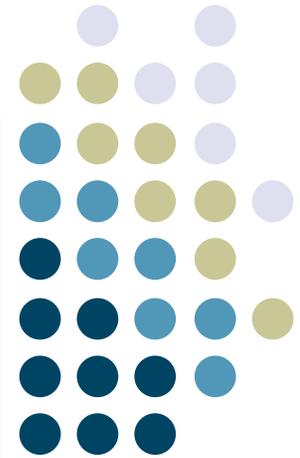


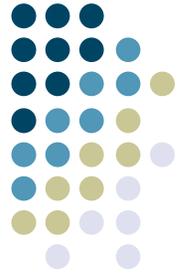
Asynchronous Programming



**Georgia
Tech**



Week 3

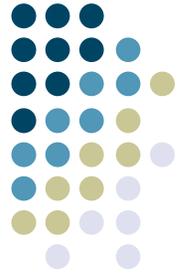


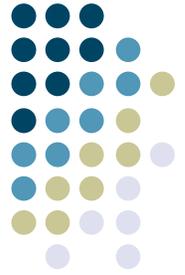
Turn-in Instructions

- A “main” file, called `gui.py`
 - See last week’s slides for how to make it “main”
 - I’ll run it from the command line
- Put in a ZIP file, along with any additional needed files
- Name the ZIP file *your_last_name.zip*
- Send to me in email, along with details on:
 - Anything special I need to do to run it
 - What platform you developed/tested it on
 - Anything else you think I should know
- **PROGRAM DUE 11:59 TUESDAY 9/19/05**

Asynchronous Programming

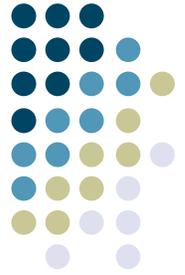
Georgia
Tech





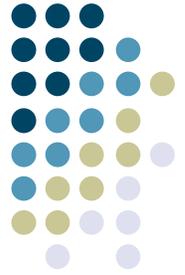
Asynchronous Programming

- Probably **the** most used idiom for interactive systems
- Why? *Interactions with the real world*
 - Must be prepared to respond to events external to your program
 - You don't know when these might occur
 - They may come from multiple sources (a user, remote users, sensors, hardware devices)
- Also, the single biggest mind-shift away from doing simple “straight line” programs
- A few canonical examples:
 - GUIs (responsive to mice, keyboard)
 - Systems that interact with hardware (*interrupts*)
 - Collaborative tools (multiple users, each doing their own thing)



Asynchrony and Modularity

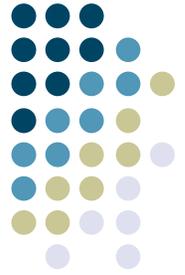
- First-time programmers:
 - Try to do everything “in line” in one flow of control
 - Works only for trivial problems
 - How would you do an “in line” program that needs to respond to multiple event sources?
 - *N.B.: It’s actually possible. In fact, it’s one of the ways that asynchronous programming works “under the hood.” We’ll talk about it later in the semester.*
- Asynchronous programming requires that you break your program down into pieces that are invoked independently whenever any external event happens
- *Modularity*



Modularity is a Good Thing

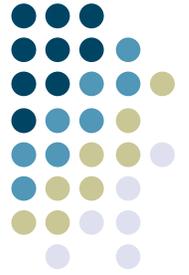
- Fortunately, modularity is a good goal *anyway*
 - Break apart code into more manageable chunks (abstraction)
 - Keep the entanglements between chunks as simple as possible (encapsulation)
 - Corollary: keep as few things global as possible
 - Treat each chunk as a “black box” that does a simple thing, and does it well (information hiding)
- Object-oriented programming is modularity on steroids (an oversimplification)
- Modularity is important when even *one* person is working on it
 - Easier to conceptualize the entire system; chunk behavior into building blocks, etc.

You can't make complexity go away completely, but you can learn techniques to manage it!



Thinking Asynchronously

- Asynchronous: things can happen at arbitrary times
- Your program will probably have two types of code in it:
 - Set-up code, that gets the initial windows on the screen, does initialization, etc.
 - A collection of program chunks that respond to particular types of events that occur
- Some terminology:
 - An *event* is some external occurrence
 - The asynchronously-callable program chunks are *event handlers*
 - An *event dispatcher* is the thing that calls your event handlers; it is typically provided by the system (language, library, OS, ...)
- Your set-up code will *install* your various event handlers, so that the event dispatcher will know which ones to call
- Much of your program's logic will reside in the event handlers!

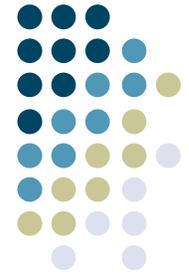


Common Idiom: GUI Callbacks

- In many GUI systems, event handlers are called *callbacks*
 - These are just functions that will be invoked when an event occurs
 - Typically, they take a predefined set of arguments (what event happened, etc.)
 - They are parts of your program that *get called back* when something happens
- How you associate your specific callback with a particular type of event depends on the particulars of the dispatch system

Example: GUI Callbacks in Jython with Swing

Georgia
Tech



```
import javax.swing as swing
```

```
def callback(event):
```

```
    print "Button was pressed:", event
```

```
window = swing.JFrame("CS6452")
```

```
button = swing.JButton("Press Me!")
```

```
button.actionPerformed=callback
```

```
window.contentPane.add(button)
```

```
window.pack()
```

```
window.show()
```

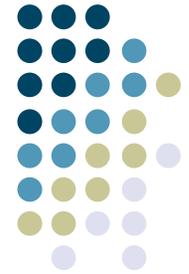


Results:

```
Button was pressed: java.awt.event.ActionEvent  
[ACTION_PERFORMED,cmd=Press  
Me!,when=72985371,modifiers=Button1] on  
javax.swing.JButton[,  
0,0,87x29,layout=javax.swing.OverlayLayout,alignmentX  
=0.0,alignmentY=0.5,border=apple.laf.AquaButtonBorde  
r@eb1670,flags=296,maximumSize=,minimumSize=,pre  
ferredSize=,defaultIcon=,disabledIcon=,disabledSelecte  
dIcon=,margin=javax.swing.plaf.InsetsUIResource  
[top=3,left=14,bottom=3,right=14],paintBorder=true,paint  
Focus=true,pressedIcon=,rolloverEnabled=false,rolloverI  
con=,rolloverSelectedIcon=,selectedIcon=,text=Press  
Me!,defaultCapable=true]
```

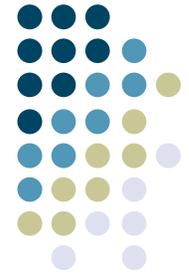
The Details of Event-Based Programming in Swing

Georgia
Tech



- In Swing, events are generated based on user input
 - Mouse clicks, movement, release
 - Key presses, releases
 - Combinations of all of the above
- Each widget gets to define what constitutes an event for it, and how callbacks will be associated with it
 - `button.actionPerformed`
 - `list.valueChanged`
- Any given widget may allow multiple kinds of callbacks to be associated with it
 - `panel.mousePressed`
 - `panel.mouseReleased`
 - `panel.mouseClicked`

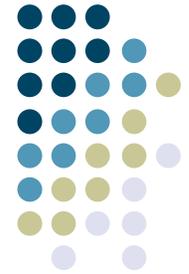
The Details of Event-Based Programming in Swing (cont'd)



- Event dispatcher calls your code when the appropriate combination of user inputs occurs
- Passes an *event* argument to your code
- Specific details contained in the event depend on type of callback:
 - `button.actionPerformed` → `ActionEvent`
 - source: the widget that generated the event
 - timestamp: when the event occurred
 - modifiers: which keys were held down when the event occurred
 - `list.valueChanged` → `ListSelectionEvent`
 - `firstIndex`: first index of changed item
 - `lastIndex`: last index of changed item
 - To get specific details of any given event type, look at the Java documentation (<http://java.sun.com/j2se/1.5.0/docs/api/>) or ask me or the TA

The Details of Event-Based Programming in Swing (cont'd)

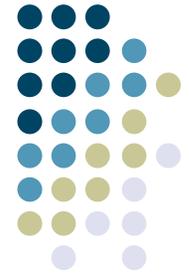
Georgia
Tech



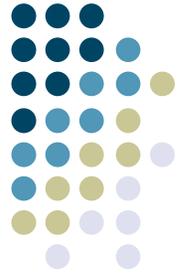
- You can call your callbacks yourself
 - They're just normal functions
 - Simulate what happens when user input occurs
- Make sure you return quickly from your event handlers!
 - The program is waiting until you finish so that it can continue running
 - Common signs of a non-returning callback:
 - Program appears to freeze
 - Program window doesn't redraw
 - Buttons become inactive

O-O and Asynchronous Programming

Georgia
Tech



- Simple callbacks are a perfectly acceptable idiom; they're the “baseline” of asynchronous programming
- If you do much callback programming, though, you begin to notice some common patterns:
 - Often need to share some data across several related callbacks
 - Often need to keep track of what happened the last time you ran the callback
 - There's a group of variables and related functions that are used only by the callback



An Example

```
import javax.swing as swing

startx = 0
starty = 0

def pressCallback(event):
    global startx, starty
    startx = event.x
    starty = event.y

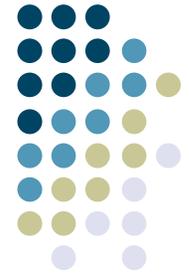
def releaseCallback(event):
    global startx, starty
    graphics = event.source.graphics
    graphics.drawLine(startx, starty, event.x, event.y)

if __name__ == "__main__":
    frame = swing.JFrame("Simple Drawing Program")
    canvas = swing.JPanel()
    canvas.preferredSize = (400, 400)
    frame.contentPane.add(canvas)
    frame.pack()
    frame.show()

    canvas.mousePressed = pressCallback
    canvas.mouseReleased = releaseCallback
```

O-O and Asynchronous Programming (cont'd)

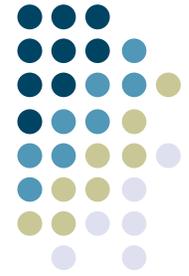
Georgia
Tech



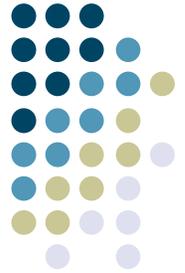
- Last mouse-down position needs to be remembered until the next time the callback is invoked
 - Can't save in a local variable, as it will be reset each time the callback is invoked
- Option #1: keep all of this cross-callback information in global variables
- Why is this a bad idea?
 - The information is specific to the drawing callbacks; nothing else should use it
 - By making it global, you increase program clutter, and the mental cycles needed to manage it
 - Worse: you run the risk that someone (you?) will misunderstand what the global variables are for, and reuse them for something else

O-O and Asynchronous Programming (cont'd)

Georgia
Tech



- The principle of data hiding:
 - Keep data as “close” to the behavior it controls as possible
 - Keep it inaccessible to everything else that doesn't need to use it
- The more of the inner workings of something you expose, the more likely it is to be used in the wrong way
- Option #2: object-oriented programming provides a nice way to handle this:
 - Each handler is an object that contains whatever information is necessary for it to execute properly
 - Internal state is not visible outside the handler object
 - Well-designed objects will allow the user to use them *only* in the way they were intended



Example of O-O Event Handling

```
import javax.swing as swing

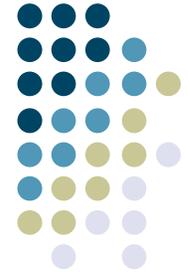
class Draw:
    def __init__(self):
        frame = swing.JFrame("Simple Drawing Program")
        canvas = swing.JPanel()
        canvas.preferredSize = (400, 400)
        frame.contentPane.add(canvas)
        frame.pack()
        frame.show()

        eventHandler = EventHandler()
        canvas.mousePressed = eventHandler.pressCallback
        canvas.mouseReleased = eventHandler.releaseCallback

class EventHandler:
    def pressCallback(self, event):
        self.startx = event.x
        self.starty = event.y

    def releaseCallback(self, event):
        graphics = event.source.graphics
        graphics.drawLine(self.startx, self.starty, event.x, event.y)

if __name__ == "__main__":
    draw = Draw()
```



Example of O-O Event Handling

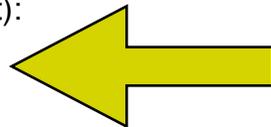
```
import javax.swing as swing
```

```
class Draw:
```

```
    def __init__(self):  
        frame = swing.JFrame("Simple Drawing Program")  
        canvas = swing.JPanel()  
        canvas.preferredSize = (400, 400)  
        frame.contentPane.add(canvas)  
        frame.pack()  
        frame.show()  
  
        eventHandler = EventHandler()  
        canvas.mousePressed = eventHandler.pressCallback  
        canvas.mouseReleased = eventHandler.releaseCallback
```

```
class EventHandler:
```

```
    def pressCallback(self, event):  
        self.startx = event.x  
        self.starty = event.y
```



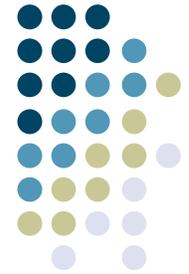
Record of last X,Y positions are stored in the EventHandler object. Not easily visible *outside* the object, easily shared among just these callbacks.

```
    def releaseCallback(self, event):  
        graphics = event.source.graphics  
        graphics.drawLine(self.startx, self.starty, event.x, event.y)
```

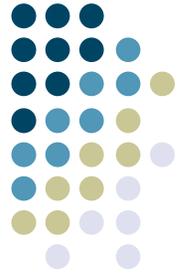
```
if __name__ == "__main__":  
    draw = Draw()
```

Objects As a Structuring Principle

Georgia
Tech



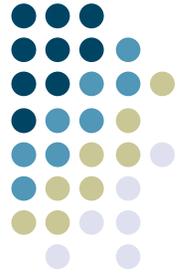
- Very often, the data in your program will have a natural structure
- In a drawing program, each drawing window will have its own contents, current mode, etc., that is not shared by any other open windows
 - All of this information can be grouped together into a *DrawingWindow* object
 - One *DrawingWindow* object per open window
 - No need to make the information needed by it global
- In a chat program, each ongoing chat has its own list of users, and its own message history
 - The user list, history, etc., could be grouped into a *Chat* object
 - One *Chat* object per ongoing chat
 - No need to make all of this information global



Creating Objects

- Where do new objects come from?
- In an event-driven program, they usually are created in response to events!
- Example:
 - User clicks “New Chat” button in GUI
 - Callback creates a new Chat object to represent the details of that chat

```
def newChat(event):  
    chat = Chat()  
    chat.users = [me, buddyList.selectedValue]
```

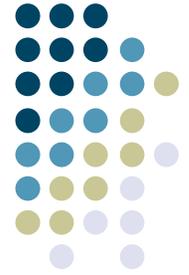


Managing Global State Effectively

- Some times, you really do need to store some stuff globally

```
allMyChats=[ ]
def newChat(event):
    chat = Chat()
    chat.users = [me, buddyList.selectedValue]
    allMyChats.append(chat)
```
- Useful idiom: keep track of objects through global data structure
 - Lists and Dictionaries are very helpful here
 - Use Lists for simple, ordered collection of stuff
 - Use Dictionaries when there's a natural *identifier* for stored objects
 - Extra bonus: since you update a collection by invoking a method on it (and not assigning to it), you avoid some of the scoping problems we talked about last week

Georgia
Tech



Paper Discussion, then
Lab Time!