Programmieren II

Threads

Alexander Fraser

fraser@cl.uni-heidelberg.de

(slides based on material from David Matuszek, U Penn and a few slides from others, see attributions on individual slides)

July 10th, 2014

Admin

- Reminder: Commitment-Frist until 13.07
- CL students register if they have met the requirements
- Non-CL students must email the ICL Sekretariat (there is no form) to be registered if they have met the requirements

- And some more breaking news...

Assignment 8

- We were thinking about doing a Bonus Blatt after Assignment 8
- However, we are actually out of time
- Therefore Assignment 8 is OPTIONAL
  - Highly recommended to do it though!
  - You will most likely use opennlp in the future a lot (or Stanford NLP which is quite similar)

Outline

- Recap
  - GUIs with Swing
  - Anonymous inner classes
  - Listeners
- Event loops
- Threads

How to build a GUI with Swing

- Create a window in which to display things—usually a JFrame (for an application), or a JApplet
- Use the setLayout(LayoutManager ***manager***) method to specify a layout manager
- Create some Components, such as buttons, panels, etc.
- Add your components to your display area, according to your chosen layout manager
- Write some Listeners and attach them to your Components
  - Interacting with a Component causes an Event to occur
  - A Listener gets a message when an interesting event occurs, and executes some code to deal with it
- Display your window

Anonymous inner classes

- Anonymous inner classes are convenient for short code (typically a single method)

    `b.addActionListener(`***anonymous inner class***`);`

- The *anonymous inner class* can be either:

    `new` ***Superclass***`(`***args***`) {` ***body*** `}`

    or

    `new` ***Interface***`() {` ***body*** `}`

- Notice that no class name is given--only the name of the superclass or interface
    - If it had a name, it wouldn’t be anonymous, now would it?
- The *args* are arguments to the superclass’s constructor (interfaces don’t have constructors)

Using an anonymous inner class

- Instead of:

  -
    ```
    okButton.addActionListener(new MyOkListener());

    class MyOkListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            // code to handle OK button click
        }
    }
    ```
- You can do this:

  -
    ```
    okButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            // code to handle OK button click
        }
    }
    );
    ```
- Keep anonymous inner classes very short (typically just a call to one of your methods), as they can really clutter up the code

Suggested program arrangement 2

```
class SomeClass extends JFrame {
    // Declare components as instance variables
    // JFrame frame; // Don’t need this
    JButton button;
    public static void main(String[] args) {
        new SomeClass().createGui();
    }
    // Define components and attach listeners in a method
    void createGui() {
        // frame = new JFrame();  // Don’t need this
        button = new JButton("OK");
        add(button); // Was: frame.add(button);
        button.addActionListener(new MyOkListener());
    }
    // Use an inner class as your listener
    class MyOkButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            // Code to handle button click goes here
        }
    }
}
```

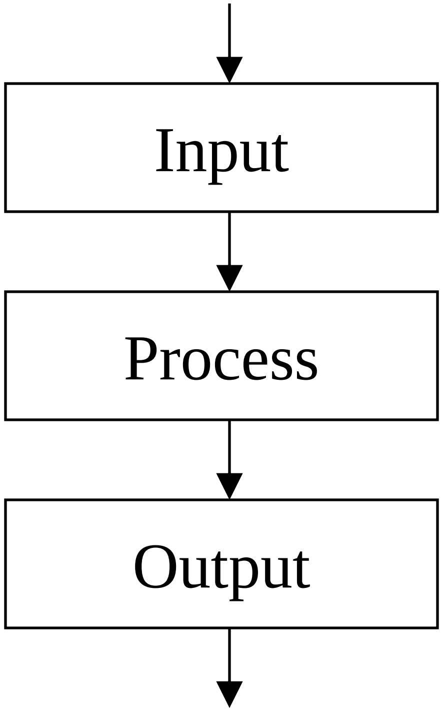
Inner Classes

- Note that the previous example defined a named inner class
- This is not recommended
- **Anonymous** inner classes are OK (I personally don't use them that much)
- The Java compiler saves inner classes in: OuterClass$InnerClass.class
- Anonymous classes are numbered: OuterClass$1.class

Event loops

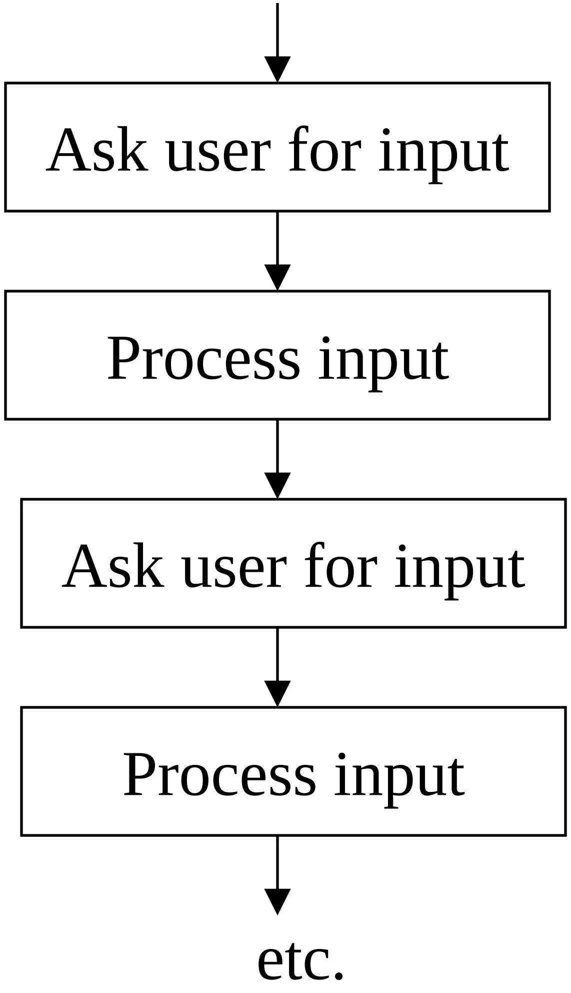
Programming in prehistoric times

- Earliest programs were all “batch” processing
- There was no interaction with the user



Very early interactive programs

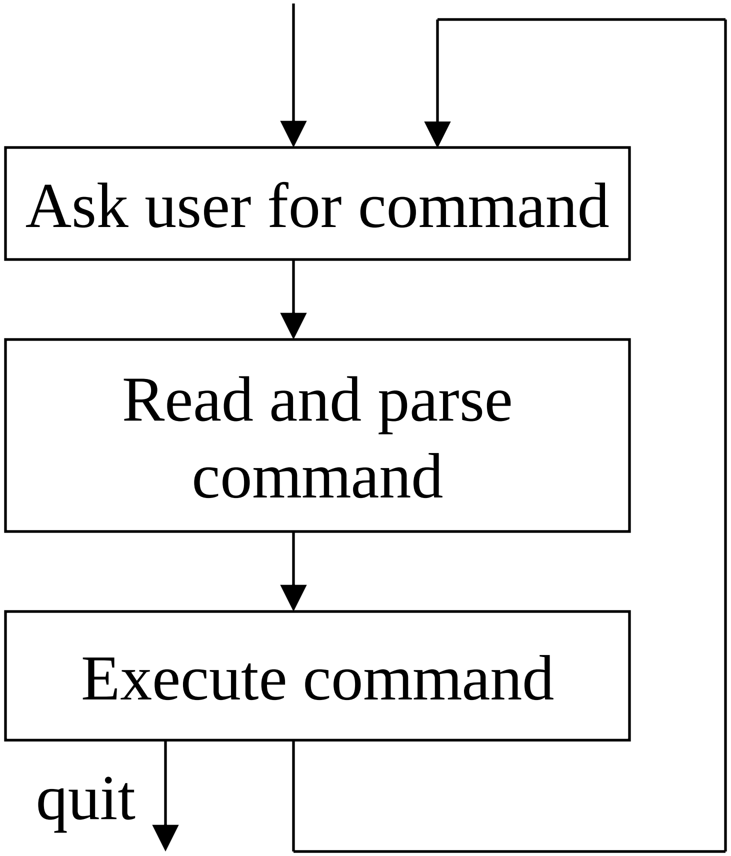
- BASIC was an early interactive language
- Still a central computer, with terminals
- Style of interaction was "filling out forms"



Command-driven programs

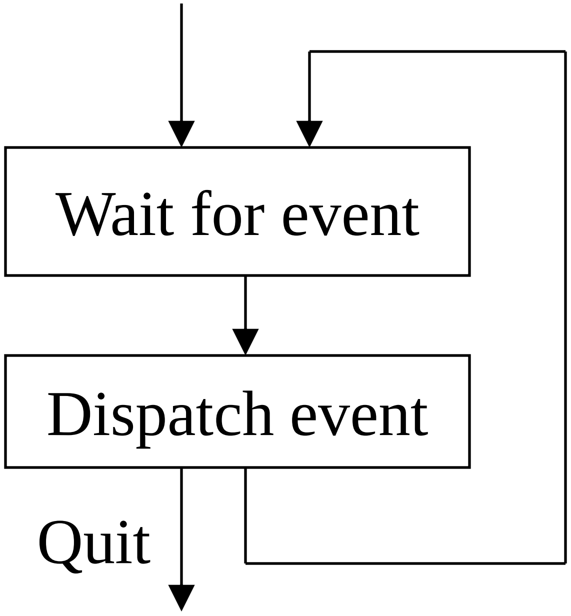
(30 years ago)

- Allow the user to enter "commands"
- Much more flexible
- Still only a single source of inputs
- Not good enough for modern programs



Modern event-driven programs

- Multiple sources of input
  - mouse clicks
  - keyboard
  - timers
  - external events
- A new program structure is required



Java hides the event loop

- The event loop is built into Java GUIs
  - GUI stands for Graphical User Interface
- Interacting with a GUI component (such as a button) causes an event to occur
- An Event is an object
- You create Listeners for interesting events
  - Listener is an *interface*; you create a Listener by implementing that interface
- The Listener method gets the Event as a parameter

Building a GUI

- To build a GUI in Java,
  - Create some Components
  - Use a layout manager to arrange the Components in a window
  - Add Listeners, usually one per Component
  - Put methods in the Listeners to do whatever it is you want done
- That's it!

Vocabulary I

- Event – an object representing an external happening that can be observed by the program
- event-driven programming – A style of programming where the main thing the program does is respond to Events
- event loop – a loop that waits for an Event to occur, then dispatches it to the appropriate code
- GUI – a Graphical User Interface (user interacts with the program via things on the screen)

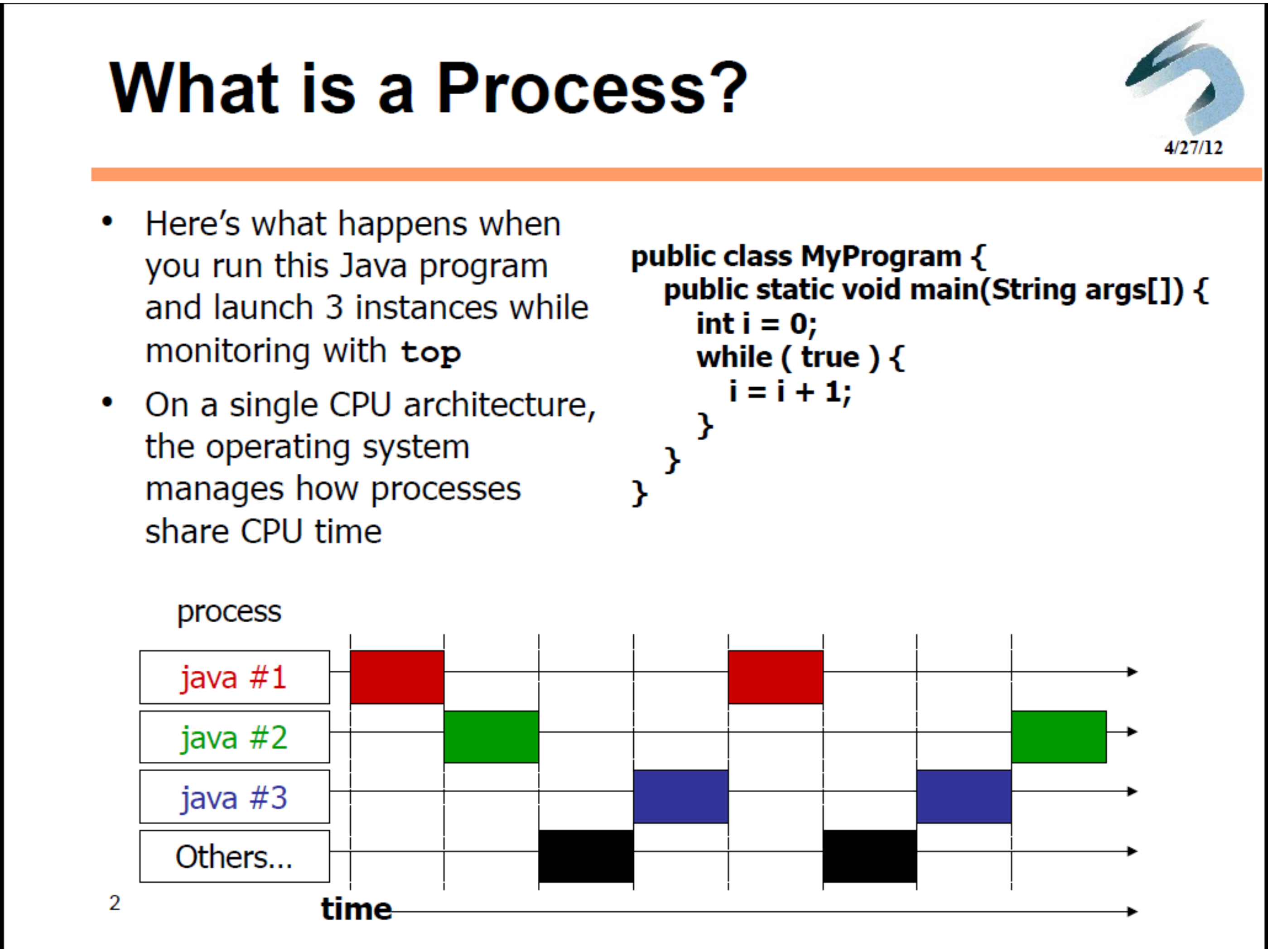
Vocabulary II

- Component – an interface element, such as a Button or a TextField
- Layout Manager – an object (provided by Java) that arranges your Components in a window
- Listener – an interface you implement to execute some code when an Event occurs

- I uploaded a file called ColorWindow.java to Moodle.
  - Look at this program to see how Listeners work in detail.

Multiprocessing

- Modern operating systems are multiprocessing
- Appear to do more than one thing at a time
- Three general approaches:
  - Cooperative multiprocessing
  - Preemptive multiprocessing
  - Really having multiple processors



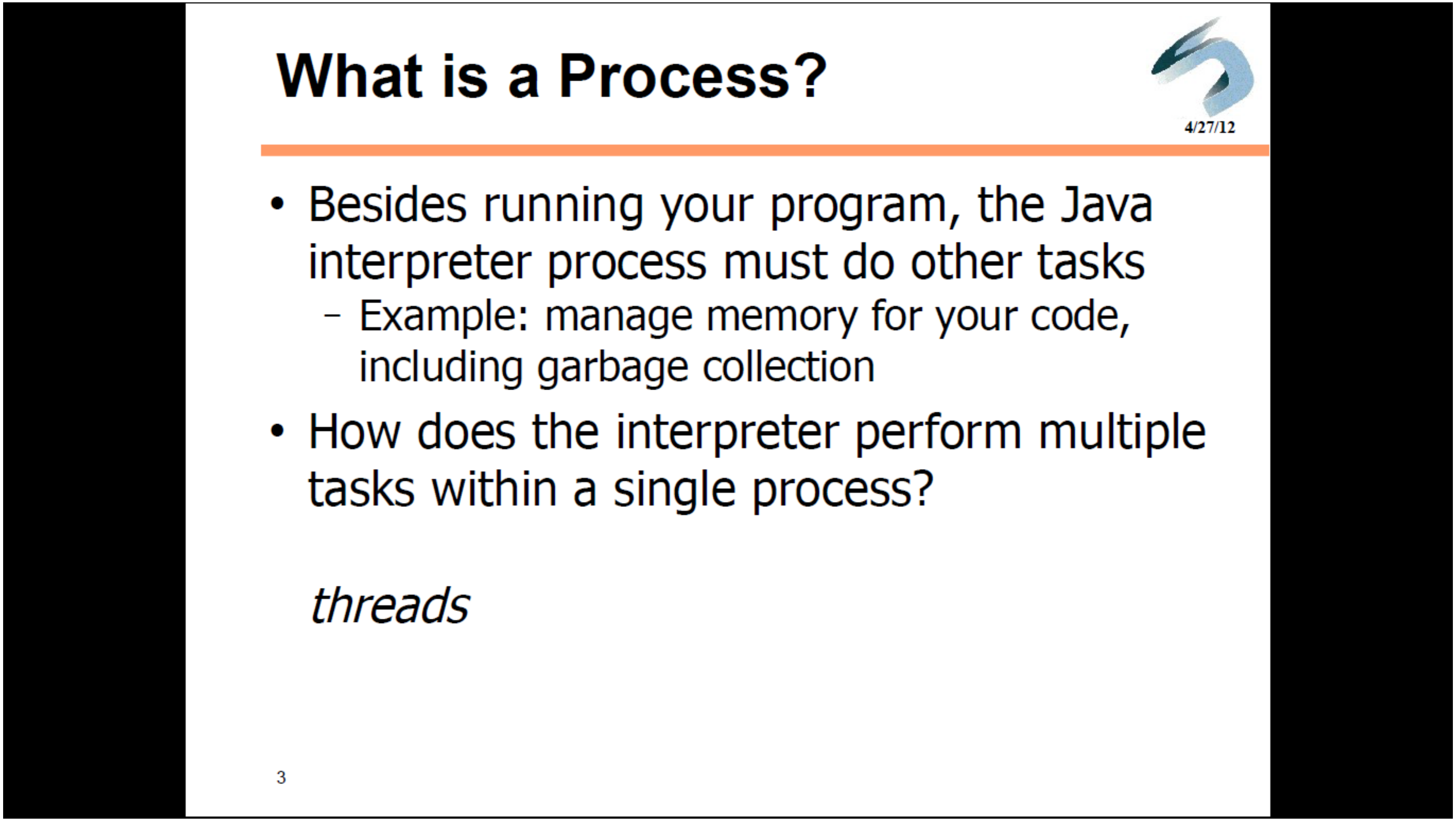
What is a Process?

4/27/12

- Here’s what happens when you run this Java program and launch 3 instances while monitoring with `top`
- On a single CPU architecture, the operating system manages how processes share CPU time

```
public class MyProgram {
   public static void main(String args[]) {
      int i = 0;
      while ( true ) {
         i = i + 1;
      }
   }
}
```

Slide from Travis Brown, Rochester Tech



What is a Process?

4/27/12

- Besides running your program, the Java interpreter process must do other tasks
  - Example: manage memory for your code, including garbage collection
- How does the interpreter perform multiple tasks within a single process?

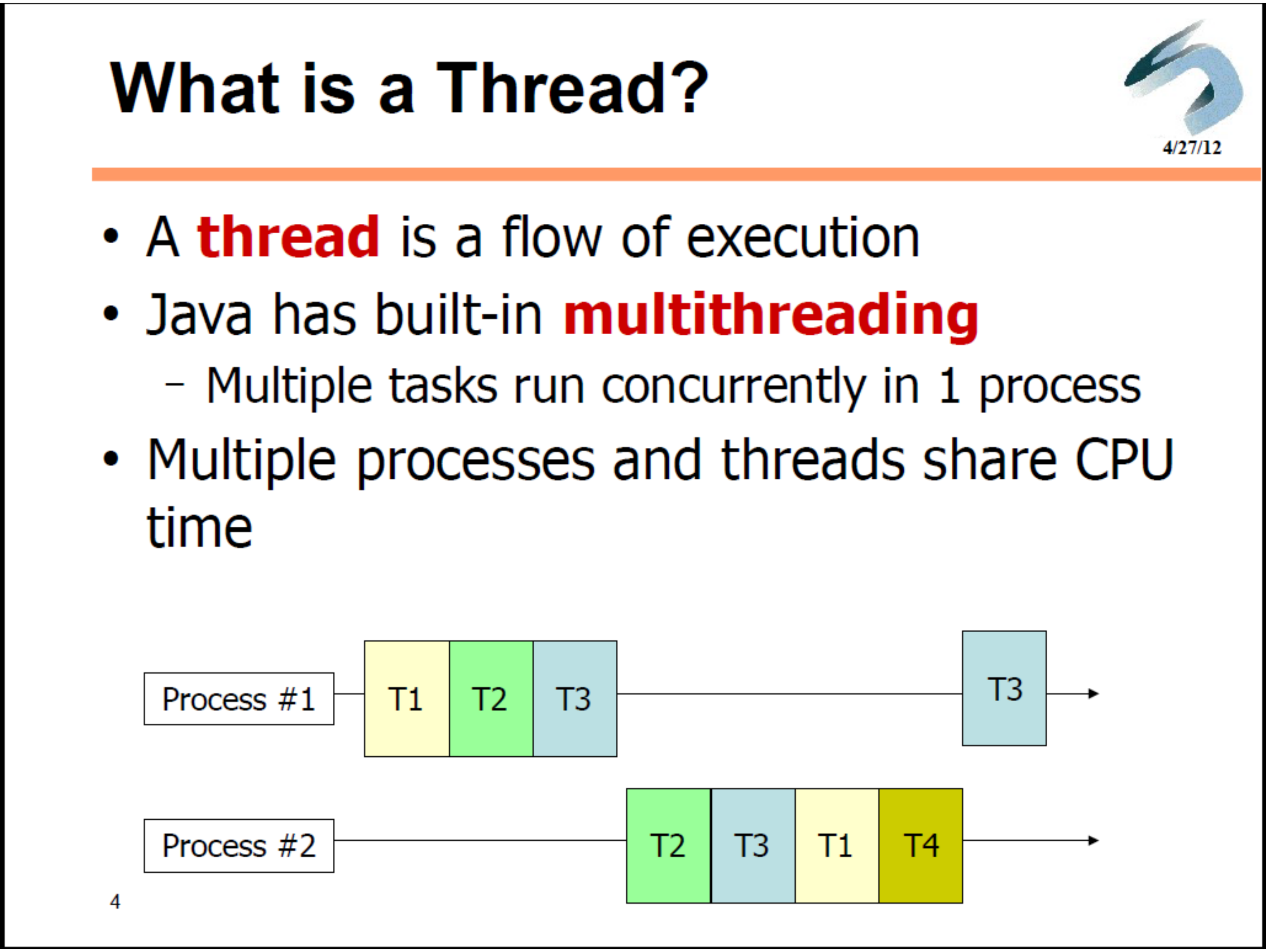
threads

3

Slide from Travis Brown, Rochester Tech

What is a Thread?

- Individual and separate unit of execution that is part of a process
  - multiple threads can work together to accomplish a common goal
- Video Game example
  - one thread for graphics
  - one thread for user interaction
  - one thread for networking



What is a Thread?

4/27/12

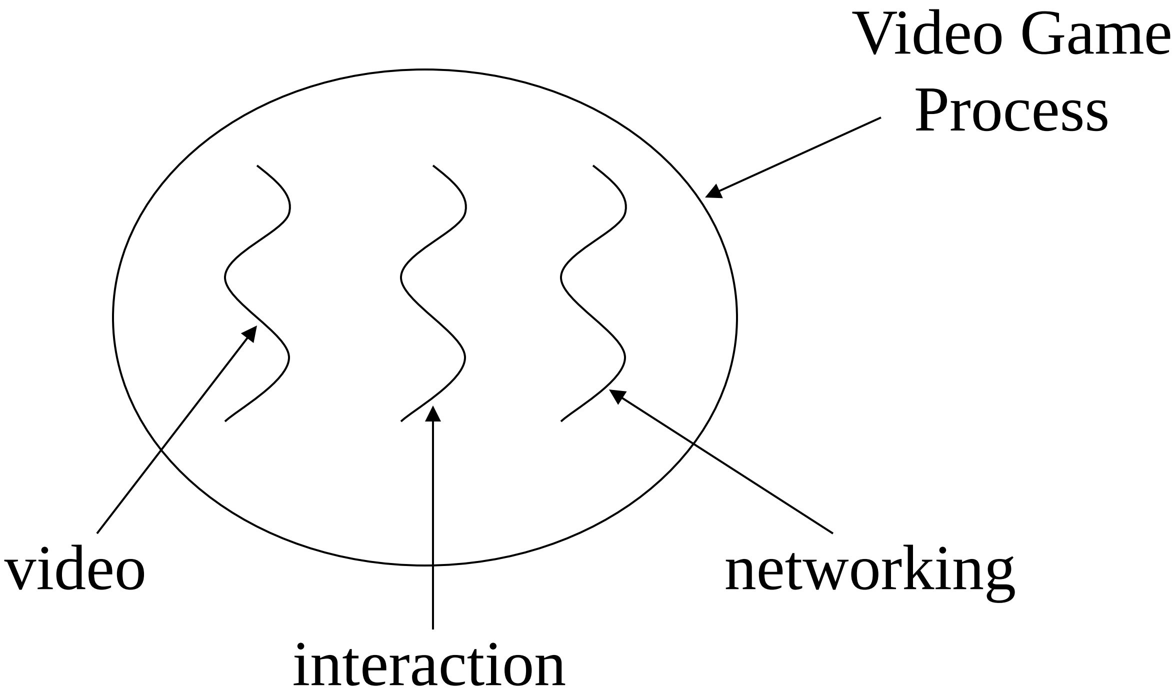
- A **thread** is a flow of execution
- Java has built-in **multithreading**
  - Multiple tasks run concurrently in 1 process
- Multiple processes and threads share CPU time

Process #1
T1
T2
T3
T3
Process #2
T2
T3
T1
T4

4

Slide from Travis Brown, Rochester Tech

What is a Thread?



Advantages

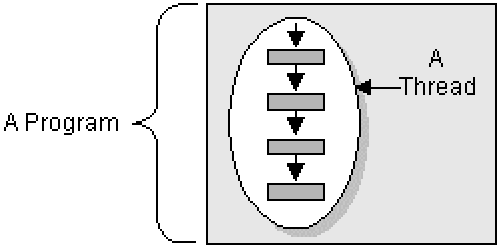
- easier to program
  - 1 thread per task
- can provide better performance
  - thread only runs when needed
  - no polling to decide what to do
- multiple threads can share resources
- utilize multiple processors if available

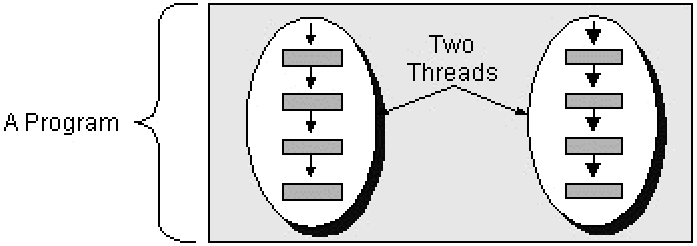
Disadvantages

- multiple threads can lead to deadlock
  - more on this later
- overhead of switching between threads

Threads

- Definition: Thread is a single Sequential Flow of Control within a program.
- Other Names: Thread = Execution Context = Lightweight Process
- Thread like a Sequential Program, has
  - A beginning, a sequence, and an end.
  - Has a single point of execution, at any given time





Slide from Moshe Fresko, Bar-Ilan

Multithreading

- Multithreading programs *appear* to do more than one thing at a time
- Same ideas as multiprocessing, but within a single program
- More efficient than multiprocessing
- Java tries to hide the underlying multiprocessing implementation

Threads

- A Thread is a single flow of control
    - When you step through a program, you are following a Thread
- Your previous programs all had one Thread
- In Java, a Thread is an Object you can create and control

Sleeping

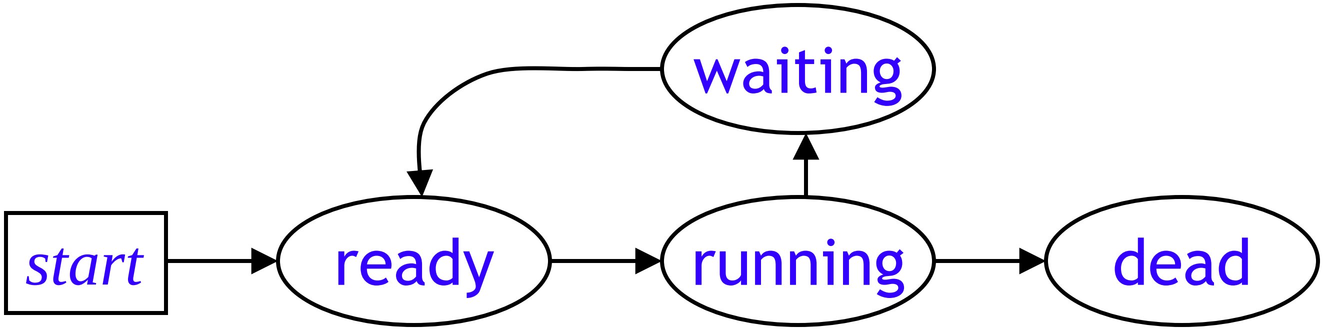
- Every program uses at least one Thread
- Thread.sleep(int ***milliseconds***);
  - A millisecond is 1/1000 of a second

-
  ```
  try { Thread.sleep(1000); }
  catch (InterruptedException e) { }
  ```
- sleep only works for the current Thread

States of a Thread

- A Thread can be in one of four states:
  - **Ready:** all set to run
  - **Running:** actually doing something
  - **Waiting,** or **blocked:** needs something
  - **Dead:** will never do anything again
- State names vary across textbooks
- You have some control, but the Java scheduler has more

State transitions



Two ways of creating Threads

- You can extend the Thread class:
  - class Animation extends Thread {…}
  - Limiting, since you can only extend one class
- Or you can implement the Runnable interface:
  - class Animation implements Runnable {…}
  - requires public void run( )
- The second is recommended for most programs

Extending Thread

-
  ```
  class Animation extends Thread {
      @Override
      public void run( ) { code for this thread }
      Anything else you want in this class
  }
  ```
- Animation anim = new Animation( );
  - A newly created Thread is in the **Ready** state
- To start the anim Thread running, call anim.start( );
- start( ) is a *request* to the scheduler to run the Thread --it may not happen right away
- The Thread should eventually enter the **Running** state

Implementing Runnable

- class Animation implements Runnable {…}
- The Runnable interface requires run( )
  - This is the "main" method of your new Thread
- Animation anim = new Animation( );
- Thread myThread = new Thread(anim);
- To start the Thread running, call myThread.start( );
  - You do not write the start() method—it's provided by Java
- As always, start( ) is a *request* to the scheduler to run the Thread--it may not happen right away

Starting a Thread

- Every Thread has a start( ) method
- *Do not* write or override start( )
- You *call* start( ) to request a Thread to run
- The scheduler then (eventually) calls run( )
- You must supply public void run( )
  - This is where you put the code that the Thread is going to run

Extending Thread: summary

```
class Animation extends Thread {
    public void run( ) {
        while (okToRun) { ... }
    }
}

Animation anim = new Animation( );
anim.start( );
```

Implementing Runnable: summary

```
class Animation extends Applet
                implements Runnable {
    public void run( ) {
        while (okToRun) { ... }
    }
}

Animation anim = new Animation( );
Thread myThread = new Thread(anim);
myThread.start( );
```

Things a Thread can do

- Thread.sleep(milliseconds)
- yield( )
- Thread me = currentThread( );
- int myPriority = me.getPriority( );
- me.setPriority(NORM_PRIORITY);
- if (otherThread.isAlive( )) { … }
- join(otherThread);

Animation requires two Threads

- Suppose you set up Buttons and attach Listeners to those buttons...
- …then your code goes into a loop doing the animation…
- …who's listening?
  - Not this code; it's busy doing the animation
- sleep(*ms*) doesn't help!

How to animate

- Create your buttons and attach listeners in your first (original) Thread
- Create a second Thread to run the animation
- Start the animation
- The original Thread is free to listen to the buttons

- *However,*
  - Whenever you have a GUI, Java *automatically* creates a second Thread for you
  - You only have to do this yourself for more complex programs

Things a Thread should NOT do

- The Thread controls its own destiny
- Deprecated methods:
  - myThread.stop( )
  - myThread.suspend( )
  - myThread.resume( )
- Outside control turned out to be a Bad Idea
- Don’t do this!

How to control another Thread

- Don’t use the deprecated methods!
- Instead, put a request where the other Thread can find it

-
  ```
  boolean okToRun = true;
  animation.start( );
  ```

-
  ```
  public void run( ) {
      while (controller.okToRun) {…}
  ```

A problem

```
int k = 0;
```

Thread #1:

```
k = k + 1;
```

Thread #2:

```
System.out.print(k);
```

- What gets printed as the value of k?
- This is a trivial example of what is, in general, a very difficult problem

Tools for a solution

- You can synchronize on an object:
  - synchronized (***obj***) { ***…code that uses/modifies obj…*** }
  - No other code can use or modify this object at the same time
  - Notice that synchronized is being used as a *statement*
- You can synchronize a method (uses this):
  - synchronized void addOne(***arg1***, ***arg2***, ***…***) { ***code*** }
  - Only one synchronized method in a class can be used at a time (other methods can be used simultaneously)
- Synchronization is a *tool*, not a solution—multithreading is in general a very hard problem

The synchronized statement

- Synchronization is a way of providing exclusive access to data
- You can synchronize on any Object, of any type
- If two Threads try to execute code that is synchronized on the **same** object, only one of them can execute at a time; the other has to wait
  - synchronized (someObject) { /* some code */ }
  - This works whether the two Threads try to execute the same block of code, or different blocks of code that synchronize on the same object
- Often, the object you synchronize on bears some relationship to the data you wish to manipulate, but this is not at all necessary

synchronized methods

- Instance methods can be synchronized:

  -
    ```
    synchronized public void myMethod( /* arguments */) {
        /* some statements */
    }
    ```
- This is equivalent to

  -
    ```
    public void myMethod( /* arguments */) {
        synchronized(this) {
            /* some statements */
        }
    }
    ```
- Static methods can also be synchronized
  - They are synchronized on the class object (a built-in object that represents the class)

Summary

- Event loops and listeners
- Processes vs threads
- Threads in Java
- Need for syncronization
  - "thread safety"

Literature

- Java Concurrency Tutorial

  http://docs.oracle.com/javase/tutorial/essential/concurrency/

- Ullenboom, Ch.

  Java ist auch eine Insel (Chapter 14)

  Galileo Computing, 2012