CSE 219
COMPUTER SCIENCE III

THREADS AND TIMERS

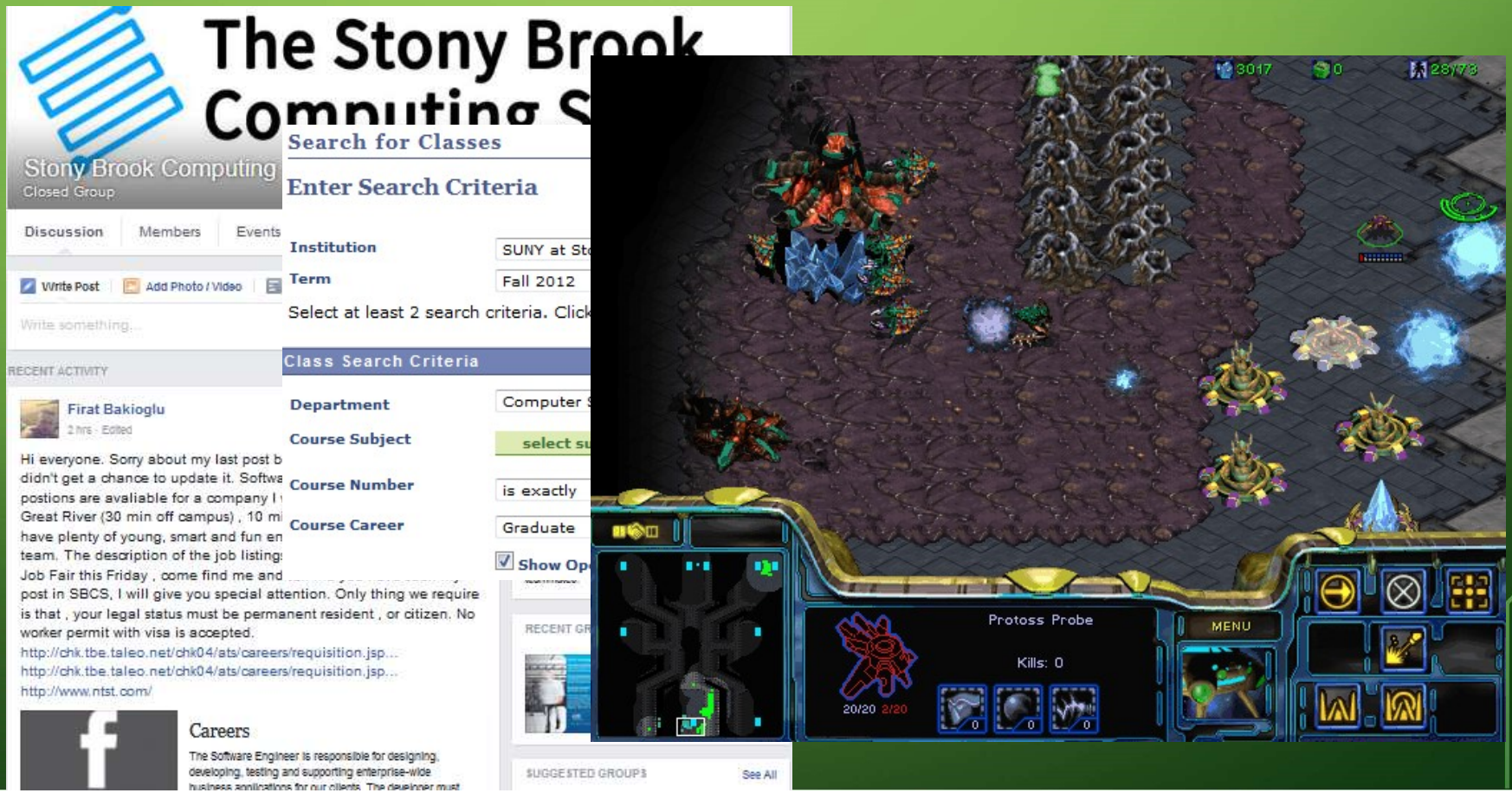
SLIDES COURTESY:

RICHARD MCKENNA

STONY BROOK UNIVERSITY

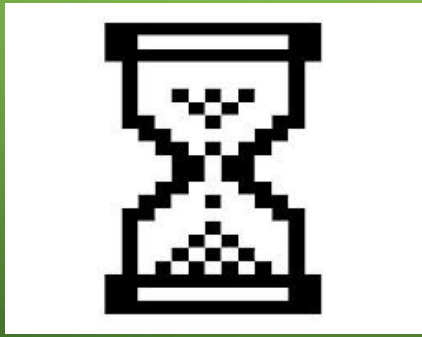
Multi-tasking

- When you're working, how many different applications do you have open at one time?



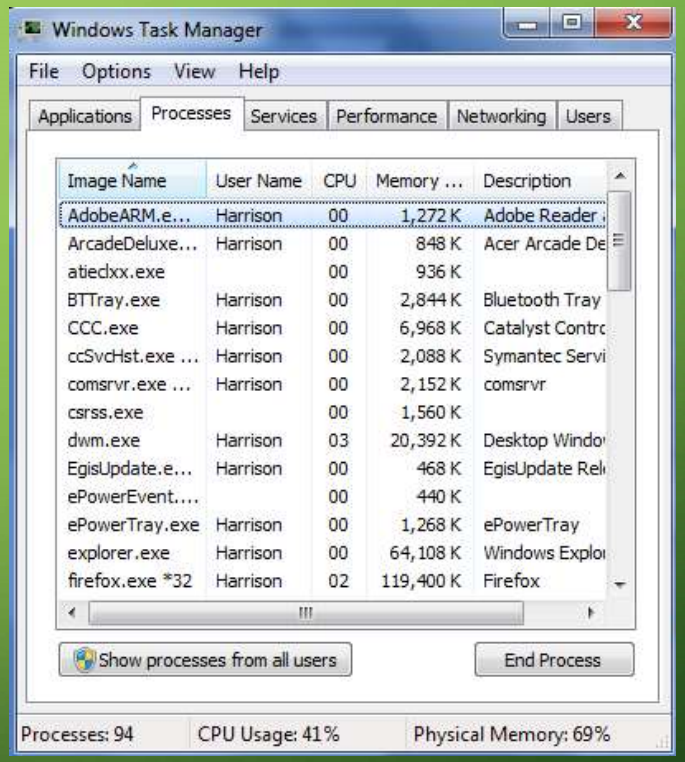
Multithreaded?

- When you request a Web page. Should IE:
  - wait for the page before doing anything else
    - OR
  - do other work while waiting
    - like responding to user input/rendering



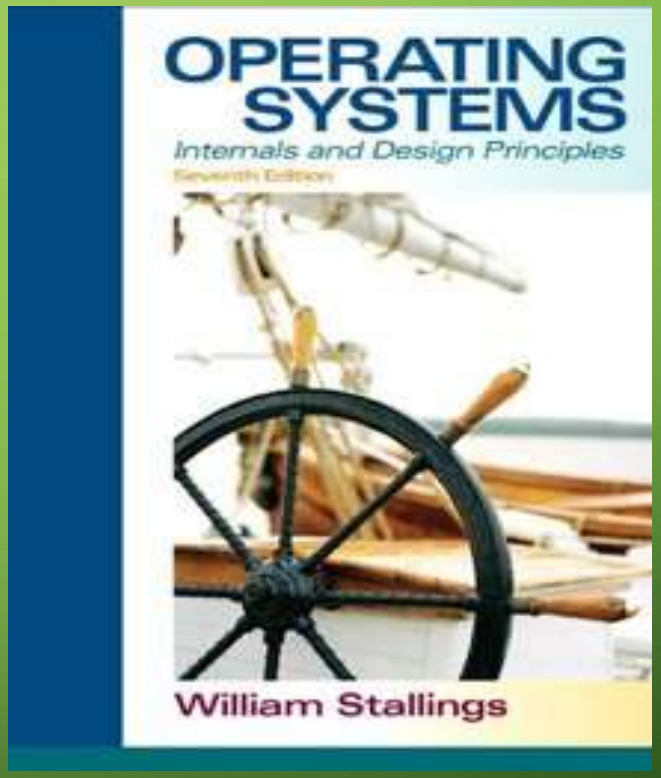
OS Multi-tasking

- How many tasks is the OS performing?
- How many CPUs does your PC have?



Tools for OS Multi-tasking

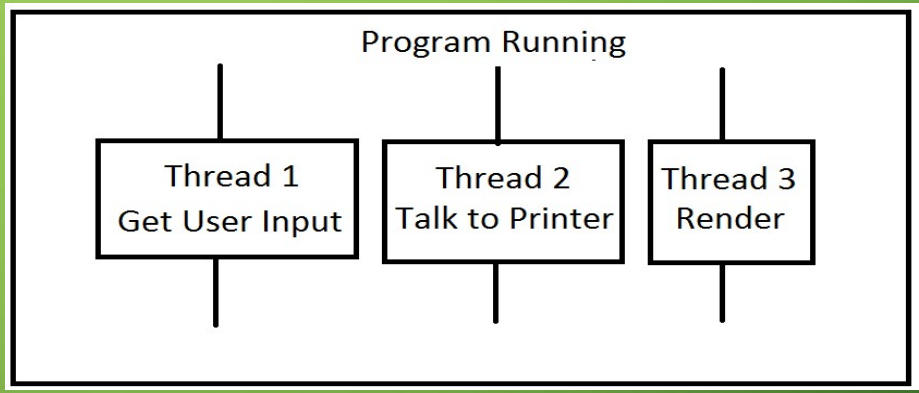
- Threads
- Thread scheduling
- Time-sharing
- Virtual Memory



Program Multi-Tasking

- Most apps need to do multiple tasks "simultaneously"

- For example:
  - getting user input
  - printing
  - Internet



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

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

Multi-Core Complicates Everything

- Intel Xeon E7
  - 10 Cores
  - 20 Threads

- We'll ignore multicore
  - let the OS work it out

- We'll assume a single core

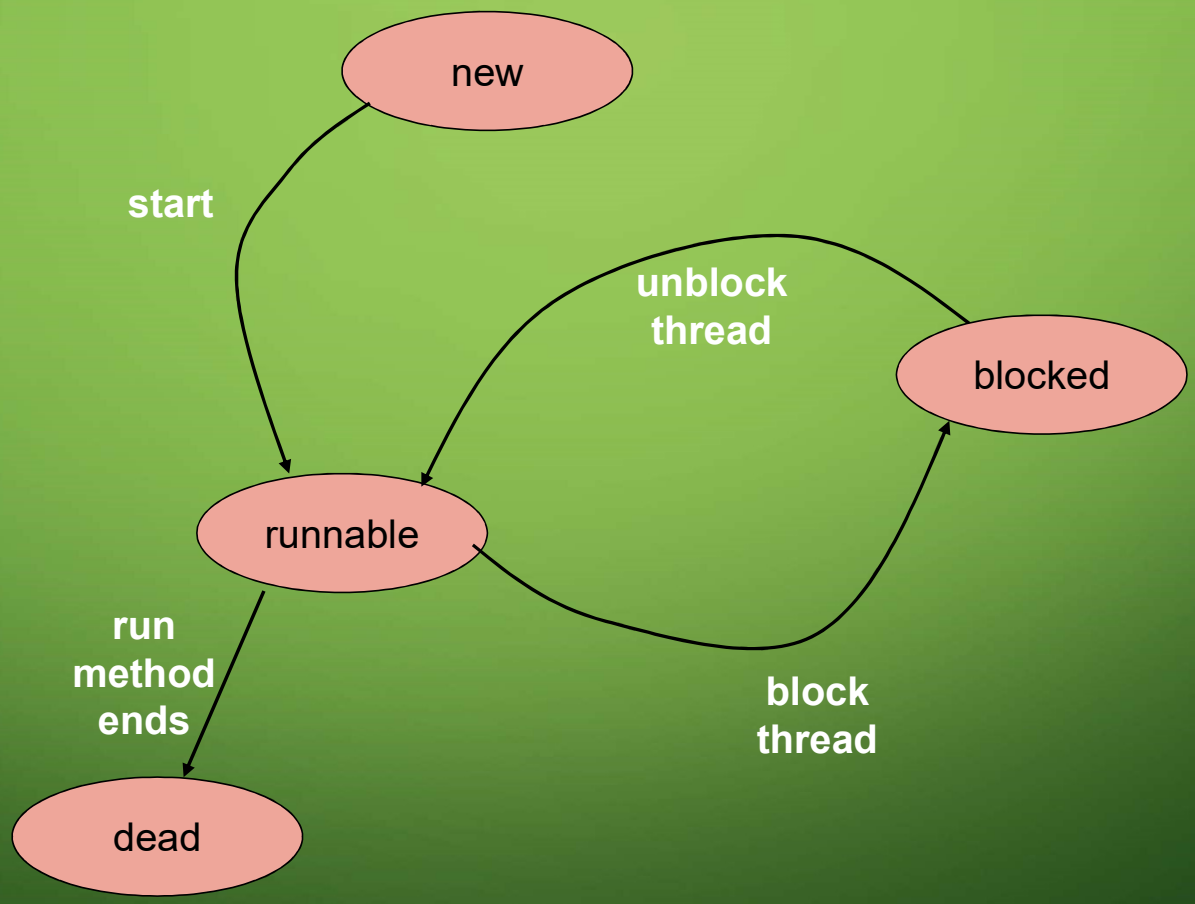
Threads and the Thread Scheduler

- You define your own threads
  - i.e. tasks
  - Note: **`main`** is its own thread

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

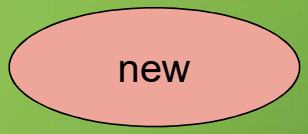
- Java's thread scheduler decides order
- What order should it use?

State transitions of a thread

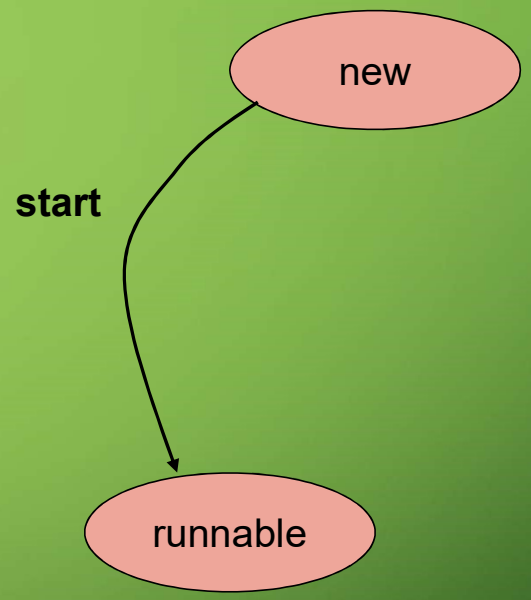


new state

- A constructed Thread object
- Not yet started
- Not yet known to thread scheduler
- Not runnable



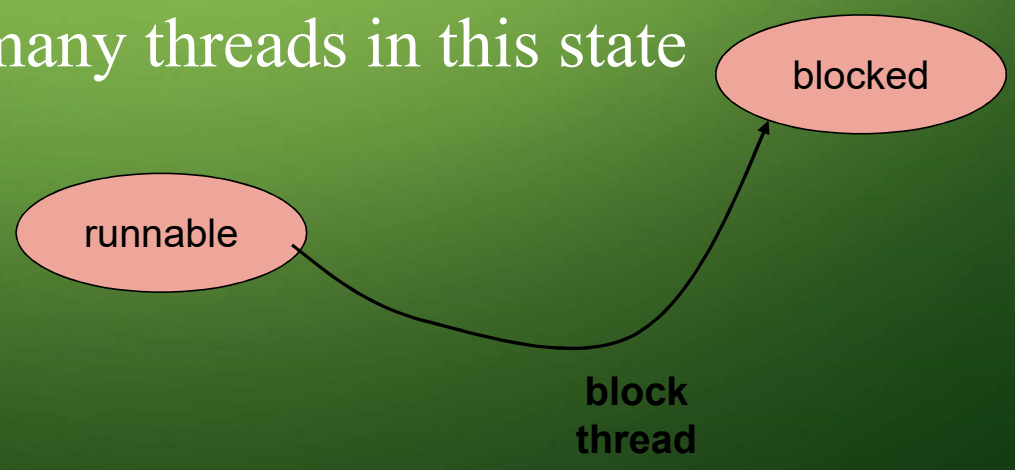
New – to – Runnable Transition



- Constructed thread is started
  - call ***start*** method on it
- Can be scheduled
- There may be many threads in this state

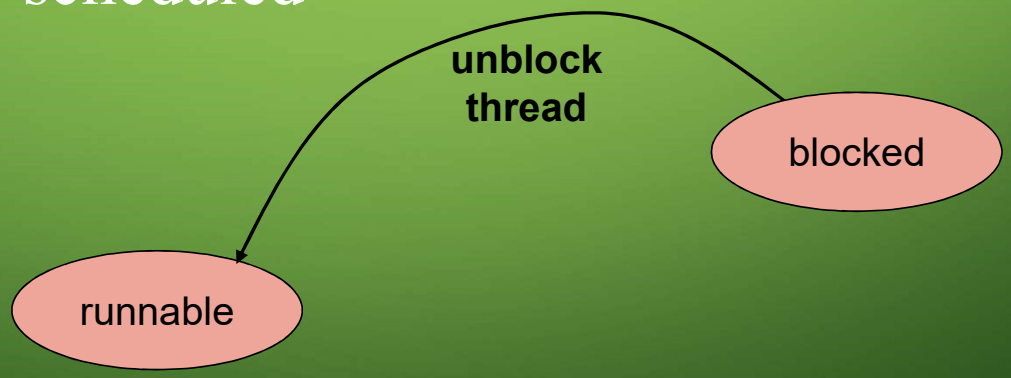
Runnable – to – Blocked Transition

- Runnable thread made unrunnable
  - call ***sleep*** method on it (for X milliseconds)
    - directly or via ***lock*** method
- Can ***not*** be scheduled
- There may be many threads in this state



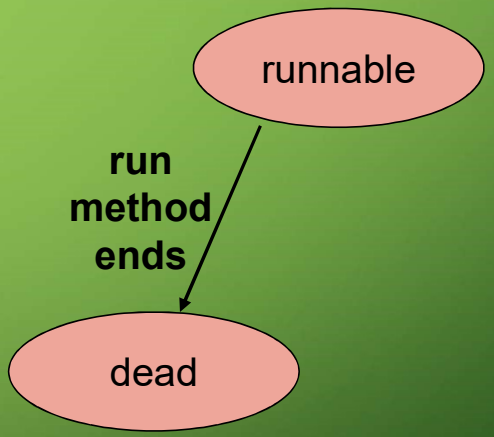
Blocked – to – Runnable Transition

- 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
- Dead 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 2 key `Thread` methods

- **`start()`**
  - makes thread runnable
  - calls the **`run`** method
  - **`Thread`** class' **`start`** method already does all of this
    - if your class that **`extends Thread`** you don't have to define **`start`**

- **`run()`**
  - executed when a thread is **`start()`**ed
  - where thread work is done (the point of it all)
  - **`Thread`** class' run method does nothing
    - if your class **`extends Thread`** you ***must*** define **`run`**
      - to specify what work your thread will do

For you, run >> start

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

`start` vs. `run`

- All code executes within a thread of execution
  - even the **`main`** method has a thread

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

- Now we have 2 threads. What about:

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

- Still just 1 thread. Why?

Run vs. Start Example

```
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) {}
        }
    }
}
```

What will `StartTester` do?

```
public class StartTester
{
    public static void main(String[] args)
    {
        RandomThread thread = new RandomThread();
        thread.start();
        while (true)
        {
            LocalDateTime today = LocalDateTime.now();
            long hour = today.getHour();
            long minute = today.getMinute();
            long second = today.getSecond();
            System.out.println(hour + ":"
                    + minute + ":" + second);
            try { Thread.sleep(10); }
            catch(InterruptedException ie) {}        }
    }
}
```

THIS *IS* A MULTITHREADED APPLICATION!

What will RunTester do?

```
public class RunTester
{
    public static void main(String[] args)
    {
        RandomThread thread = new RandomThread();
        thread.run();
        while (true)
        {
            LocalDateTime today = LocalDateTime.now();
            long hour = today.getHour();
            long minute = today.getMinute();
            long second = today.getSecond();
            System.out.println(hour + ":"
                    + minute + ":" + second);
            try { Thread.sleep(10); }
            catch(InterruptedException ie) {}        }
    }
}
```

THIS IS *NOT* A MULTITHREADED APPLICATION!

`Runnable` interface

- Alternative approach:
  - use implements **`Runnable`**

  AND
  - define **`start`** *AND* **`run`**

- **`Runnable`** interface has 1 method: **`run`**

```
public class RandomRunnable implements Runnable {
    private Thread proxyThread = null;
    private boolean die = false;

    public void kill() { die = true; }

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

    public void start() {
        if (proxyThread == null) {
            proxyThread = new Thread(this);
            proxyThread.start();
        }
    }
}
```

```
public Thread(Runnable target)
```

Specifies whose run method should be called when the thread executes

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();
  }
}
```

Thread.sleep will put a Thread into SLEEP mode with a recommendation that it stay there for the given number of milliseconds Thread.yield will put it into WAIT mode so it may run again straight away, or a higher process thread might step in.

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

- Why?
  - it's deprecated
  - Why?
    - 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

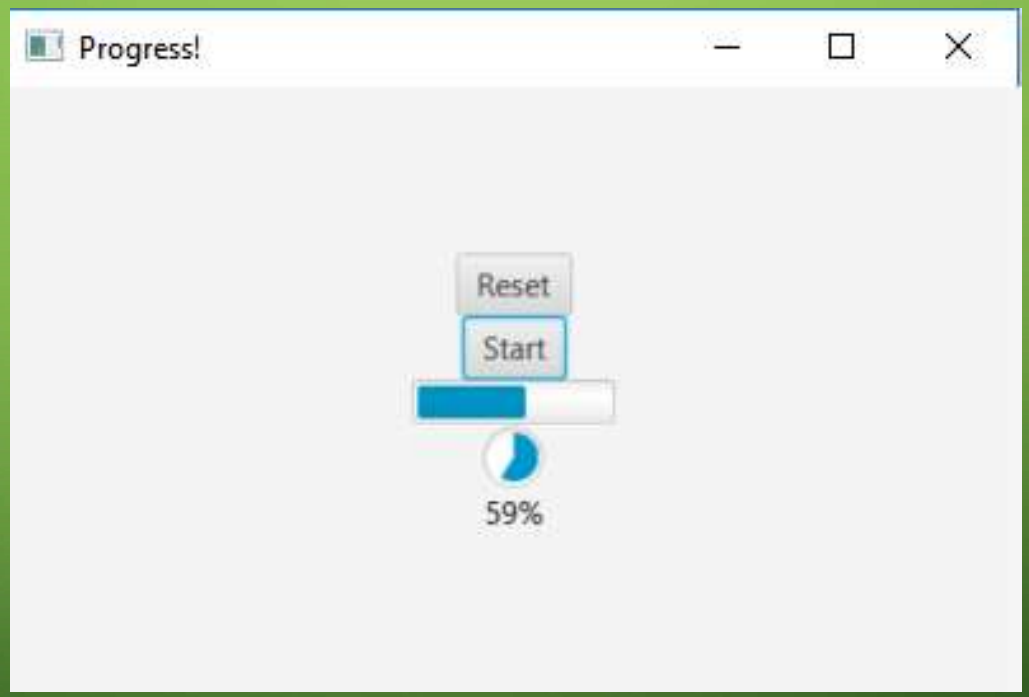
```
public class NiceThread extends Thread
{
   private boolean die = false;

   public void askToDie() { die = true; }

   public void run() {
      while (!die) {
         // do work here
         try {sleep(someAmountOfTime); }
         catch (InterruptedException ie) {   }
      }
      // set affairs in order}
   }
   // DEAD IS DEAD
```

JavaFX and Threads

- A bit tricky
- Why?
  - JavaFX has its own threads
- To have Thread update JavaFX control
  - extend Task
  - customize call() method
  - use Platform.runLater to execute run method that updates components

Progress Bars and Indicators

- Common need for Threads in UIs



```
public class ProgressBarExample extends Application {
    Button resetButton = new Button("Reset");
    Button startButton = new Button("Start");
    ProgressBar pBar = new ProgressBar();
    ProgressIndicator pIndicator = new ProgressIndicator();

    @Override
    public void start(Stage primaryStage) {
        VBox pane = new VBox();
        pane.setAlignment(Pos.CENTER);
        pane.getChildren().add(resetButton);
        pane.getChildren().add(startButton);
        pane.getChildren().add(pBar);
        pane.getChildren().add(pIndicator);
    …
```

```
startButton.setOnAction(e -> {
    Task<Void> task = new Task<Void>() {
        @Override
        protected Void call() throws Exception {
            for (int i = 0; i <= 100; i++) {
            ProgressUpdater updater = new ProgressUpdater(i/100.0);
            Platform.runLater(updater);
            try { Thread.sleep(10); }
                catch (InterruptedException ie) {
                    ie.printStackTrace();
                }
            }
            return null;
        }
    };
    Thread thread = new Thread(task);
    thread.start();
});
```

```
class ProgressUpdater implements Runnable {
        double progressValue;
        public ProgressUpdater(double initP) {
            progressValue = initP;
        }
        @Override
        public void run() {
            pBar.setProgress(progressValue);
            pIndicator.setProgress(progressValue);
        }
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

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 `Timer` do the counting**

Java Timers

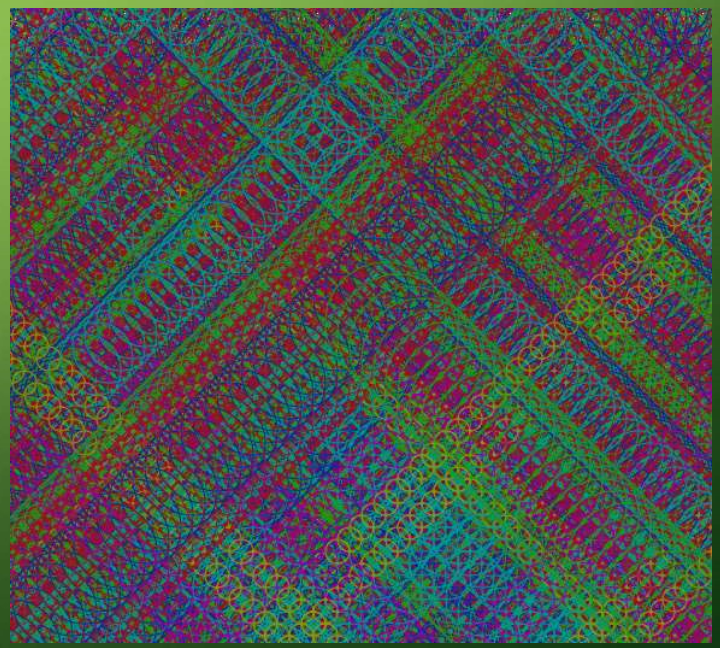
- Execute **`TimerTask`**s on schedule
  - via its own hidden thread

- What do we do?
  - define our own **`TimerTask`**
  - put work in run method
  - construct our task
  - construct a timer
  - schedule task on timer

- **`cancel`** unschedules our task (i.e. kills it)

What about JavaScript?

- Can also be multithreaded
- Need a kilt?
  - http://www3.cs.stonybrook.edu/~cse219/KiltPatternGenerator
- How?
  - `setInterval`
  - `clearInterval`



set/clearInterval

- **setInterval**
  - starts a Thread
  - returns the thread

- **clearInterval**
  - stops the thread

```
var timer = setInterval(render, 30);
…
clearInterval(timer);
```

clearInterval

- Starts a Thread
- Returns the thread

```
clearIntervaltimer =
setInterval(render, 30);
…
```