Join, I

Java also has the equivalent of `pthread_join`:

```
> ThreadExample x = new ThreadExample()
> Thread t = new Thread(x)
> t.start()
I am a new thread!
> t.join()
> t.getState()
TERMINATED
```

In this case, at moment when `t.join()` call was made the Thread had already ended

Join, II

`join` merely waits for thread to end

A Java thread returns no exit value

How a Thread Can End

Java thread termination model is similar to `pthread` model

How a thread can end:

- `run` method terminates normally (reaches end or calls `return`)
- Exception thrown and not caught

Also, can "ask" a thread to terminate by calling its `interrupt` method

(Java `interrupt` is similar to canceling a `pthread` whose "cancellation type" is "deferred")

Example, I

Runnable class:

```
public class ThreadExample implements Runnable {
  public void run() {
    System.out.println("I am a new thread!
                        Going to sleep for 15 seconds ...");
    try {
        Thread.sleep(15000);
    } catch (Exception ex) {
    }
  }
}
```

Example, II

Execution:

```
> ThreadExample x = new ThreadExample()
> Thread t = new Thread(x)
> t.getState()
NEW
> t.start()
I am a new thread!  Going to sleep for 15 seconds ...
> t.getState()
TIMED_WAITING
> t.interrupt()
> t.getState()
TERMINATED
```

Detach

Java also has the equivalent of `pthread_detach`: make it a "daemon"

Example

```
> ThreadExample x = new ThreadExample()
> Thread t = new Thread(x)
> t.setDaemon(true)
> t.join()
```

In this case, `t.join()` will not return until thread ends (IF it ends!)

More Java Thread Management

Java also has thread priorities
(vary from 1 to 10, default is 5)

Abilities not present in `pthreads`:

- `dumpStack()` – prints stack
- `yield()` – "hint" to scheduler
- `toString()`

Locking in Java

Java has class `ReentrantLock`

Most commonly used methods are `lock`, `unlock`

"Reentrant" means same as Pthreads "recursive" – same thread can re-lock a lock that it already holds

(To drop the lock, the same number of unlocks must occur)

More Locking Methods

In keeping with Java's "more is more" philosophy, `ReentrantLock` has MANY more methods ...

`isLocked`

`tryLock`

`getOwner` – which thread holds the lock

`hasQueuedThreads` – are other threads waiting?

`getQueueLength` – how many?

`getQueuedThreads` – identity of waiting threads

etc.

How to Get/Drop Lock

Use `try/finally`:

```
ReentrantLock aLock = new ReentrantLock();

aLock.lock();
try {
    // access shared variables
    // that are protected by this lock
} finally {
    aLock.unlock();
}
```

Notice: NO "`catch`" block – only `try` & `finally`

`finally` block will execute if ANY exception is thrown in the `try` block

Of course if exception is thrown data might be left in undesirable state

Aside: Finally, I

`finally` clause is executed BEFORE code in `catch` clause but written AFTER — confusing to many

Java requires textual ordering try-catch-finally even though execution order is try-finally-catch

```
try {
    1. ... main code (throws exception) ...
} catch (Exception e) {
    3. ... exception handler ...
} finally {
    2. ... cleanup code ...
}
```

If you write code in order try-finally-catch, Java compiler complains " error: 'catch' without 'try' "

Aside: Finally, II

Instead could write:

```
try {
    try {
        1. ... main code ...
    } finally {
        2. ... cleanup code ...
    }
} catch (Exception e) {
    3. ... exception handler ...
}
```

Baroque and also confusing, but control flow matches textual order

Condition Variables

To create a condition variable that is associated with a particular lock:

```
ReentrantLock aLock = new ReentrantLock();

Condition reasonToWait = aLock.newCondition();
```

`Condition` is a Java "interface"

May have any number of conditions associated with same lock:

```
ReentrantLock aLock = new ReentrantLock();

Condition cond1 = aLock.newCondition();
Condition cond2 = aLock.newCondition();
Condition cond3 = aLock.newCondition();
```

Operations on Condition Variables, I

Pthreads	Java Condition interface
wait	await
signal	signal
broadcast	signalAll

Unlike in Pthreads, no need to explicitly manipulate associated lock because `Condition` object is associated with `ReentrantLock` object:

```
ReentrantLock lock = new ReentrantLock();
Condition cond = lock.newCondition();
```

Operations on Condition Variables, II

Pthreads:

```
pthread_mutex_t lock;
pthread_cond_t cond;
    ...
//  waits for "cond" to be signal-ed
pthread_cond_wait(&cond, &lock);
```

Java:

```
ReentrantLock lock = new ReentrantLock();
Condition cond = lock.newCondition();
    ...
//  waits for "cond" to be signal-ed
cond.await();
```

Hoare vs. Mesa Operation

Even though Java supports condition variables, it follows the Mesa model

... meaning: must call `await` in body of `while`, not body of `if`

Q: Why?

A1: `Condition` is not part of the core language, it is only a library class ... Java doesn't really "support" `Condition`

A2: Because of this advantage vs. Hoare model stated when we studied Hoare vs. Mesa:

"[Mesa model] does not require forced (and possibly non-optimal) Hoare-style immediate switch to signaled thread."

An Interesting Statement

Java textbook chapter uses `signalAll` in its examples then on page 748 says:

> Another method, signal, unblocks only a single thread from the wait set, chosen at random. That is more efficient than unblocking all threads, but there is a danger. If the randomly chosen thread finds that it still cannot proceed, then it becomes blocked again. If no other thread calls signal again, then the system deadlocks.

This statement is WRONG!

Why It's Wrong, I

1. Textbook statement is wrong

`signalAll` "wakes up" (i.e., changes state to runnable) all waiting threads but order in which they eventually run is unpredictable

So even with `signalAll` there remains a danger that an awakened thread will have to re-sleep & sequence of events it needs to re-awaken will not occur

Why It's Wrong, II

2. Textbook statement is terrible advice

Your code should NEVER depend on use of `signalAll` vs. `signal`

If you need one vs. the other then you have written code that is not "live" – i.e., not guaranteed to make progress

A concurrent program that is not live is BROKEN CODE

The ONLY valid reason to use `signalAll` vs. `signal` is performance optimization; for correct operation EITHER should be acceptable, ALWAYS

Another Interesting Statement

Later on page 748 Java textbook chapter says:

> In practice, using conditions correctly can be quite challening.

This is true

Monitor-like Behavior

Java provides language support for monitor lock: **synchronized** keyword

At most one of `foo`, `bar` may run at any moment:

```
public class ThreadExample implements Runnable {
    private int var1;   // instance variables
    private int var2;

    public synchronized void foo() {
      // statements accessing var1 and/or var2
    }

    public synchronized void bar() {
      // statements accessing var1 and/or var2
    }
}
```

Synchronized Keyword

`synchronized` means: at most one of all the methods declared `synchronized` may be executing at a time

This applies across ALL threads

This is not the complete monitor concept—that requires "wait" and "signal" operations—but it is the "monitor lock" part of the concept