“Chapter 0”: MVC review / Threading & Concurrency

CSCI 251

Android App Development

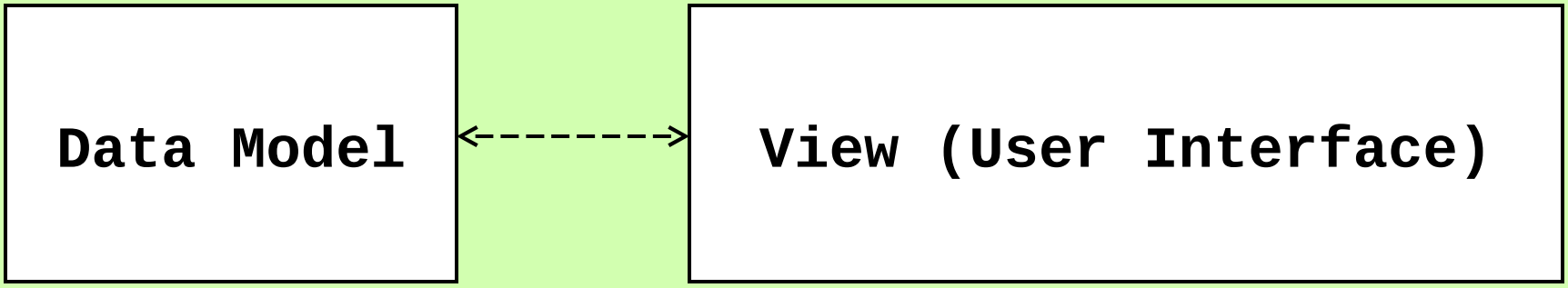
Part I:
Model / View / Controller
Review
(courtesy of Prof. Lambert)

TUI vs GUI

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

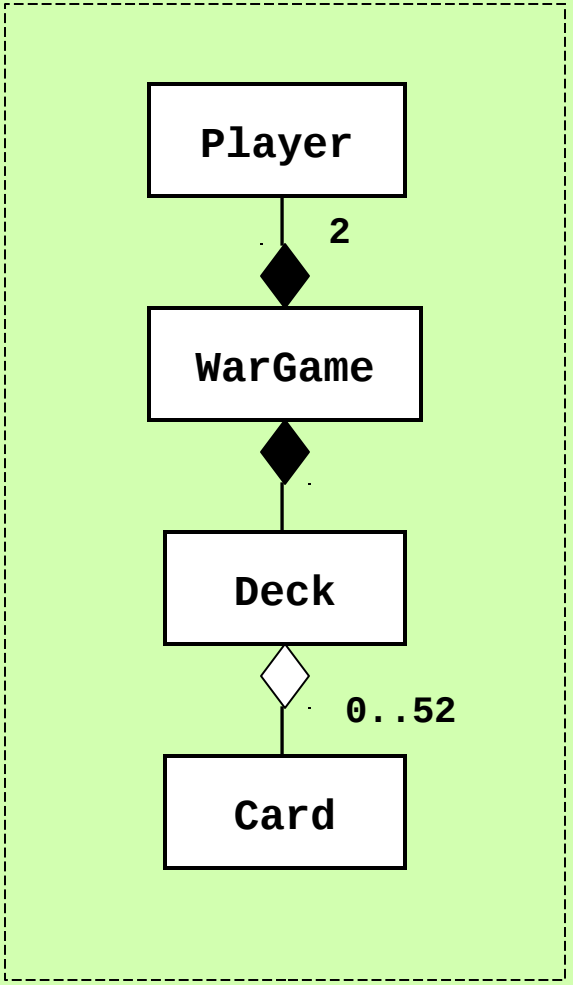
The view can be a TUI or a GUI, on the same data model

Model/View Pattern

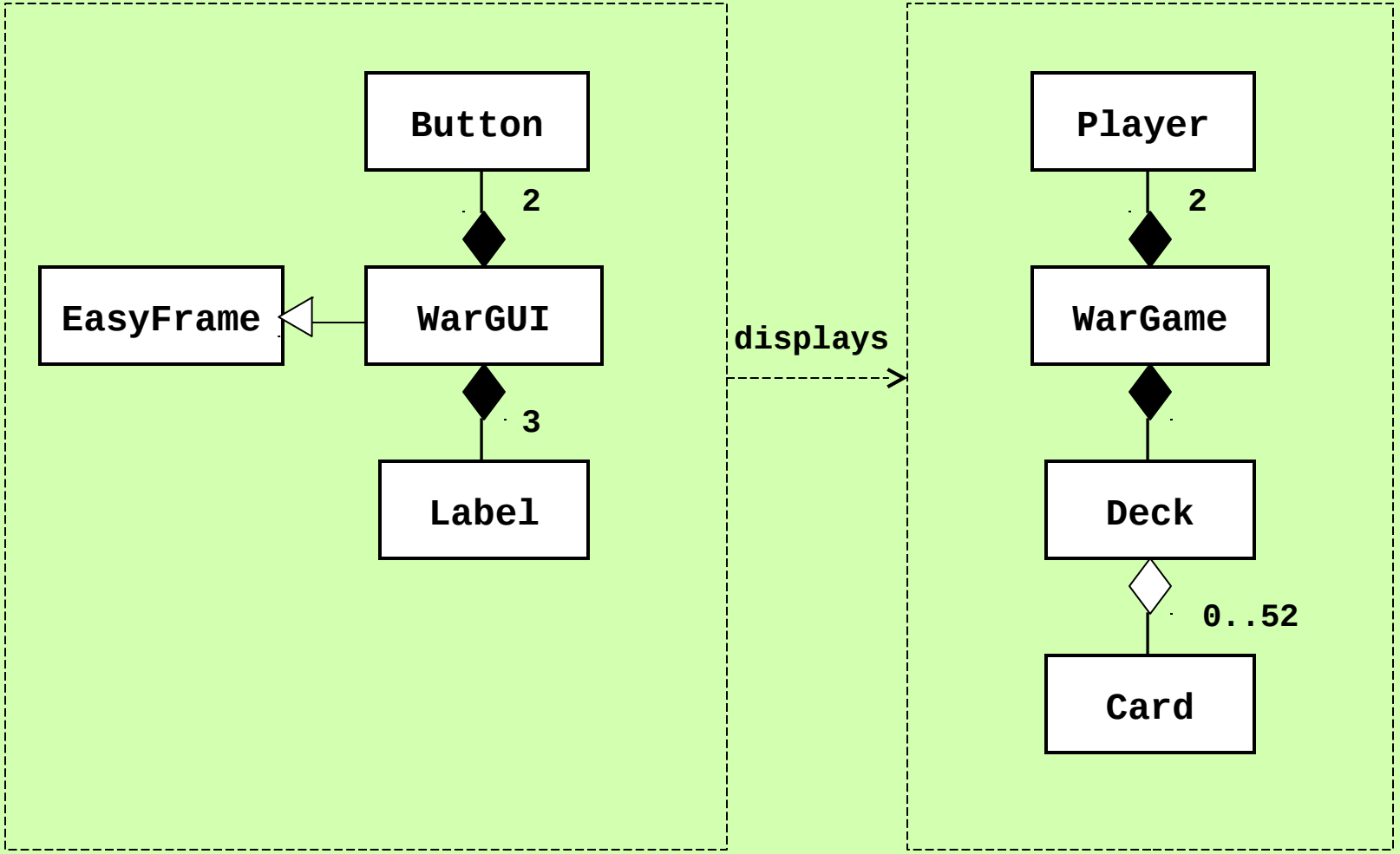
- Classes in the data model, such as **`WarGame`**, **`Player`**, **`Deck`**, and **`Card`**, manage the application's data

- Classes in the view, such as **`WarGUI`**, **`EasyFrame`**, **`PhotoImage`**, **`Button`**, and **`Label`**, present the data to users

The Data Model for the War Game

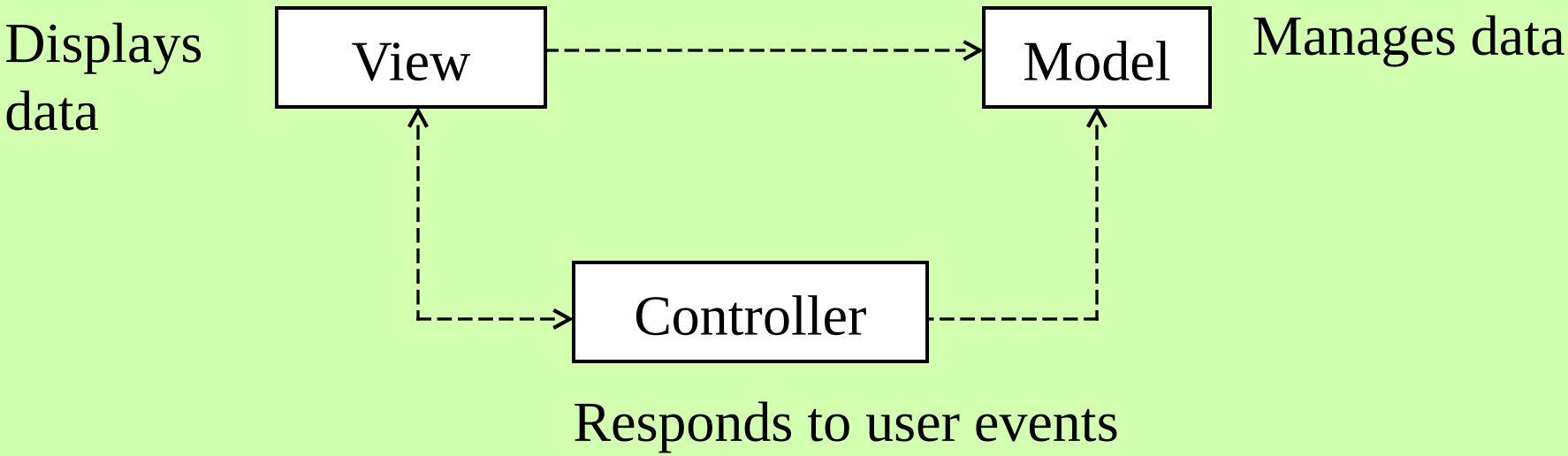


The Data Model and the View (GUI)



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*



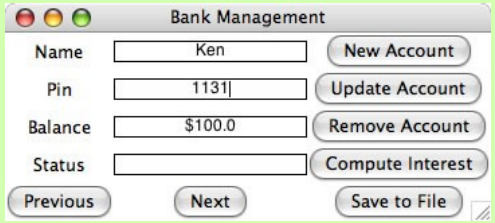
The Controller

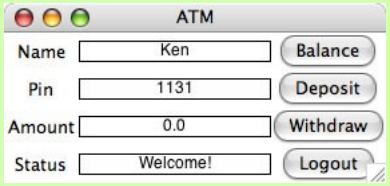
- In Python, the controller consists of the methods that respond to user events
- These methods are defined in the main view class, and are associated with the **`command`** attribute of the buttons
- Other types of widgets can have their own event-handling methods

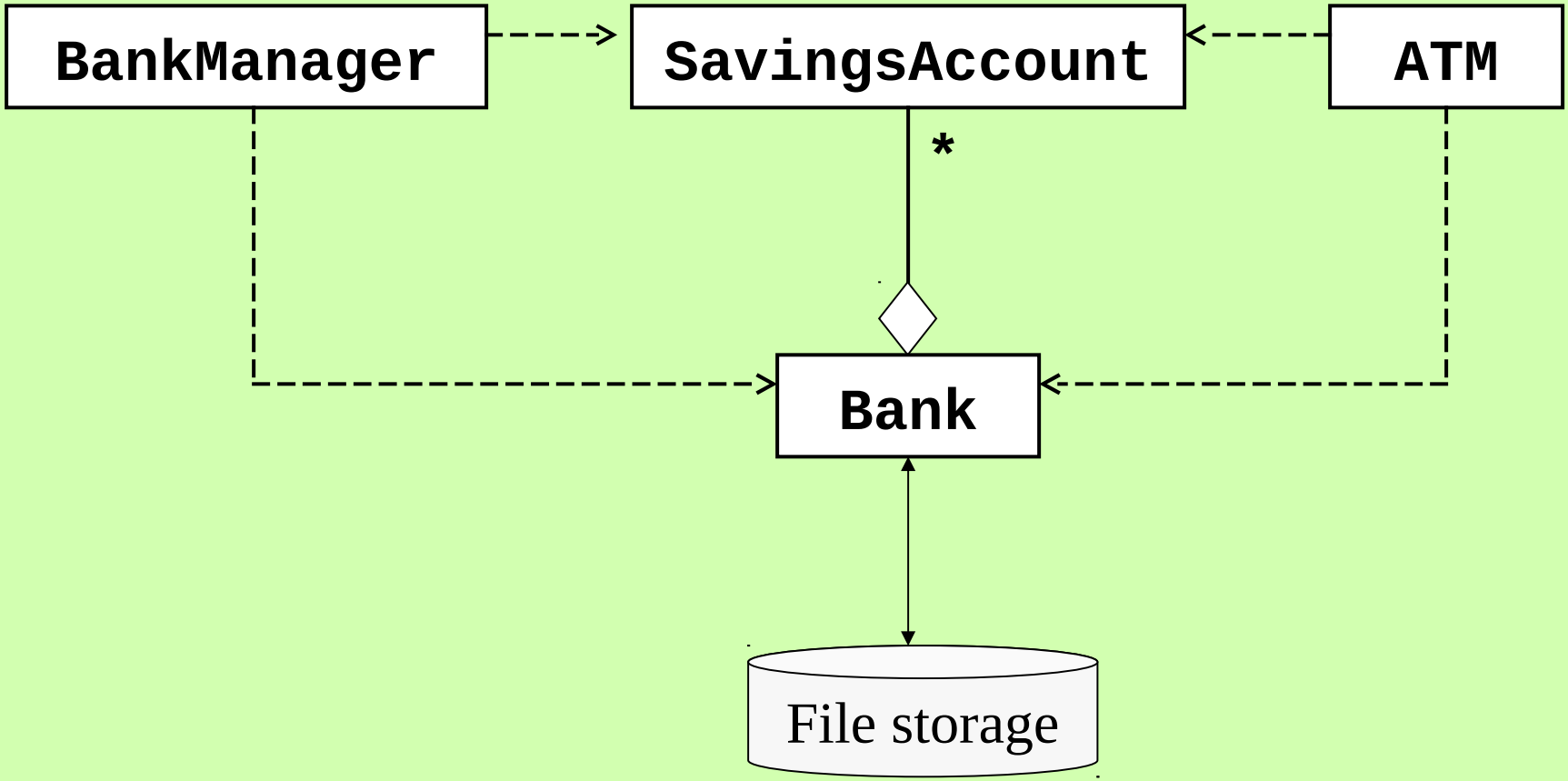
Event-Driven Programming

- 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

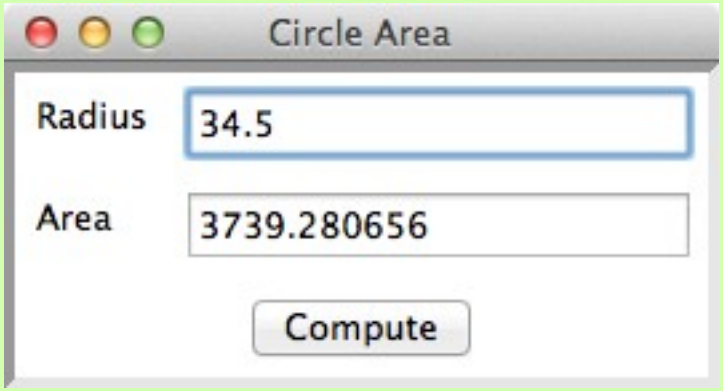
A Banking System







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,
                       columnspan = 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()
```

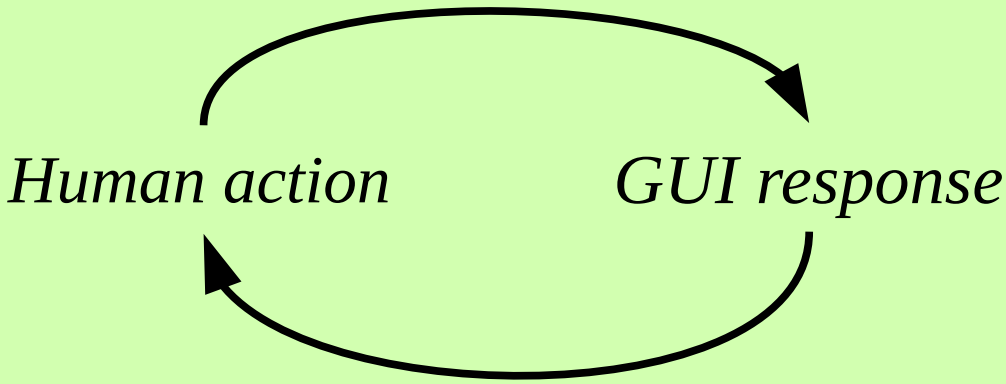
```
class CircleWithGUI(EasyFrame):
    """Computes and displays the area of a circle."""

    . . .


    # 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

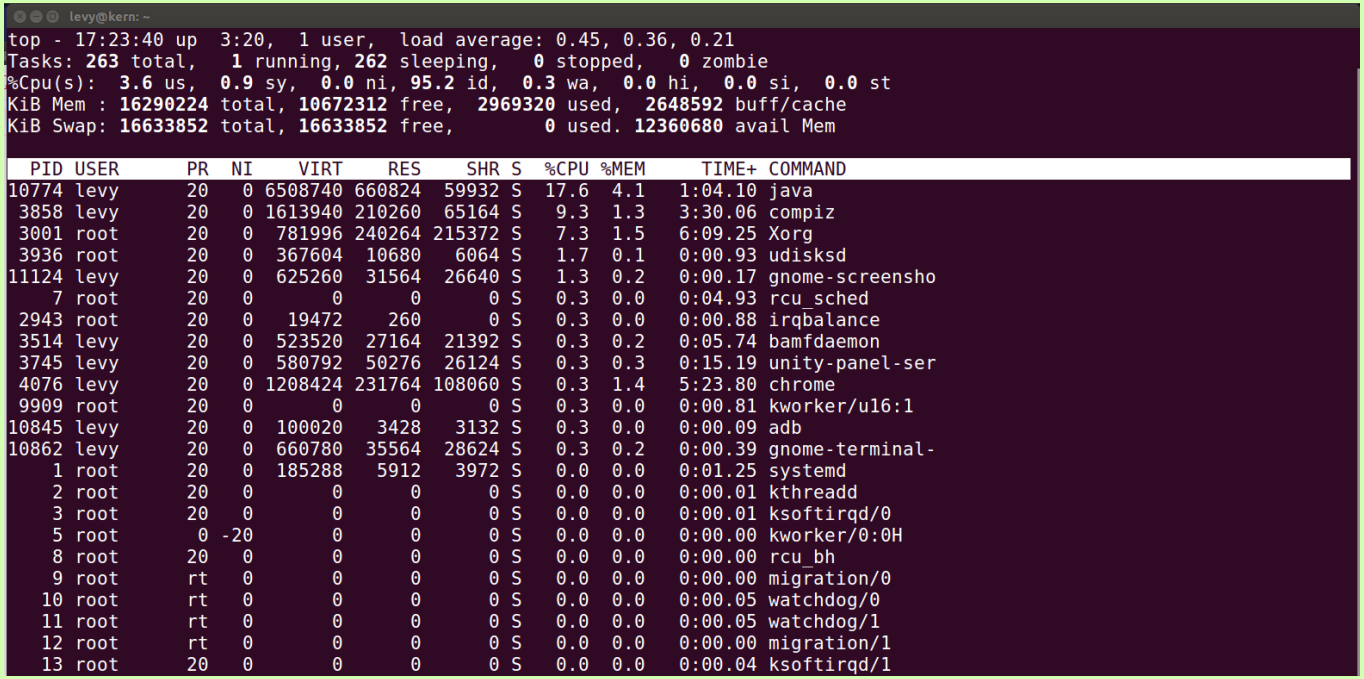
So what's the solution?

- *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:



* 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 ...

```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];
    }

}
```

```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];
    }

}
```

Global data (visible everywhere)

```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];
    }

}
```

Stack vars (available only in this method)

```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];
    }

}
```

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?