

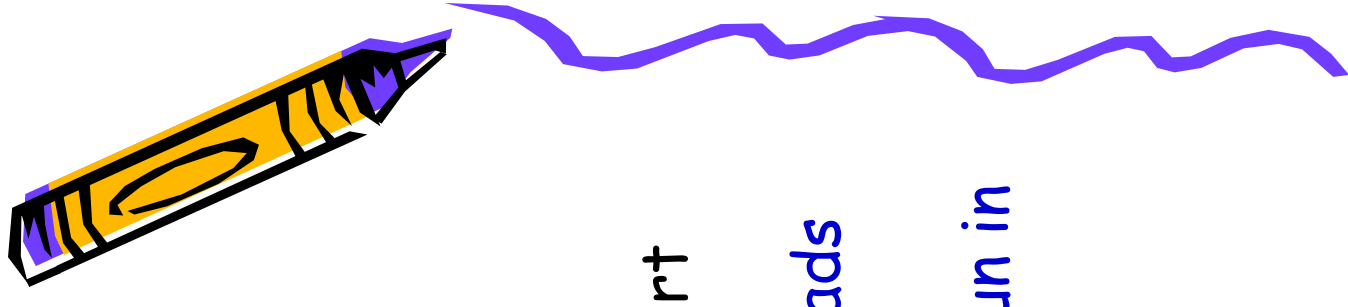
# Thread

- Thread Basics
- Thread Synchronization
- Animations




# Thread

- Thread: program unit that is executed independently
- Multiple threads “run simultaneously”
- Virtual machine executes each thread for short time slice
- Thread scheduler activates, deactivates threads
- Illusion of threads/process running in parallel
- Multiprocessor computers: threads actually run in parallel



# Running Threads

- Define class that implements Runnable
- Runnable has one method void run()
- Place thread action into run method
- Construct object of runnable class
- Construct thread from that object
- Start thread



```
public class MyRunnable implements
Runnable
{
    public void run()
    {
        // thread action
        try { do work }
        catch (InterruptedException ex) {}
    }
    ...
}
```

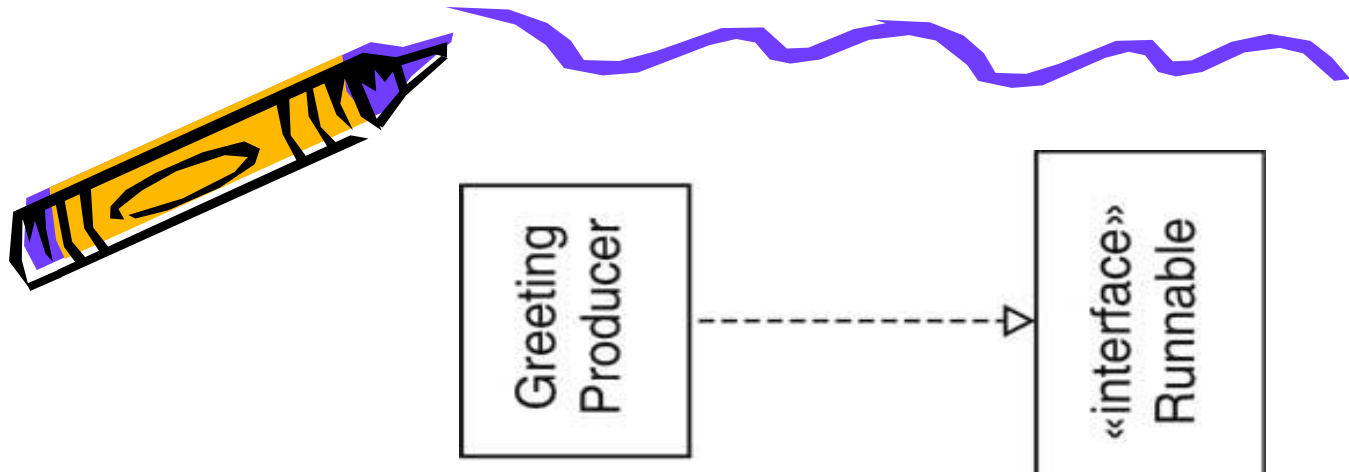
```
Runnable r = new MyRunnable();
Thread t = new Thread(t);
t.start();
```



# Run two threads in parallel

[Ch9/greeting/GreetingProducer.java](#)

[Ch9/greeting/ThreadTeste](#)

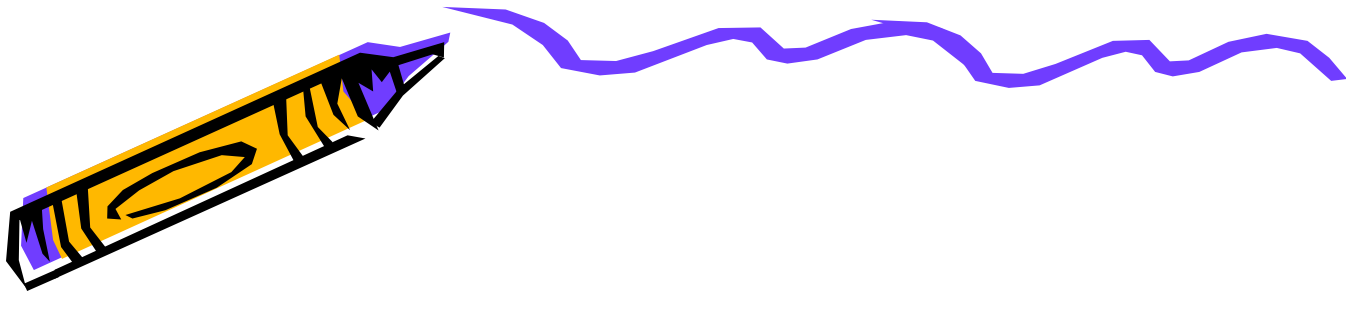


# Thread methods

```
thread.start ( );  
b = thread.isAlive ( );  
thread.interrupt ( );  
b = thread.isInterrupted ( );
```

## Static Thread Methods

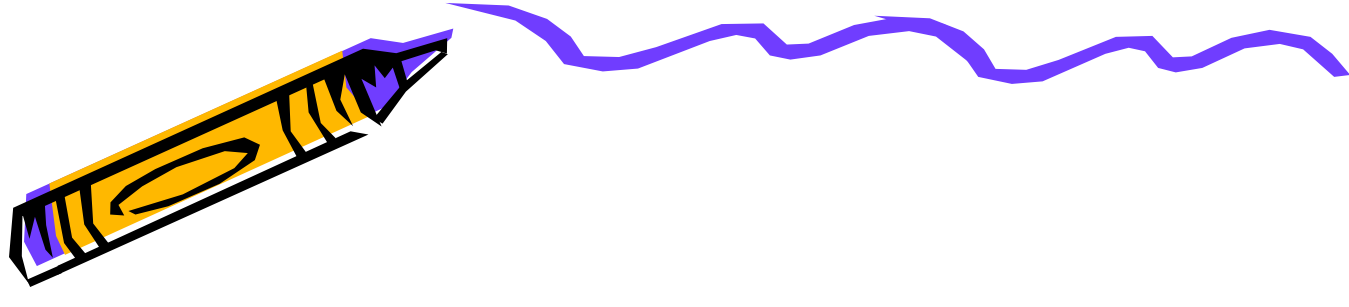
```
curr = Thread.currentThread ( );  
b = Thread.interrupted ( );  
Thread.sleep (millis);
```



# Running Threads

- Note: output not exactly alternating

```
1: Hello, World!  
1: Goodbye, World!  
2: Hello, World!  
2: Goodbye, World!  
3: Hello, World!  
3: Goodbye, World!  
4: Hello, World!  
4: Goodbye, World!  
5: Hello, World!  
5: Goodbye, World!  
6: Hello, World!  
6: Goodbye, World!  
7: Hello, World!  
7: Goodbye, World!  
8: Hello, World!  
9: Goodbye, World!  
9: Hello, World!  
10: Goodbye, World!  
10: Hello, World!
```

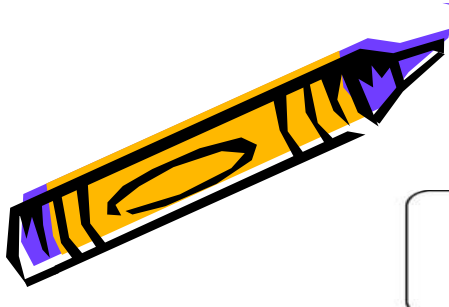
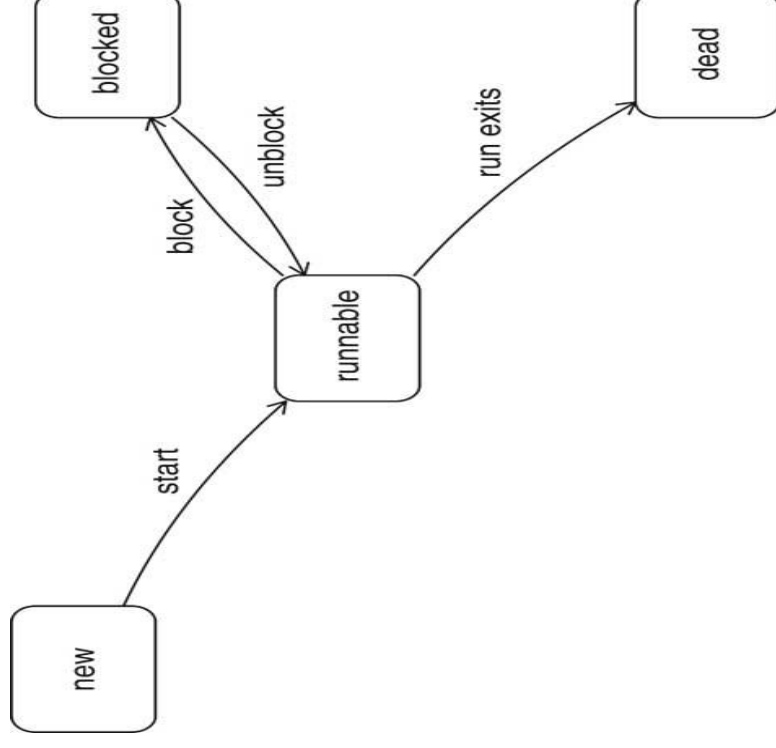


# Thread States

Reasons for thread to change state to blocked :

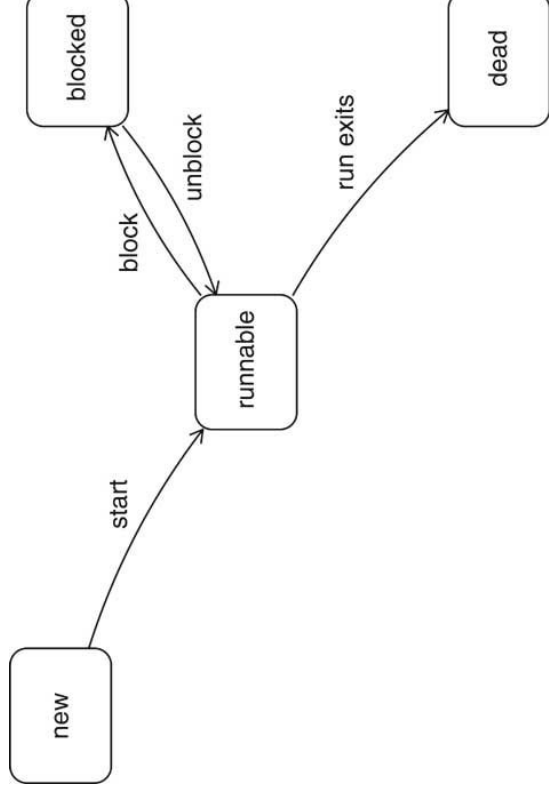
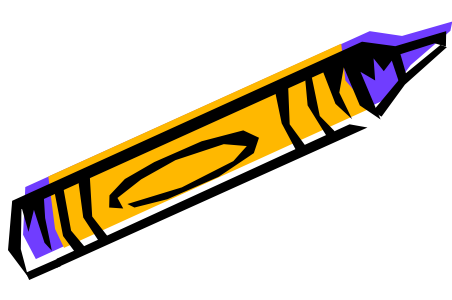
- *Sleeping*
- *Waiting for I/O*
- *Waiting to acquire lock (later)*
- *Waiting for condition (later)*

Unblocks only if reason for block goes away



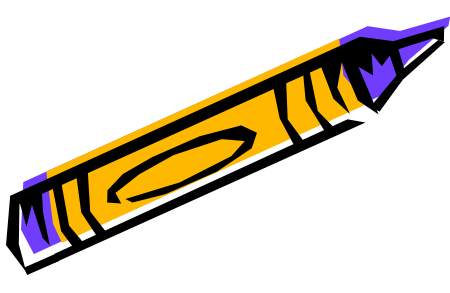
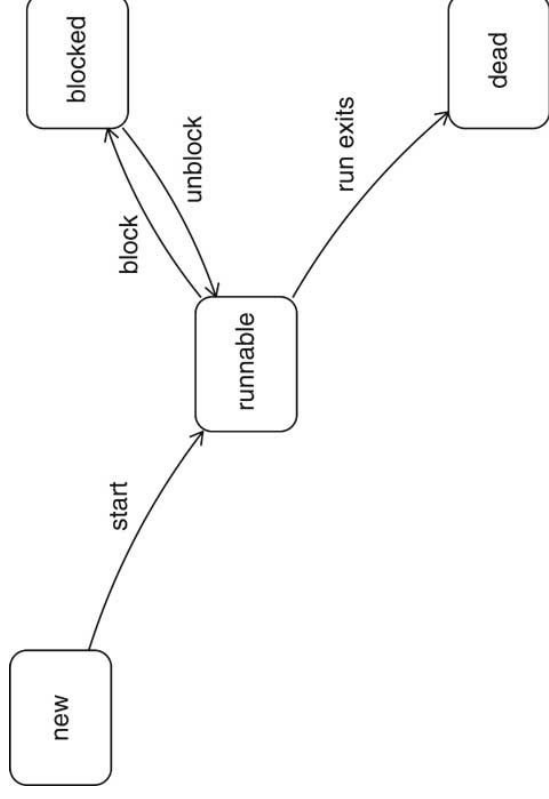
# Scheduling Threads

- Scheduler activates new thread if
  - a thread has completed its time slice
  - a thread has blocked itself
  - a thread with higher priority has become runnable
- Scheduler determines new thread to run
  - looks only at runnable threads
  - picks one with max priority



# Terminating Threads

- Thread terminates when run exits
- Sometimes necessary to terminate running thread
- Don't use deprecated **stop** method
- Interrupt thread by calling `interrupt`
- Calling `t.interrupt()` doesn't actually interrupt `t`; just sets a flag
- Interrupted thread must sense interruption and exit its run method
- Interrupted thread has chance to clean up



# Extends Thread

```
public class HelloThread extends Thread
```

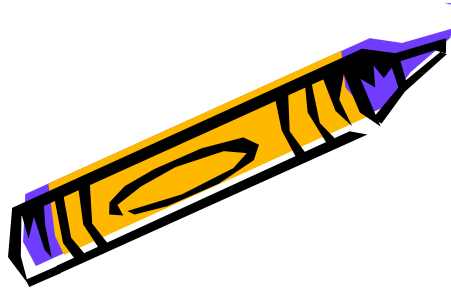
```
{  
    public void run()
```

```
{  
    System.out.println("Hello from a thread!");  
}
```

```
public static void main(String args[])
```

```
{  
    new HelloThread().start();  
}
```

```
}
```

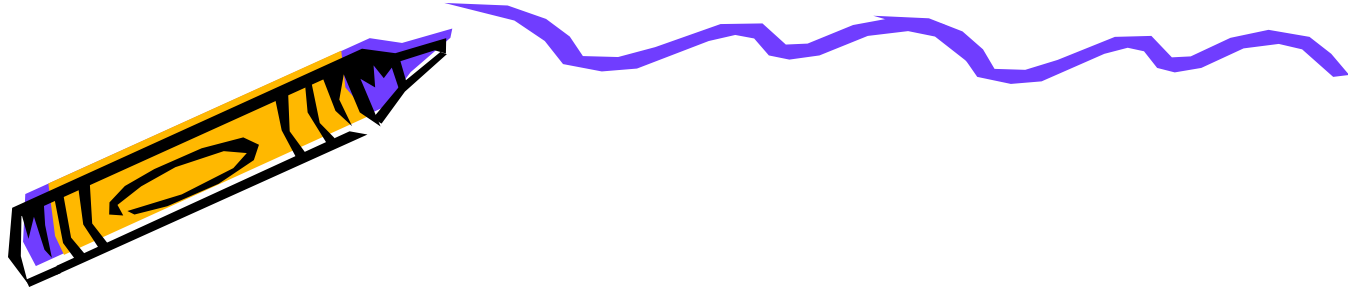


# Thread methods

```
thread.start ( );  
b = thread.isAlive ( );  
thread.interrupt ( );  
b = thread.isInterrupted ( );
```

## Static Thread Methods

```
curr = Thread.currentThread ( );  
b = Thread.interrupted ( );  
Thread.sleep (millis);
```



# Thread Synchronization

```
Producer Thread

int i = 1;
while (i <= count)
{
    if (! queue.isFull())
    {
        queue.add(i + ": " + greeting);
        i++;
    }

    Thread.sleep((int) (Math.random()
        * DELAY));
}
}
```



```
Consumer Thread

int i = 1;
while (i <= count)
{
    if (! queue.isEmpty())
    {
        Object greeting = queue.remove();
        System.out.println(greeting);
        i++;
    }

    Thread.sleep((int)
        (Math.random() * DELAY));
}
```

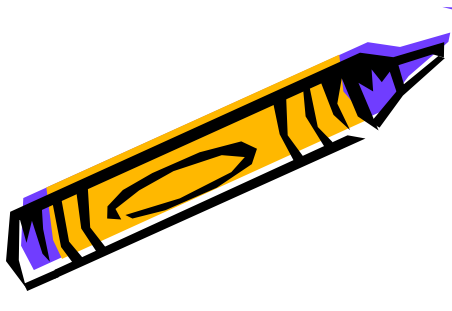


# Thread Synchronization

- Expected Program Output

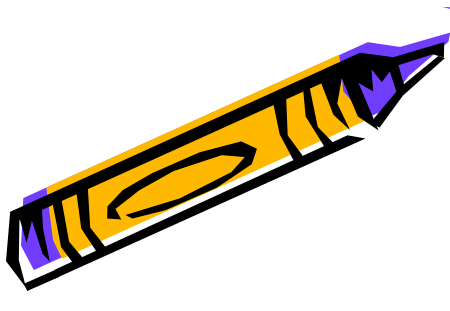
```
1: Hello, World!  
1: Goodbye, World!  
2: Hello, World!  
3: Hello, World!
```

```
..99: Goodbye, World!  
100: Goodbye, World!
```



# Thread Synchronization

- Why is Output Corrupted?
- Sometimes program gets stuck and doesn't complete
- Can see problem better when turning debugging on `queue.setDebug(true);`
- [Ch9/queue1/ThreadTester.java](#)
- [Ch9/queue1/Producer.java](#)
- [Ch9/queue1/Consumer.java](#)
- [Ch9/queue1/BoundedQueue.java](#)

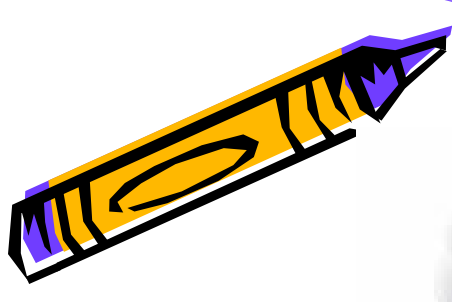


# Race Conditions

A race condition occurs *if the effect of multiple threads on shared data depends on the order in which the threads are scheduled*

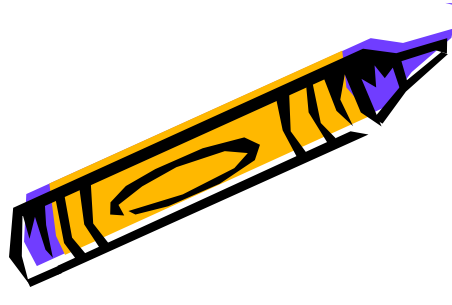


Illusion of correctness!



# Locks

- Thread can *acquire* lock
- When another thread tries to acquire same lock, it blocks
- When first thread *releases* lock, other thread is unblocked and tries again
- Two kinds of locks
  - Objects of class implementing `java.util.concurrent.Lock` interface type, usually `ReentrantLock`
  - Locks that are built into every Java object



# Reentrant Locks (java 5 interface)

- `aLock = new ReentrantLock();`

```
.aLock.lock();  
try  
{  
    protected code  
}  
finally  
{  
    aLock.unlock();  
}
```



# Deadlocks

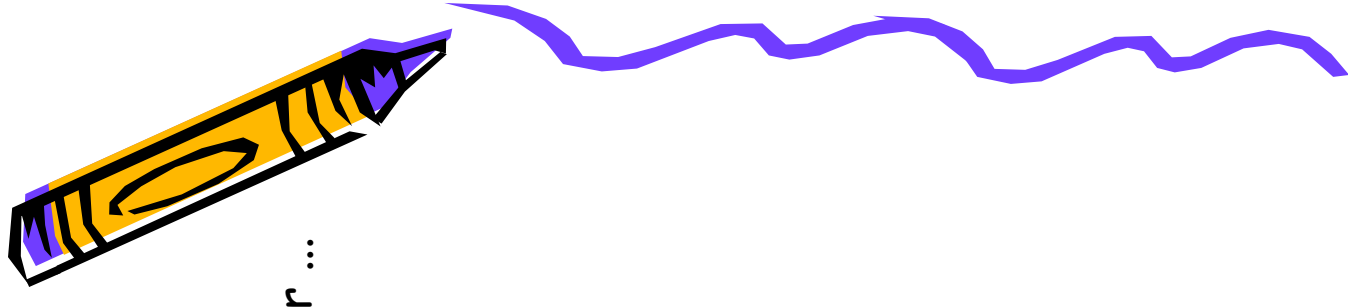
...if no thread can proceed because waiting for another ...

- if (lqueue.isFull()) queue.add(...);  
can still be interrupted

- Must move test inside add method

```
public void add(E newValue)
{
    queueLock.lock();
    try
    {
        while (queue is full)
            wait for more space
        ...
    }
    finally { queueLock.unlock(); }
```

- Problem: nobody else can call remove



# Avoiding Deadlocks

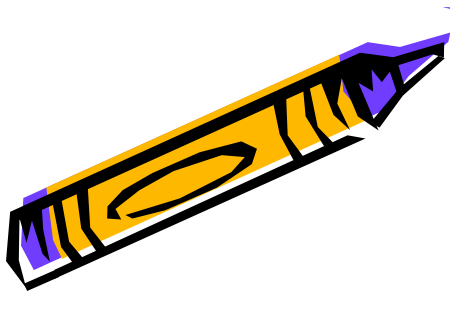
- Use *condition* object to manage "space available" condition

```
private Lock queueLock = new ReentrantLock();
```

```
private Condition spaceAvailableCondition = queueLock.newCondition();
```

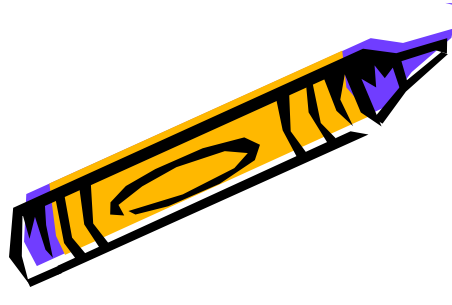
- Call `await` when condition is not fulfilled:

```
public void add(E newValue)
{
    while (size == elements.length)
spaceAvailableCondition.await();
    ...
}
```



# Object Locks

- Each object has a lock
- Calling a synchronized method acquires lock of implicit parameter
- Leaving the synchronized method releases lock
- Easier than explicit Lock objects
- public class BoundedQueue<E>  
    {  
        public synchronized void add(E newValue) { ... }  
        public synchronized E remove() { ... }  
    ...  
    }



# Object Locks



- `Object.wait` blocks current thread and adds it to wait set
- `Object.notifyAll` unblocks waiting threads

```
public synchronized void add(E newValue)
    throws InterruptedException
{
    while (size == elements.length) wait();
    elements[tail] = newValue;
    notifyAll(); // notifies threads waiting to remove elements
}
```

- [Ch9/queue3/BoundedQueue.java](#)

