for cb in self.callbacks:
				cb()


def myCallback():
	now = time.localtime(time.time())
	print "The time is now " + str(now[3]) + ":" + str(now[4]) + ":" + str(now[5])

if __name__ == "__main__":
	disp = Timer()
	disp.registerCallback(myCallback)
	swing.JFrame("hello").show()
```



Now the **timer thread** is executing here...



While the **main thread** executes here!



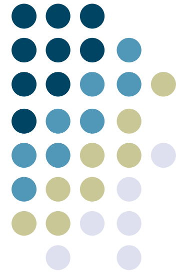
A Word of Caution...

- Be careful about the data that threads modify!
- You want to ensure that two threads can never modify the same data at the same time
- An example from the real world (from Lorenzo Alvisi):

Jack	Jill
Look in fridge, out of milk	
Leave for store	
Arrive at store	Look in fridge, out of milk
Buy milk	Leave for store
Arrive home, put milk away	Arrive at store
	Buy milk
	Arrive home, put milk away
	Oh no!

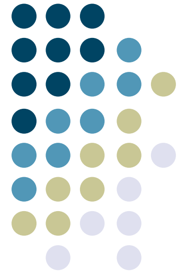
- “Milk” and “Fridge” are the shared data structures in this example

Solution: *Locks*



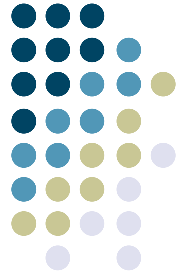
- Scary CS term:
  - Locks provide a way to *synchronize* threads
  - Read: they make sure only one thread at a time is running in code that mucks with data that is used by multiple threads
- Create a new *lock object* you'll use to protect a region of code that shouldn't be mucked with by multiple threads at the same time:
  - self.lock = threading.Lock()
- *Acquire* the lock before reading or writing data that might be accessed by another thread:
  - self.lock.acquire()
- *Release* the lock when you're done:
  - self.lock.release()

Example: a Counter Class

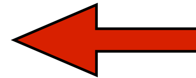


```
import threading, time, random

class Counter:
	def __init__(self):
		self.count = 0;

	def increment(self):
		self.count = self.count + 1
		return self.count

counter = Counter()

class Worker:
	def __init__(self, name):
		self.thread = threading.Thread(target=self.run)
		self.thread.start()
		self.name = name

	def run(self):
		for i in range(10):
			value = counter.increment()
			time.sleep(random.randint(10, 100) / 1000.0)
			print self.thread.getName(), "finished", value

	for i in range(10):
		w = Worker(i)
```

Example: a Counter Class



```
import threading, time, random

class Counter:
    def __init__(self):
        self.count = 0;

    def increment(self):
        self.count = self.count + 1
        return self.count

counter = Counter()

class Worker:
    def __init__(self, name):
        self.thread = threading.Thread(target=self.run)
        self.thread.start()
        self.name = name

    def run(self):
        for i in range(10):
            value = counter.increment()
            time.sleep(random.randint(10, 100) / 1000.0)
            print self.thread.getName(), "finished", value

    for i in range(10):
        w = Worker(i)
```

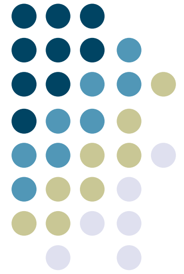


The Problem: Multiple workers may try to run this line of code at the same time.

Worker 1 looks up the value of self.count and adds 1 to it.

Worker 2 does the same thing. Gets the **same value** that Worker 1 sees.

Both then assign to self.count. Effective result is that one increment has been lost.

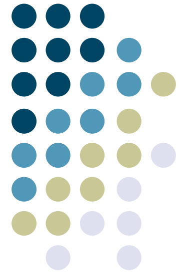


Example: a Thread-Safe Counter

```
import threading, time, random

class Counter:
    def __init__(self):
        self.lock = threading.Lock()
        self.count = 0;

    def increment(self):
        self.lock.acquire()
        value = self.count = self.count + 1
        self.lock.release()
        return value

counter = Counter()

class Worker:
    def __init__(self, name):
        self.thread = threading.Thread(target=self.run)
        self.thread.start()
        self.name = name

    def run(self):
        for i in range(10):
            value = counter.increment()
            time.sleep(random.randint(10, 100) / 1000.0)
            print self.thread.getName(), "finished", value

    for i in range(10):
        w = Worker(i)
```

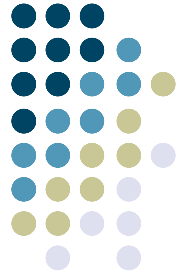
This is the *critical section* of code. Protect it with a lock!

Another Word of Caution...



- You have to synchronize *just* enough code to make it safe
  - If you don't synchronize enough, you'll get hard-to-track errors
  - If you synchronize too much, you do away with the advantage of threads in the first place (only one thing's running at a time)
  - Good practice: use different locks to protect different resources
    - Gives maximum *concurrency*
- Worse case: **deadlock**
  - You can use locks in a way that prevents *any* code from running!
  - Happens when you are using more than one lock:
    - Thread 1 holds Lock A, and is trying to acquire Lock B
    - Thread 2 holds Lock B, and is trying to acquire Lock A
    - Neither can progress

Georgia Tech



Why Learn About Threads?

- If *everything* you do has an event-based programming model, you probably don't need to know about threads
  - But not everything has this model...
- Without events, you'll often have to write code that *blocks* waiting on something to happen
  - Put it in a thread, and keep the rest of your program going
  - Can "wrap" this in an event dispatcher to make it look like any other event source (like the Timer class)
- Examples of things that might block:
  - Network I/O: Reading from the network is even slower. Plus, the other guy might never respond.
  - Waiting for some time to pass. See the Timer class before.
- Threads are necessary for things like this!