

# “Chapter 0”: MVC review / Threading & Concurrency

CSCI 251

Android App Development

Part I:  
Model / View / Controller  
Review  
(courtesy of Prof. Lambert)

# TUI vs GUI

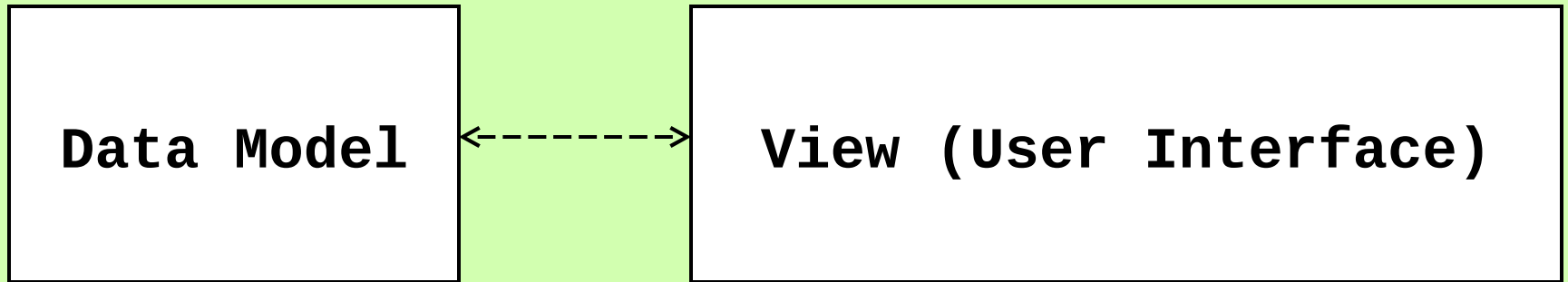
```
TestingWar — bash — 47x17
Player 1: 10 of Hearts
Player 2: 7 of Hearts
Cards go to Player 1
Player 1: Jack of Hearts
Player 2: 9 of Hearts
Cards go to Player 1
Player 1: Ace of Spades
Player 2: 2 of Diamonds
Cards go to Player 2
Player 1: 8 of Clubs
Player 2: 2 of Hearts
Cards go to Player 1
Player 1: Ace of Clubs
Player 2: 10 of Spades
Cards go to Player 2
Player 1 wins, 30 to 22!
nile:testingwar lambertk$
```

Text-based I/O  
Sequential process



Direct manipulation  
Event-driven process

# The Model/View Pattern



The *data model* consists of software components that manage a system's data

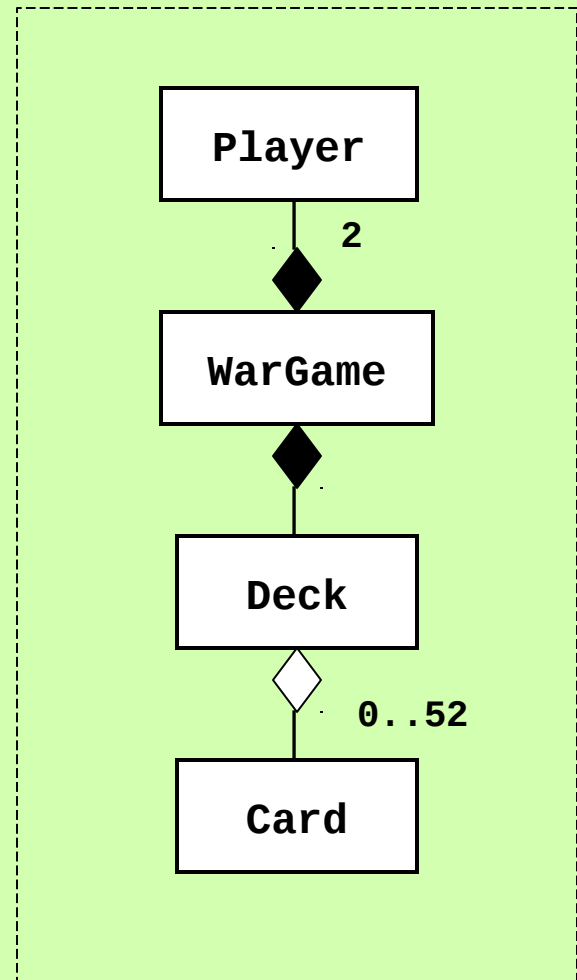
The *view* consists of software components that allow human users to view and interact with the data model

The view can be a TUI or a GUI, on the same data model

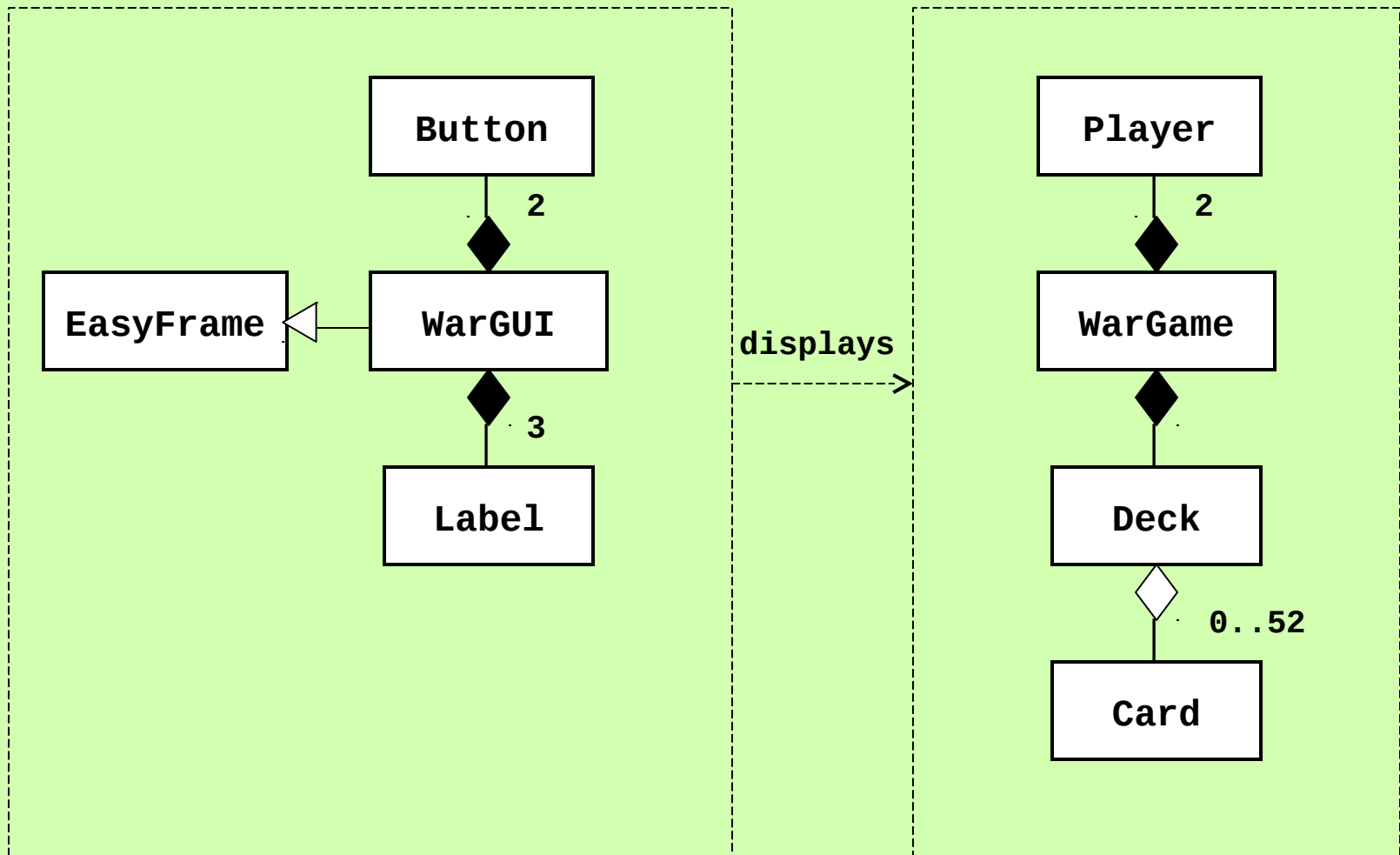
# Model/View Pattern

- Classes in the data model, such as **WarGame**, **Player**, **Deck**, and **Card**, manage the application's data
- Classes in the view, such as **WarGUI**, **EasyFrame**, **PhotoImage**, **Button**, and **Label**, present the data to users

# The Data Model for the War Game

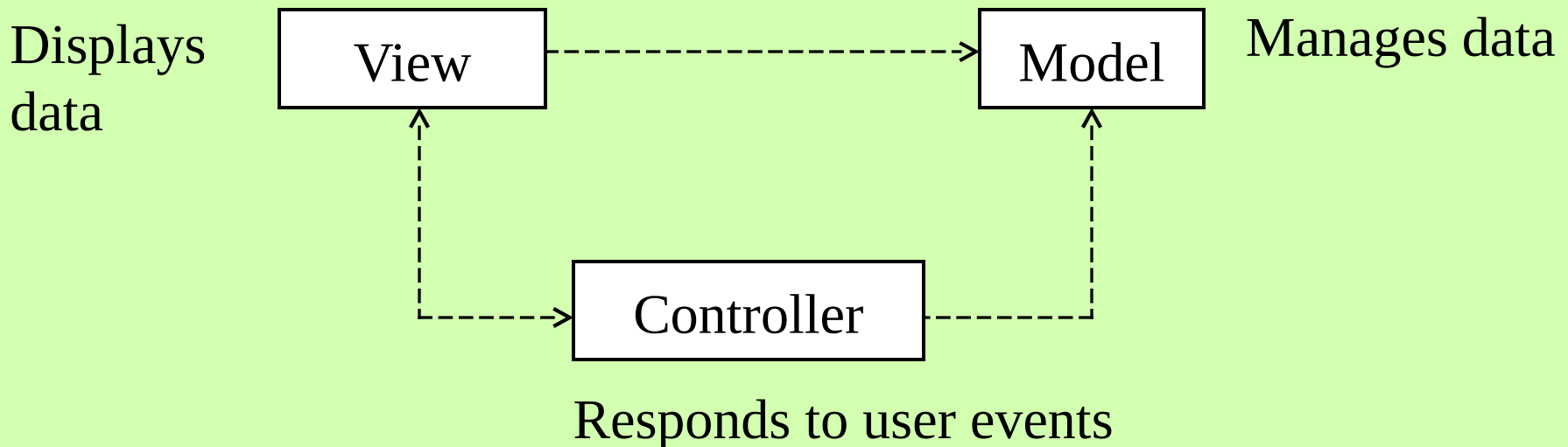


# The Data Model and the View (GUI)



# Model/View/Controller Pattern

GUI-based, event-driven programs can be further decomposed by gathering code to handle user interactions into a third component called the *controller*





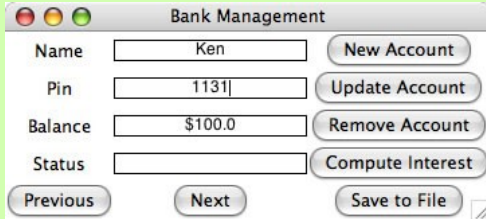
# The Controller

- In Python, the controller consists of the methods that respond to user events
- These methods are defined in the main view class, and are associated with the **command** attribute of the buttons
- Other types of widgets can have their own event-handling methods

# Event-Driven Programming

- Set up a window with its widgets
- Connect it to a data model
- Wait for users to press buttons, enter text, drag the mouse, etc.
- Respond to these events by running methods that update the data model and the view

# A Banking System



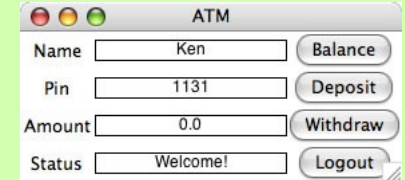
Bank Management

Name:

Pin:

Balance:

Status:



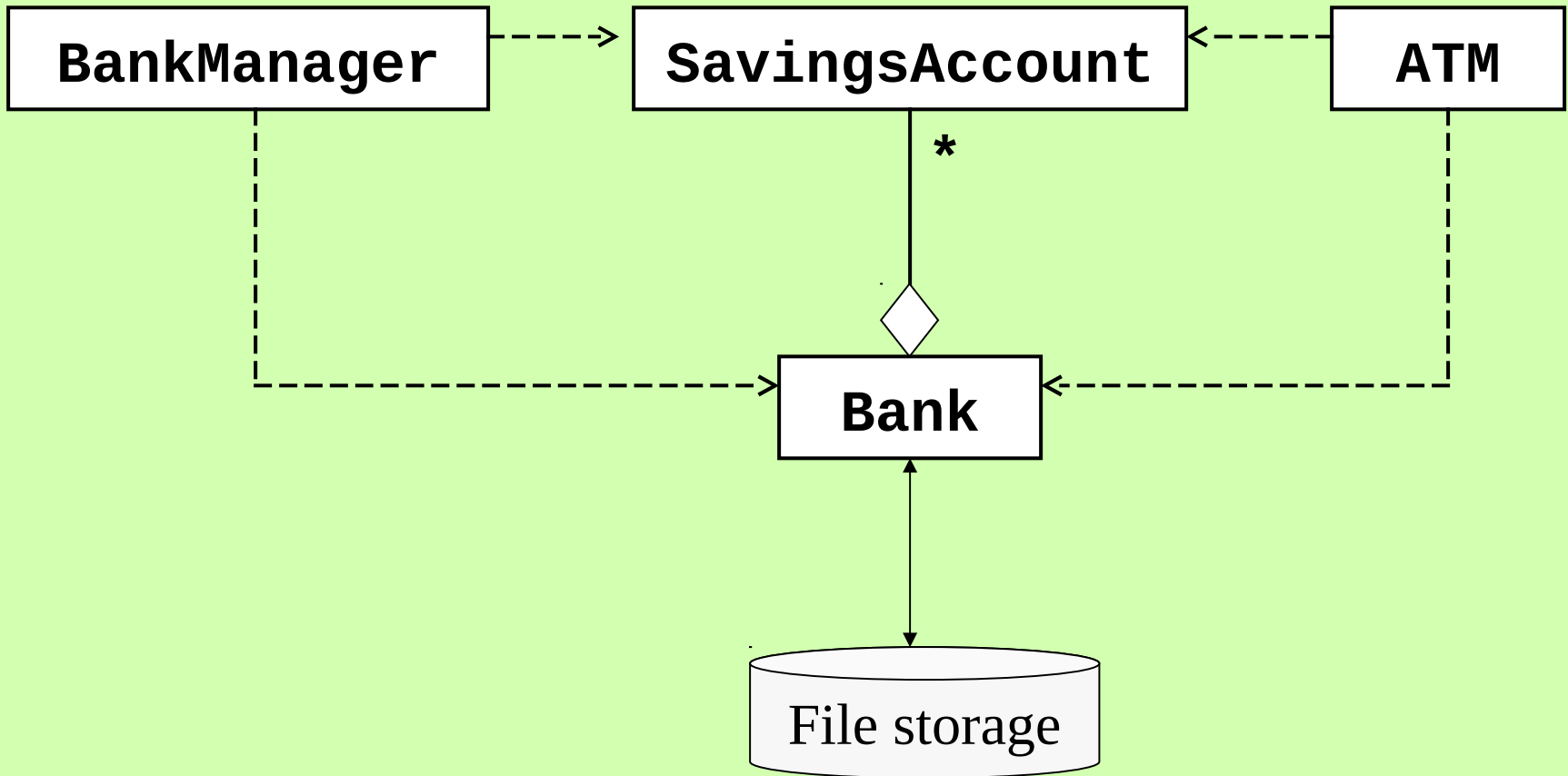
ATM

Name:

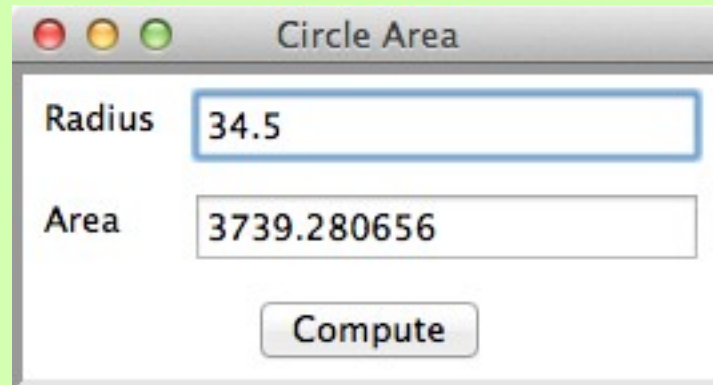
Pin:

Amount:

Status:



# MVC / GUI: Behind the Scenes with a Simpler Example



# MVC / GUI: Behind the Scenes

```
class CircleWithGUI(EasyFrame):
    """Computes and displays the area of a circle."""

    def __init__(self):
        """Sets up the window and widgets."""
        EasyFrame.__init__(self, title = "Circle Area")

        # Label and field for the input
        self.addLabel(text = "Radius",
                      row = 0, column = 0)
        self.radiusField = self.addFloatField(value = 0.0,
                                              row = 0,
                                              column = 1)

        # Label and field for the output
        self.addLabel(text = "Area",
                      row = 1, column = 0)
        self.areaField = self.addFloatField(value = 0.0,
                                              row = 1,
                                              column = 1)

        # The command button
        self.addButton(text = "Compute", row = 2, column = 0,
                       colspan = 2, command = self.computeArea)
```

```
class CircleWithGUI(EasyFrame):
    """Computes and displays the area of a circle."""

    . . .

# The event handling method for the button
def computeArea(self):
    """Inputs the radius, computes the area,
    and outputs the result."""
    radius = self.radiusField.getNumber()
    area = math.pi * radius ** 2
    self.areaField.setNumber(area)

#Instantiate and pop up the window.
if __name__ == "__main__":
    CircleWithGUI().mainloop()
```

```
class CircleWithGUI(EasyFrame):
    """Computes and displays the area of a circle."""
    . . .

# The event handling method for the button
def computeArea(self):
    """Inputs the radius, computes the area,
    and outputs the result."""
    radius = self.radiusField.getNumber()
    area = math.pi * radius ** 2
    self.areaField.setNumber(area)

#Instantiate and pop up the window.
if __name__ == "__main__":
    CircleWithGUI().mainloop()
```

# What will happen now?

```
class CircleWithGUI(EasyFrame):
    """Computes and displays the area of a circle."""
    . . .

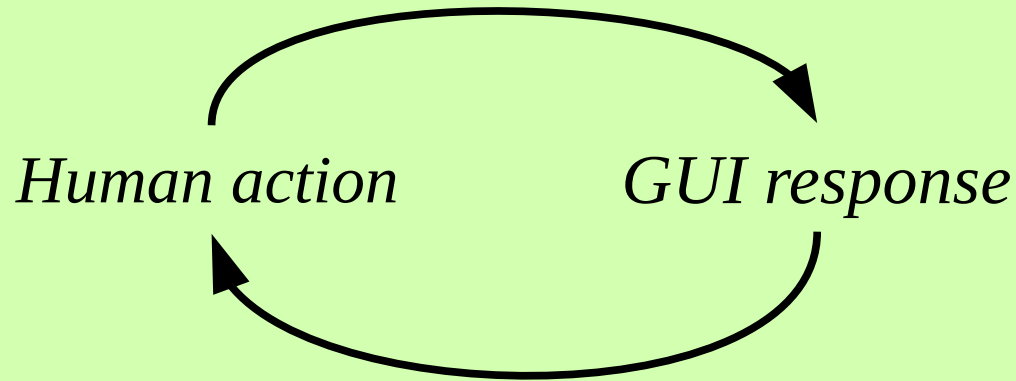
# The event handling method for the button
def computeArea(self):
    """Inputs the radius, computes the area,
    and outputs the result."""
    radius = self.radiusField.getNumber()
    area = math.pi * radius ** 2
    self.areaField.setNumber(area)

#Instantiate and pop up the window.
if name == " main ":
    print("About to enter mainloop ...")
    CircleWithGUI().mainloop()
    print("Here I am, rock you like a hurricane!")
```

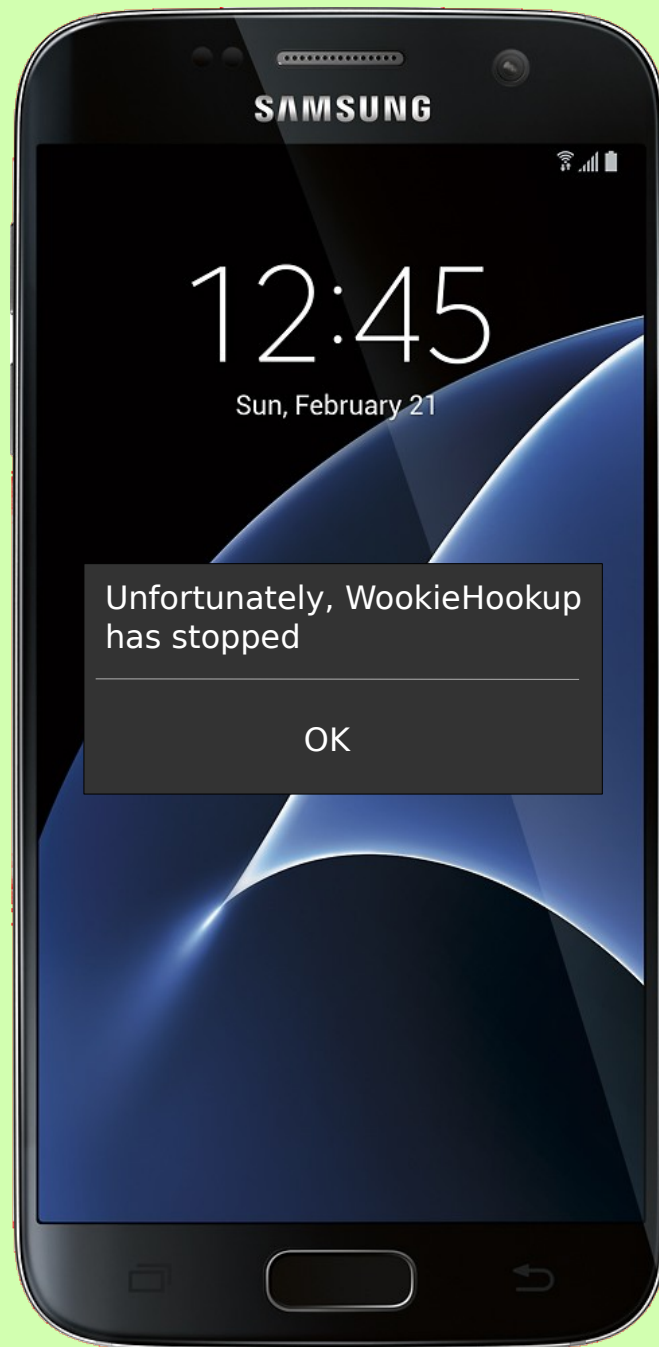


# So what's the problem?

- The entire program (model / view / controller ) is executing on a single *thread*.
- For a simple kind of interaction, this approach is fine:



- On a *real-time device* (phone, tablet, robot), this approach is too constraining and leads to ...



SAMSUNG



12:45

Sun, February 21

Unfortunately, WookieHookup  
has stopped

OK

# So what's the solution?

- *Multi-threading* : The model (computation, downloading, communication, etc.) component must run *concurrently* with the view/controller component, each on its own thread.
- In brief: **threads execute independently of each other, but share the same data.**
- Let's look at threading (a.k.a. *concurrency*) in more detail, apart from Android ...

Part II:  
Multi-threading /  
Concurrency

# Thread vs. Process

- *Process* : an executing instance of a program\*
- In Unix (Linux, OS X) we can use the **top** (or **ps**) command to tell us what processes are running:

```
levy@kern: ~
top - 17:23:40 up 3:20, 1 user, load average: 0.45, 0.36, 0.21
Tasks: 263 total, 1 running, 262 sleeping, 0 stopped, 0 zombie
%Cpu(s): 3.6 us, 0.9 sy, 0.0 ni, 95.2 id, 0.3 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 16290224 total, 10672312 free, 2969320 used, 2648592 buff/cache
KiB Swap: 16633852 total, 16633852 free, 0 used, 12360680 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
10774	levy	20	0	6508740	660824	59932	S	17.6	4.1	1:04.10	java
3858	levy	20	0	1613940	210260	65164	S	9.3	1.3	3:30.06	compiz
3001	root	20	0	781996	240264	215372	S	7.3	1.5	6:09.25	Xorg
3936	root	20	0	367604	10680	6064	S	1.7	0.1	0:00.93	udisksd
11124	levy	20	0	625260	31564	26640	S	1.3	0.2	0:00.17	gnome-screensho
7	root	20	0	0	0	0	S	0.3	0.0	0:04.93	rcu_sched
2943	root	20	0	19472	260	0	S	0.3	0.0	0:00.88	irqbalance
3514	levy	20	0	523520	27164	21392	S	0.3	0.2	0:05.74	bamfdaemon
3745	levy	20	0	580792	50276	26124	S	0.3	0.3	0:15.19	unity-panel-ser
4076	levy	20	0	1208424	231764	108060	S	0.3	1.4	5:23.80	chrome
9909	root	20	0	0	0	0	S	0.3	0.0	0:00.81	kworker/u16:1
10845	levy	20	0	100020	3428	3132	S	0.3	0.0	0:00.09	adb
10862	levy	20	0	660780	35564	28624	S	0.3	0.2	0:00.39	gnome-terminal-
1	root	20	0	185288	5912	3972	S	0.0	0.0	0:01.25	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.01	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:00.01	ksoftirqd/0
5	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/0:0H
8	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_bh
9	root	rt	0	0	0	0	S	0.0	0.0	0:00.00	migration/0
10	root	rt	0	0	0	0	S	0.0	0.0	0:00.05	watchdog/0
11	root	rt	0	0	0	0	S	0.0	0.0	0:00.05	watchdog/1
12	root	rt	0	0	0	0	S	0.0	0.0	0:00.00	migration/1
13	root	20	0	0	0	0	S	0.0	0.0	0:00.04	ksoftirqd/1

\* definition courtesy of Dr. J.S. Plank

# Anatomy of a Process

- Each process takes up its own space in the computer's memory (RAM), sub-divided into **segments** based on how the process uses it:
  - **Stack**: memory available locally in a function (automatically allocated / deallocated when the function is called / returns).
  - **Heap**: *dynamic* memory; i.e., allocated at run-time
  - **Globals** (a.k.a. **data**): allocated at compile time; available throughout process
- A precise example would require assembly language or C, but we can get a good sense use Java ...

```
import java.util.Random;

class Example {

    public static double PI = 3.14159;

    public double circleArea(double radius)
    {
        double a = PI * radius * radius;

        return a;
    }

    public static void main(String [] args)
    {
        int n = (new Random()).nextInt();

        Integer [] a = new Integer [n];
    }
}
```

```
import java.util.Random;

class Example {

    public static double PI = 3.14159;

    public double circleArea(double radius)
    {
        double a = PI * radius * radius;

        return a;
    }

    public static void main(String [] args)
    {
        int n = (new Random()).nextInt();

        Integer [] a = new Integer [n];
    }
}
```

Global data  
(visible  
everywhere)



```
import java.util.Random;

class Example {

    public static double PI = 3.14159;

    public double circleArea(double radius)
    {
        double a = PI * radius * radius;

        return a;
    }

    public static void main(String [] args)
    {
        int n = (new Random()).nextInt();

        Integer [] a = new Integer [n];
    }
}
```

Stack vars  
(available  
only in this  
method)

```
import java.util.Random;

class Example {

    public static double PI = 3.14159;

    public double circleArea(double radius)
    {
        double a = PI * radius * radius;

        return a;
    }

    public static void main(String [] args)
    {
        int n = (new Random()).nextInt();

        Integer [] a = new Integer [n];
    }
}
```

Heap  
allocation  
(size unknown  
till run-time)

# Thread: A “Lightweight” Process

- Now we have the vocabulary to talk about threads: a *thread is a “lightweight” process that has its own stack but shares heap and globals with other threads in the same process.*
- Q: Why must each thread have its own stack?

# Thread: A “Lightweight” Process

- Now we have the vocabulary to talk about threads: a *thread is a “lightweight” process that has its own stack but shares heap and globals with other threads in the same process.*
- Q: Why must each thread have its own stack?
- A: So it can execute function calls (i.e., do its own work) independently of other threads.

# Threads in Python

```
from threading import Thread
from time import sleep

class Value:

    def __init__(self):
        self.value = 0

    def __str__(self):
        return str(self.value)

def updater(v):

    while True:

        v.value += 1
        sleep(1)

val = Value()

thread = Thread(target=updater, args=(val,))
thread.daemon = True
thread.start()

while True:

    answer = input('Ready to quit ? (y/n) ')
    if answer == 'y':
        break
    print('Okay, new value is ' + str(val))
```

# Threads in Python

```
from threading import Thread
from time import sleep

class Value:
    def __init__(self):
        self.value = 0
    def __str__(self):
        return str(self.value)

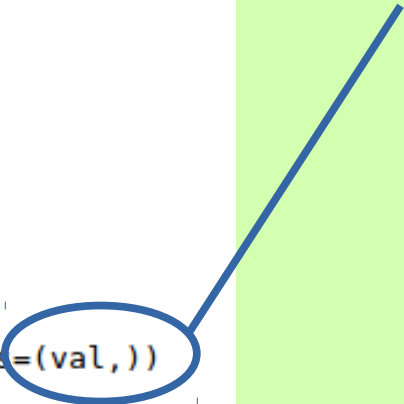
def updater(v):
    while True:
        v.value += 1
        sleep(1)

val = Value()

thread = Thread(target=updater, args=(val,))
thread.daemon = True
thread.start()

while True:
    answer = input('Ready to quit ? (y/n) ')
    if answer == 'y':
        break
    print('Okay, new value is ' + str(val))
```

Note that args must be a tuple: so if just one arg, it must end in a comma.



# Threading: Additional Remarks

- As this Python example shows, threading is essential for **asynchronous** computing – i.e., when different kinds of activity have to be happening at their own time scales.
- Many Android apps won't need to use threading; however: *whenever you have some “heavy lifting” to do outside your main (GUI) thread (loading large images, downloading mp3's, etc.), you should think about threading.*
- Android Development Kit provides the `AsyncTask` class to help with this.
- Every solution creates another problem – can you think of what problem(s) threading creates?