# Threads and Multithreading

# 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

# 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

- Allows you to do more than one thing at once
  - Play music on your computer's CD player,
  - Download several files in the background,
  - while you are writing a letter
- Multithreading is essential for animation
  - One thread does the animation
  - Another thread responds to user inputs

# Threads

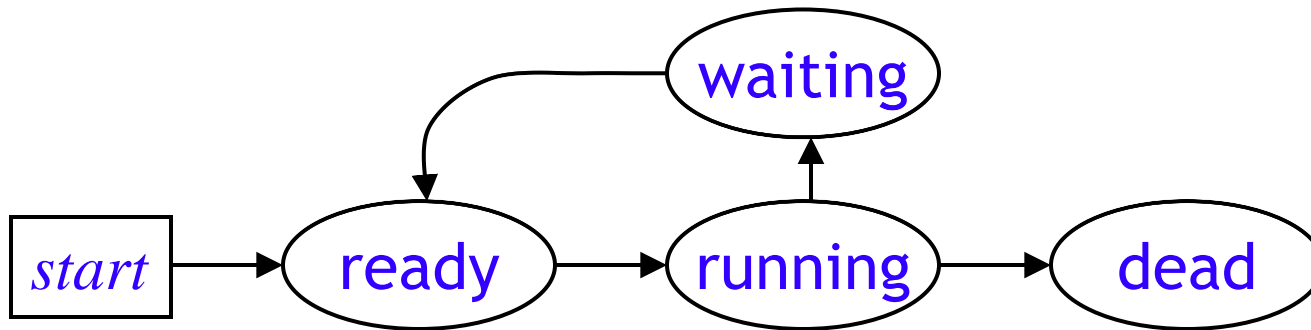
- A Thread is a single flow of control
  - When you step through a program, you are following a Thread
- Your previous programs all had one Thread
- A Thread is an Object you can create and control

# 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

- 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( )
- I recommend the second for most programs

# Extending Thread

- class Animation extends Thread {
    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

- 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( );
```

- 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;
```

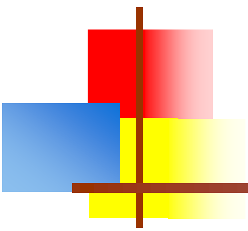
Thread #2:

```
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 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:
  - `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 End