Threads & Timers

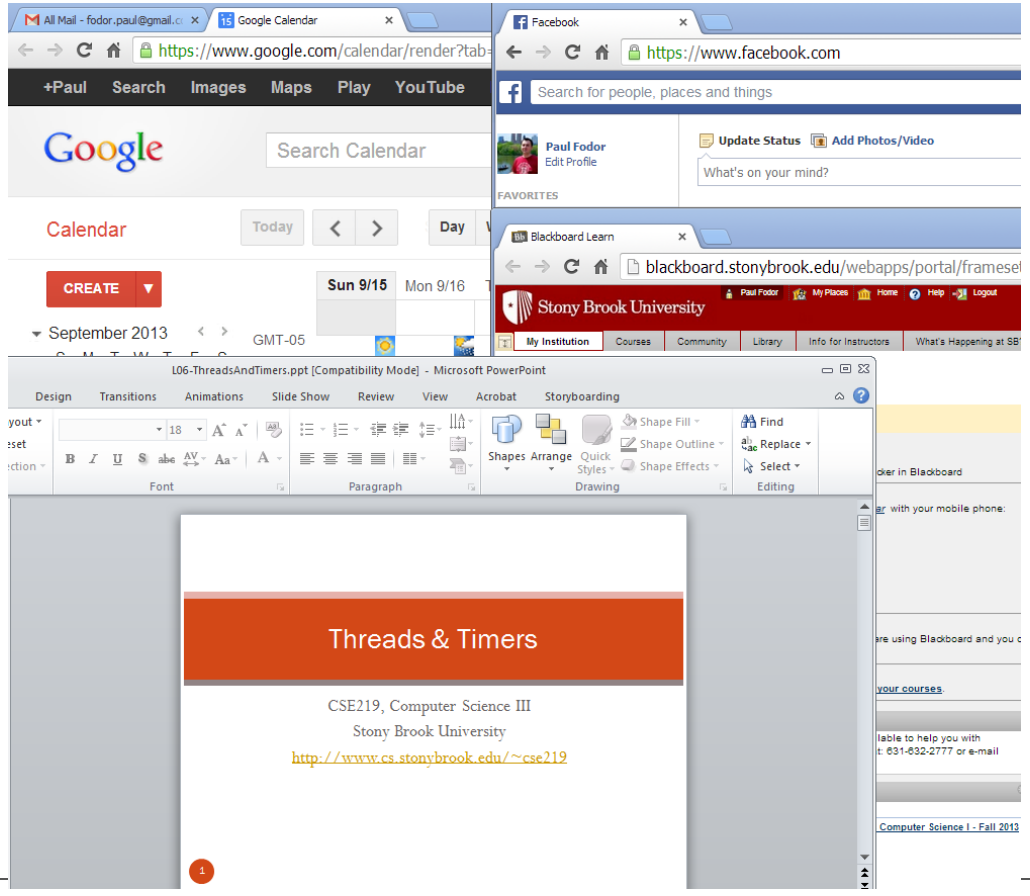
CSE219, Computer Science III

Stony Brook University

http://www.cs.stonybrook.edu/~cse219

Multi-tasking

- When you're working, how many different applications do you have open at one time? Many! ~100 even if you have only a few visible.



Multithreaded?

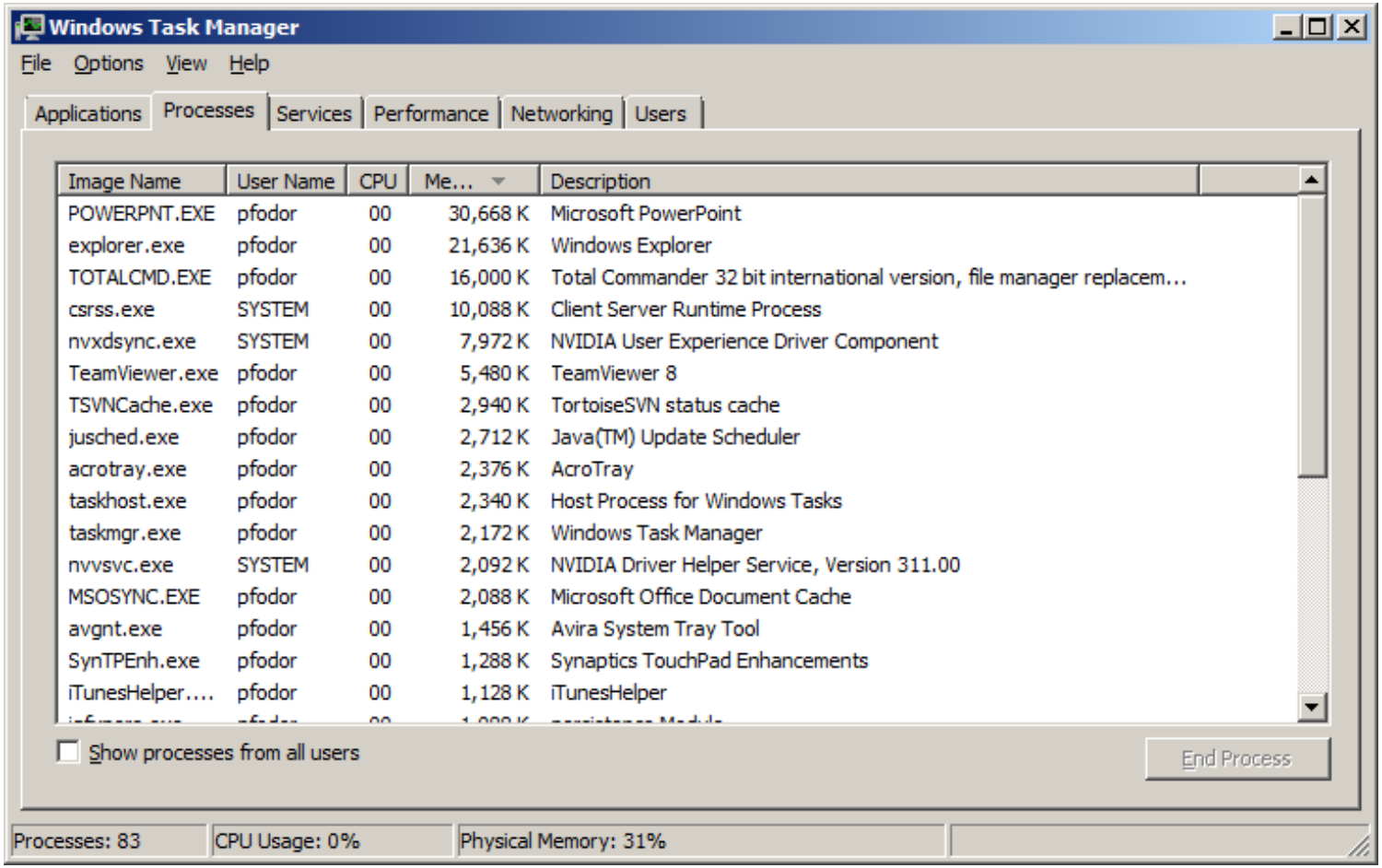
- When you request a Web page. Should the IE client:
  - wait for the page before doing anything else

OR

  - do other work while waiting
    - like responding to user input/rendering

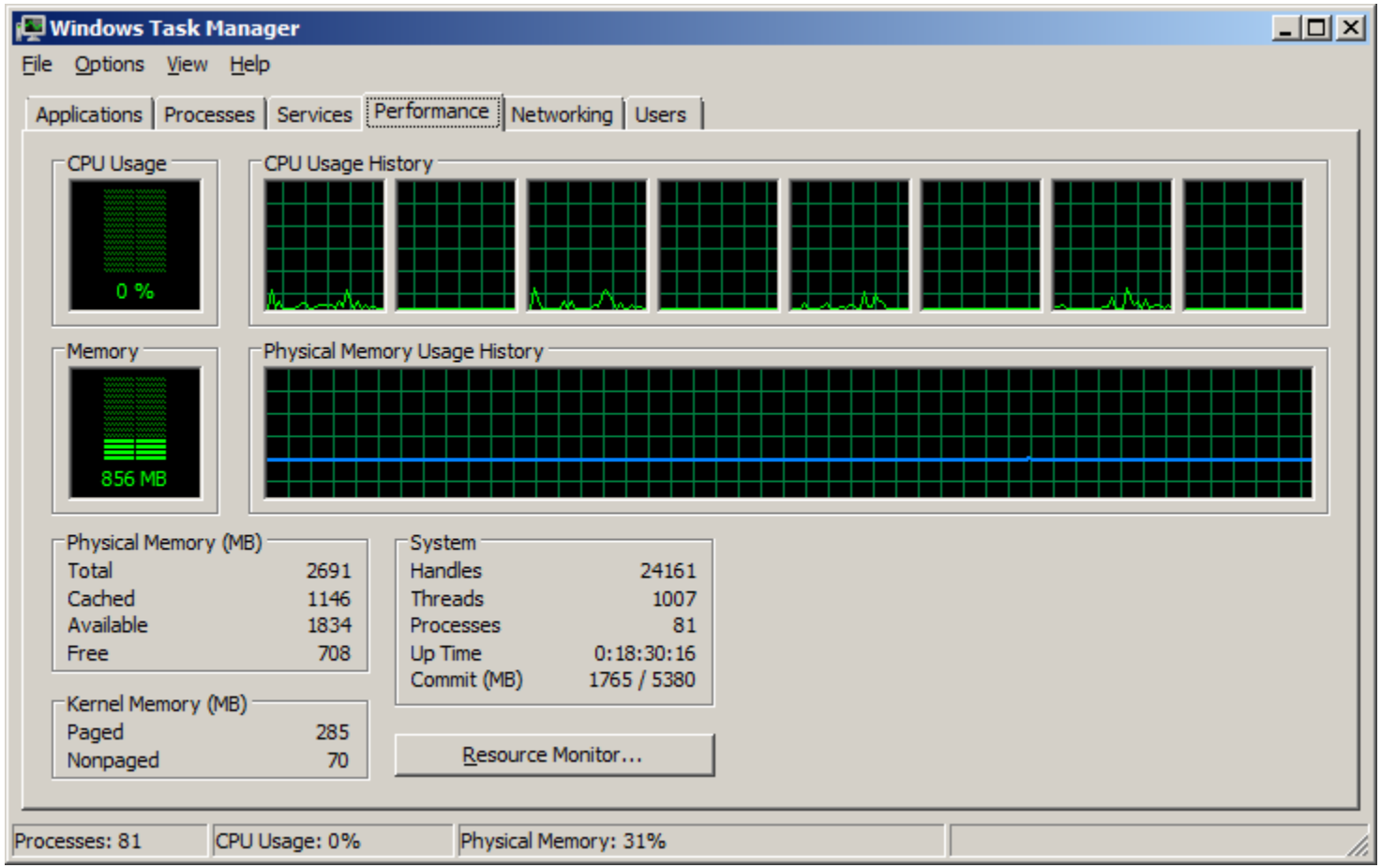
OS Multi-tasking

- How many tasks is the OS performing?
  - Press CTRL+Shift+ESC



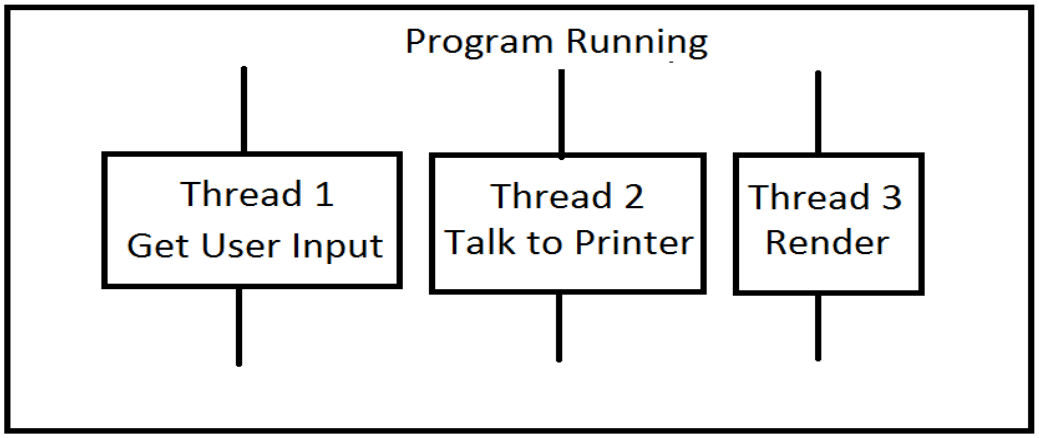
OS Multi-tasking

- How many CPUs does your PC have?



Program Multi-Tasking

- Most apps need to do multiple tasks "simultaneously"
- For example:
  - getting user input
  - printing
  - Internet browsing



- How would you do this?
  - using threads (that you define)

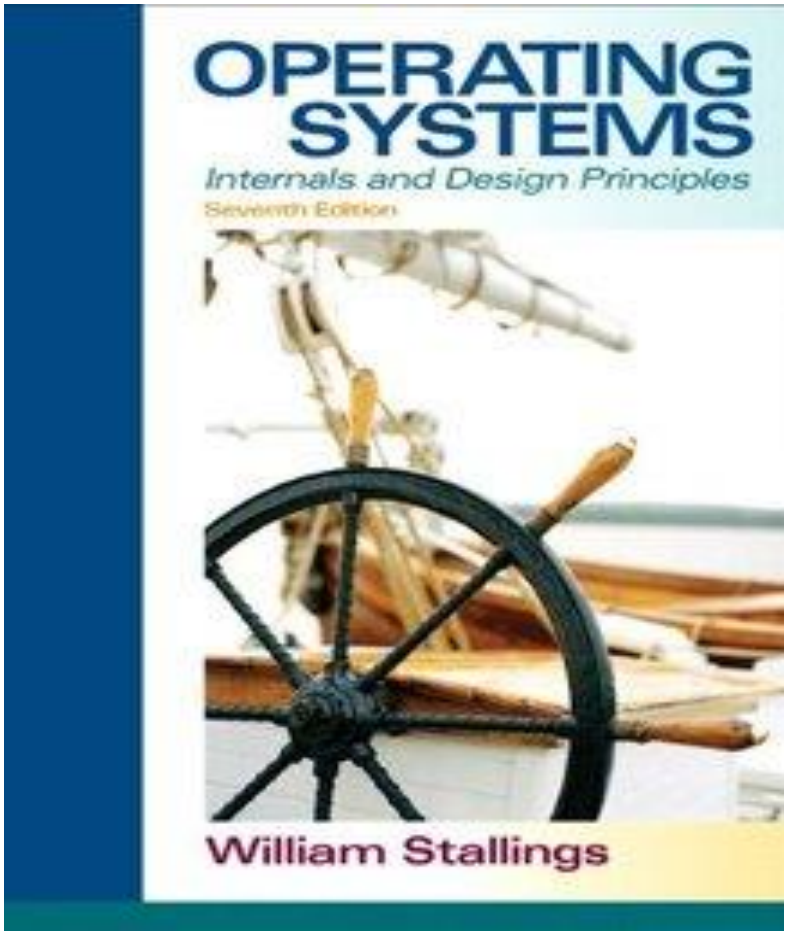
  AND
  - using a thread scheduler (that the JVM provides)

Tools for OS Multi-tasking

- Thread scheduling
- Time-sharing
- Virtual Memory
- Operating Systems

topics covered in:

CSE 306 at Stony Brook U.



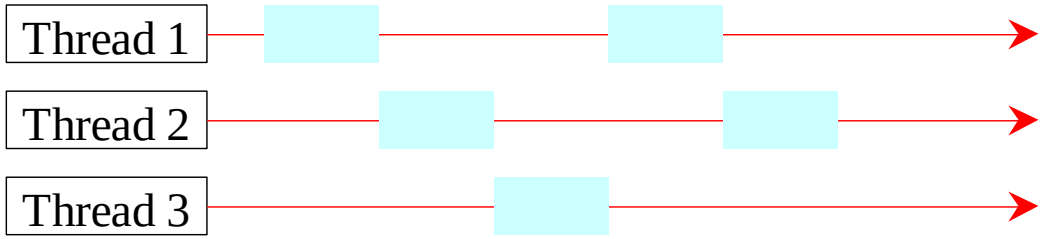
Multi-Core Complicates Everything

- Intel Xeon E7
  - 10+ Cores
  - 20+ Threads

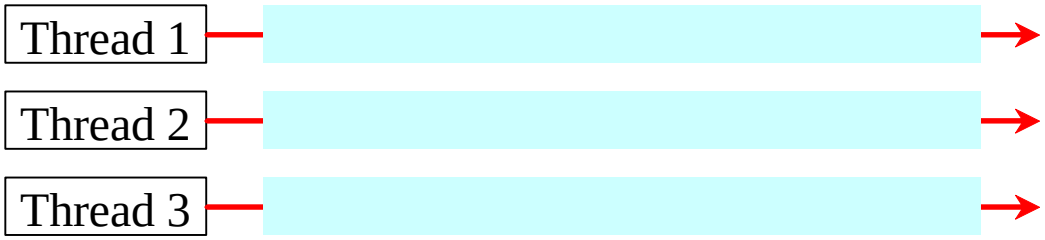
- let the OS work it out

Multi-Core Complicates Everything

Multiple threads sharing a single CPU



Multiple threads on multiple CPUs



Threads and the Thread Scheduler

- You define your own threads

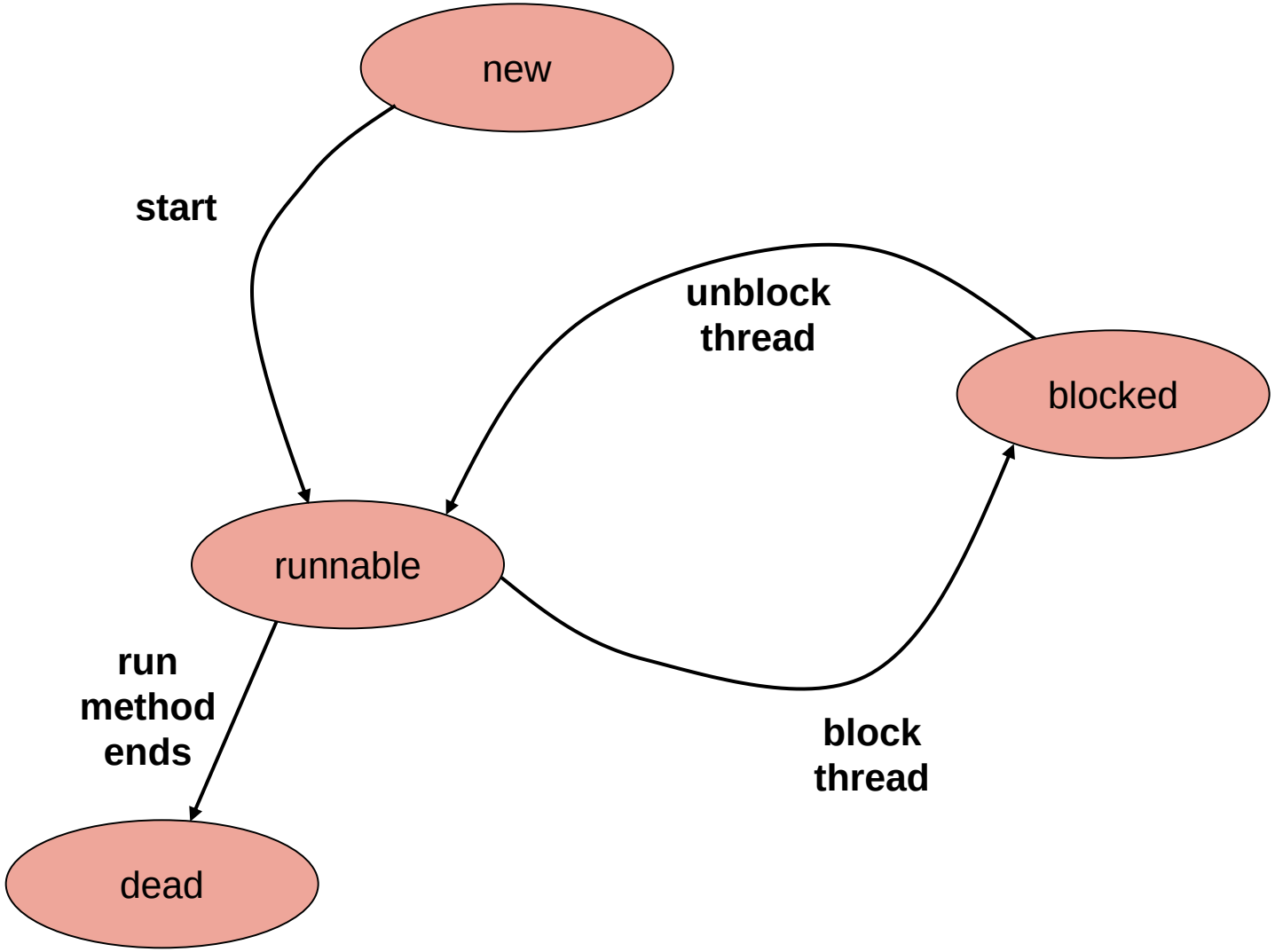
Extend **`java.lang.Thread`**

  - i.e. tasks
  - Note: main is its own thread

- You make your threads *runnable*
  - i.e. start them

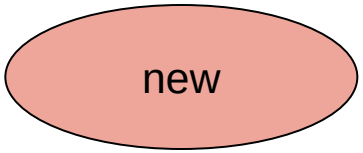
- Java's thread scheduler decides order!

State transitions of a thread



new state

- A constructed Thread object

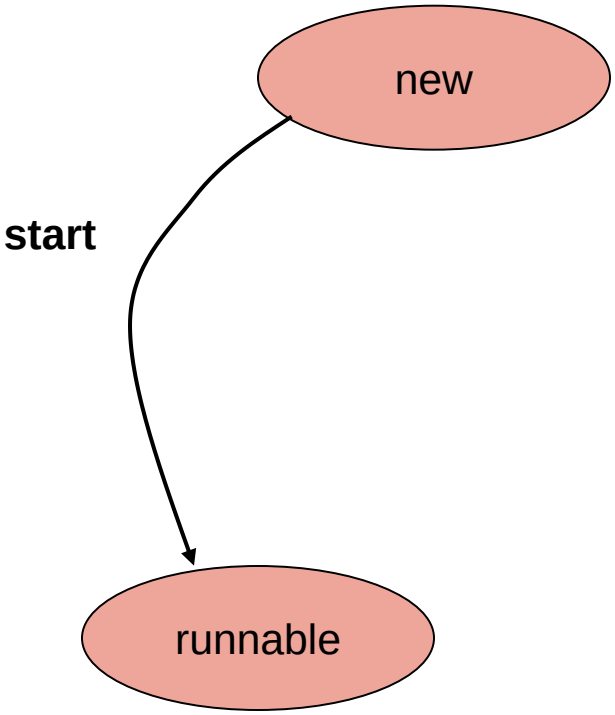


- Not yet started
- Not yet known to thread scheduler
- Not runnable

New – to – Runnable Transition

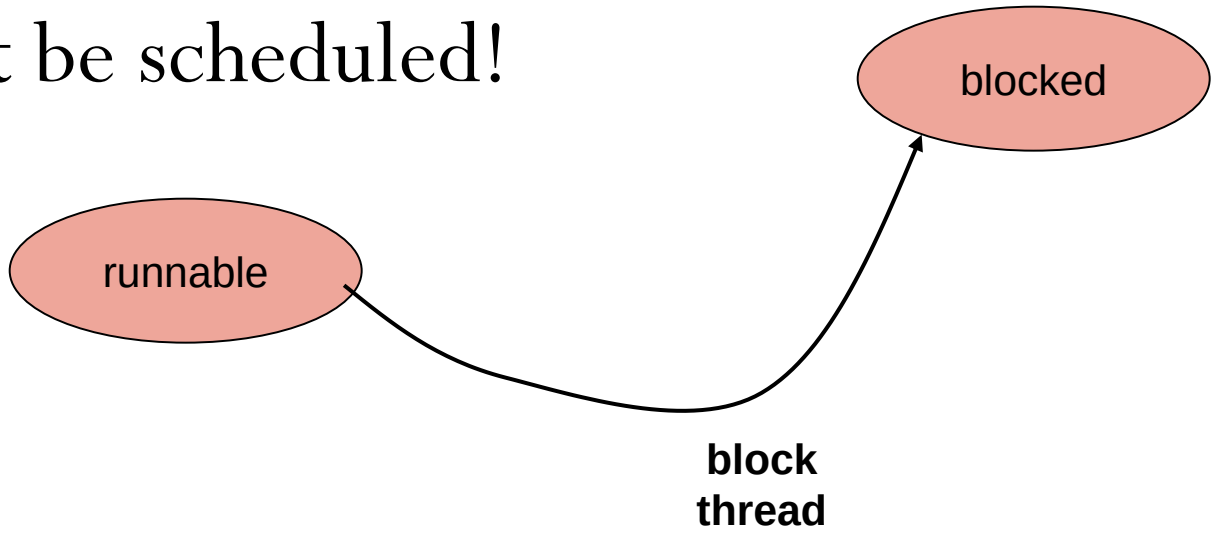
- Constructed thread is started
  - call *start* method on it
- Can be scheduled!

- There may be many threads in this state.



Runnable – to – Blocked Transition

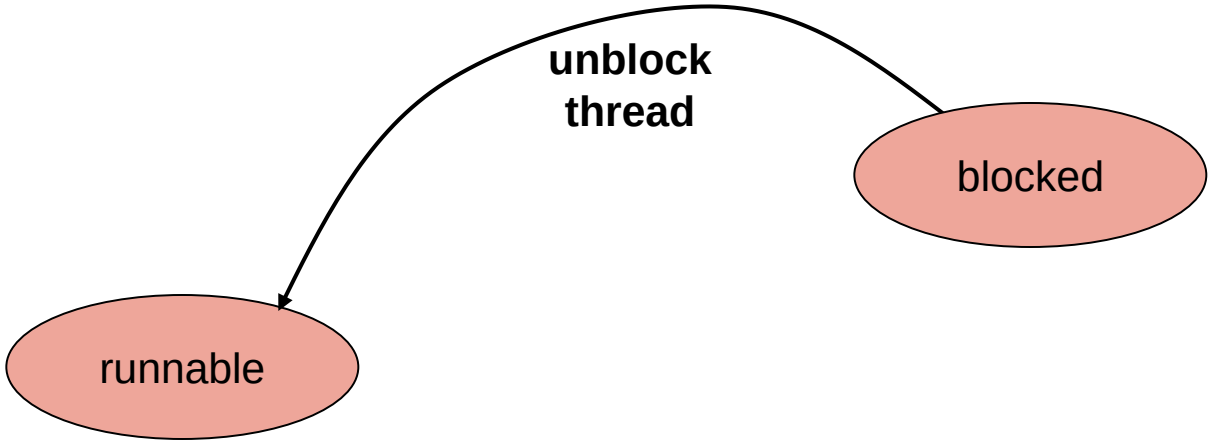
- Runnable thread made unrunnable
  - call *sleep* method on it (for X milliseconds)
    - directly or via *lock* method
- Can **not** be scheduled!



- Again, there may be many threads in this state

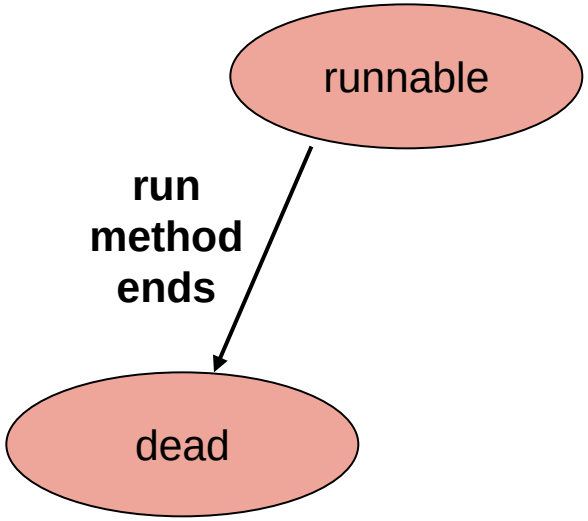
Blocked – to – Runnable Transition

- Unrunnable thread made runnable
  - sleep time expires
    - and is not renewed
      - *unlock* method ensures this
- Can be scheduled



Runnable – to – Dead Transition

- Run method completes
- Cannot be rescheduled
- A dead thread is Dead
- Call `isAlive` to take a pulse



Defining your own threads

```
public class MyThread extends Thread {
        ...
        public void run() {
                // task to do when
                // the thread is started
        }
}
```

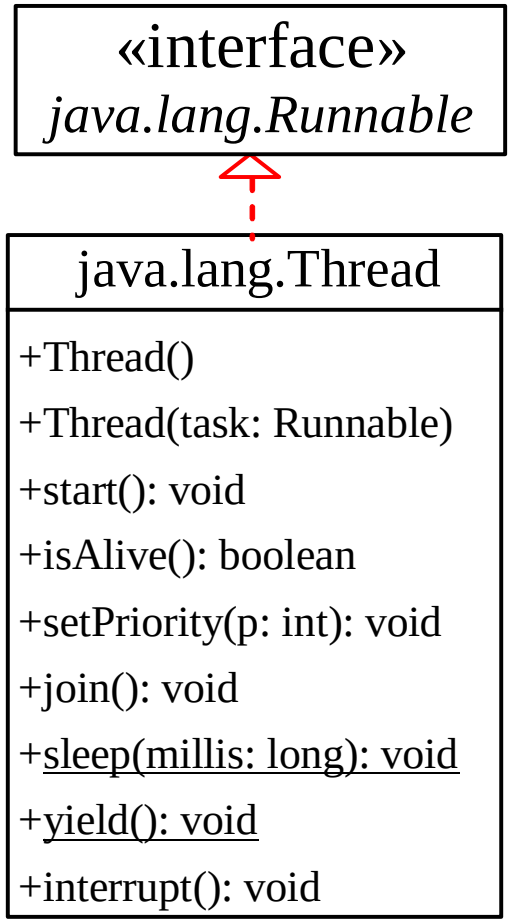
- Create a new thread:

```
MyThread mT = new MyThread();
```

- Run the thread:

```
mT.start();
```

The Thread Class



Creates a default thread.

Creates a thread for a specified task.

Starts the thread that causes the run() method to be invoked by the JVM.

Tests whether the thread is currently running.

Sets priority p (ranging from 1 to 10) for this thread.

Waits for this thread to finish.

Puts the runnable object to sleep for a specified time in milliseconds.

Causes this thread to temporarily pause and allow other threads to execute.

Interrupts this thread.

The 2 key Thread methods

- *start()*
  - makes thread runnable
  - calls the *run* method
  - Thread class' start method already does this
    - if your class that extends *Thread* **you don't have to define** *start*
- *run()*
  - executed when a thread is started (with the method `start()`)
  - `run()` is where thread work is done
  - The `Thread` superclass' run method does nothing
    - if your class extends Thread **you must define** *run()*
      - to specify what work your thread will do

`run()`

Method Summary	
void	run() When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.

- **`run()`** may do one thing or many
  - via iteration
  - it may even exist for the duration of the program

start vs. run

- The **main** method has a thread
- We write:

```
public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
        …
}
```

  - Now we have 2 threads: main and t.
- What about:

```
public static void main(String[] args) {
        MyThread t = new MyThread();
        t.run();
        …
}
```

- Still just 1 thread: t.run() is just a method call!

start vs. run

```
public class RandomThread extends Thread {
    public void run() {
        while (true) {
            int num = (int) (Math.random() * 10);
            System.out.println("\t\t\t\t" + num);
            try { Thread.sleep(10);
            } catch(InterruptedException ie) {}
        }
    }
}
```

/* An InterruptedException is thrown when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted, either before or during the activity. Occasionally a method may wish to test whether the current thread has been interrupted, and if so, to immediately throw this exception. E.g.,

```
  if (Thread.interrupted())
       throw new InterruptedException();
         // Clears interrupted status!
*/
```

start vs. run

```
import java.util.Calendar;
import java.util.GregorianCalendar;
public class StartTester {
    public static void main(String[] args){
        RandomThread thread = new RandomThread();
        thread.start();
        while (true) {
            Calendar today = new GregorianCalendar();
            long hour = today.get(Calendar.HOUR);
            long minute = today.get(Calendar.MINUTE);
            long second = today.get(Calendar.SECOND);
            System.out.println(hour + ":"
                    + minute + ":" + second);
            try { Thread.sleep(10);
            } catch(InterruptedException ie) {}
        }
    }
}
```

THIS *IS* A MULTITHREADED APPLICATION!

start vs. run

```
import java.util.Calendar;
import java.util.GregorianCalendar;
public class RunTester {
    public static void main(String[] args) {
        RandomThread thread = new RandomThread();
        thread.run(); // Only this main thread is running
        while (true) {
            Calendar today = new GregorianCalendar();
            long hour = today.get(Calendar.HOUR);
            long minute = today.get(Calendar.MINUTE);
            long second = today.get(Calendar.SECOND);
            System.out.println(hour + ":"
                    + minute + ":" + second);
            try {
                Thread.sleep(10);
            } catch (InterruptedException ie) {
            }
        }
    }
}
```

Creating Tasks and Threads


```
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

Runnable interface

- The `Runnable` interface has 1 method: `run()`
- Alternative threading approach:
  - use `implements Runnable`

  AND
  - define `run()`

Using the `Runnable` Interface to Create and Launch Threads

- Create and run three threads:
  - The first thread prints the letter *a* 100 times.
  - The second thread prints the letter *b* 100 times.
  - The third thread prints the integers 1 through 100.

TaskThreadDemo.java

```
public class TaskThreadDemo {
    public static void main(String[] args) {
        // Create tasks
        Runnable printA = new PrintChar('a', 100);
        Runnable printB = new PrintChar('b', 100);
        Runnable print100 = new PrintNum(100);
        // Create threads
        Thread thread1 = new Thread(printA);
        Thread thread2 = new Thread(printB);
        Thread thread3 = new Thread(print100);
        // Start threads
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

TaskThreadDemo.java

```
// The task for printing a specified character in specified times
class PrintChar implements Runnable {
    private char charToPrint; // The character to print
    private int times; // The times to repeat
    /**
     * Construct a task with specified character and number of times to print
     * the character
     */

    public PrintChar(char c, int t) {
        charToPrint = c;
        times = t;
    }

    /**
     * Override the run() method to tell the system what the task to perform
     */
    public void run() {
        for (int i = 0; i < times; i++) {
            System.out.print(charToPrint);
        }
    }
}
```

TaskThreadDemo.java

```
// The task class for printing number from 1 to n for a given n
class PrintNum implements Runnable {

    private int lastNum;

    /**
     * Construct a task for printing 1, 2, ... i
     */
    public PrintNum(int n) {
        lastNum = n;
    }

    /**
     * Tell the thread how to run
     */
    public void run() {
        for (int i = 1; i <= lastNum; i++) {
            System.out.print(" " + i);
        }
    }
}
```

The Static yield() Method

You can use the yield() method to temporarily release time for other threads.

```
public void run() {
  for (int i = 1; i <= lastNum; i++){
    System.out.print(" " + i);
    Thread.yield();
  }
}
```

Every time a number is printed, the print100 thread is yielded. So, the numbers are printed after the characters.

The Static sleep(milliseconds) Method

The sleep(long mills) method puts the thread to sleep for the specified time in milliseconds.

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    try {
      if (i >= 50) Thread.sleep(1);
    }
    catch (InterruptedException ex) {
    }
  }
}
```

Every time a number (>= 50) is printed, the print100 thread is put to sleep for 1 millisecond.

isAlive(), interrupt(), and isInterrupted()

- The isAlive() method is used to find out the state of a thread.
  - It returns true if a thread is in the Ready, Blocked, or Running state;
  - it returns false if a thread is new and has not started or if it is finished.
- The interrupt() method interrupts a thread in the following way: If a thread is currently in the Ready or Running state, its interrupted flag is set; if a thread is currently blocked, it is awakened and enters the Ready state, and an java.io.InterruptedException is thrown.
- The isInterrupt() method tests whether the thread is interrupted.

Thread Priority

- Each thread is assigned a default priority of Thread.NORM_PRIORITY.
- You can reset the priority using setPriority(int priority).
- Some constants for priorities include
  Thread.MIN_PRIORITY
  Thread.MAX_PRIORITY
  Thread.NORM_PRIORITY

GUIs and Threads

- What if we want to make our frame multi-threaded?
  - implement Runnable
- GUI event handling and painting code executes in a single thread, called the *event dispatcher thread*.
- This ensures that each event handler finishes executing before the next one executes and the painting isn't interrupted by events.

GUIs and Threads

Platform.runLater(): If you need to update a GUI component from a non-GUI thread, you can use that to put your update in a queue and it will be handle by the GUI thread as soon as possible.

FlashText.java

```
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
public class FlashText extends Application {
    private String text = "";
    @Override
    public void start(Stage primaryStage) {
        StackPane pane = new StackPane();
        Label lblText = new Label("Programming is fun");
        pane.getChildren().add(lblText);
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    while (true) {
                        if (lblText.getText().trim().length() == 0) {
                            text = "Welcome";
                        } else {
                            text = "";
                        }
```

```
                    Platform.runLater(new Runnable() {
                        @Override
                        public void run() {
                            lblText.setText(text);
                        }
                    });
                    Thread.sleep(200);
                }
            } catch (InterruptedException ex) {
            }
        } }).start();
    Scene scene = new Scene(pane, 200, 50);
    primaryStage.setTitle("FlashText");
    primaryStage.setScene(scene);
    primaryStage.show();
}
public static void main(String[] args) {
    launch(args); } }
```

FlashTextUsingLambda.java

```
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
public class FlashTextUsingLambda extends Application {
    private String text = "";
    @Override
    public void start(Stage primaryStage) {
        StackPane pane = new StackPane();
        Label lblText = new Label("Programming is fun");
        pane.getChildren().add(lblText);
        new Thread(() -> {
            try {
                while (true) {
                    if (lblText.getText().trim().length() == 0) {
                        text = "Welcome";
                    } else {
                        text = "";
                    }
```

```
                    Platform.runLater(() -> lblText.setText(text));
                    Thread.sleep(200);
                }
            } catch (InterruptedException ex) {
            }
        }).start();
        Scene scene = new Scene(pane, 200, 50);
        primaryStage.setTitle("FlashText");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Killing a thread

- Threads usually perform actions repeatedly
- What if you want to tell a thread to stop doing what it's doing?
  - This takes cooperation between threads
- Do not use the **`stop`** method --- it's deprecated:
  - It kills threads immediately
  - **A thread's run method may be mid-algorithm when killed**
- Preferred option: ask thread to kill itself. How?
  - via your own instance variable
    - make it a loop control for run
    - lets the thread set its affairs in order before dying

Typical run structure

```
public class NiceThread extends Thread {
    private boolean die = false;
    public void askToDie() {
        die = true;
    }
    public void run() {
        while (!die) {
            // do work here
            try {
                sleep(1000);
            } catch (InterruptedException ie) {
            }
        }
        // set affairs in order: DEAD IS DEAD
    }
    public static void main(String[] args){
        NiceThread t = new NiceThread();
        t.start();
        t.askToDie();
    }
}
```

Timer Threads

- Common Problem:
  - Need program to do something X times/second
- Like what?
  - count time
  - display time
  - update and render scene
- 2 Java Options:
  - have your thread do the counting
  - have a Java `java.util.Timer` instance do the counting

Java Timers

- Execute **`TimerTasks`** on schedule
  - via its own hidden thread
- What do we do?
  - define our own **`TimerTask`**
  - put work in **`run()`** method
  - construct our task
  - construct a timer
  - schedule task on timer
- **`cancel`** method unschedules our task (i.e. kills it)

```
import java.util.Timer;
import java.util.TimerTask;
public class TimerDemo {
    int i = 0;
    class MyTimerTask extends TimerTask {
        public void run() {
            System.out.println("Test " + (++i));
        }
    }
    public TimerDemo() {
        Timer timer = new Timer();
        timer.schedule(new MyTimerTask(), 0, 100);
        System.out.println("TimerTask scheduled.");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            System.out.println("got interrupted!");
        }
        timer.cancel();
        System.out.println("TimerTask finished.");
    }
    public static void main(String args[]) {
        TimerDemo td = new TimerDemo();
    }
}
```

Run:
TimerTask scheduled.
Test 1
Test 2
Test 3
Test 4
Test 5
…
Test 49
Test 50
TimerTask finished.