# Threads

# Objectives

- Understand the concept of multithreading
- Understand the life cycle of a thread
- Understand thread synchronization
- Be familiar with thread scheduling and prioritization techniques

# Threads

- Provide
  - Concurrent paths of execution
  - A way to enable of parallel processing
    - Multi processor/core offer true parallelism
    - Single processor/core give the illusion
- Why Use Threads?
  - Improve program performance
  - Execute separate but coordinated tasks
  - Run background tasks

# Java Programs are Multi-Threaded

- Every uses multiple thread
  - The “main” thread executes `main()`
  - Another thread performs garbage collection
- Many library classes use threading techniques to improve perceived performance and protect data integrity

# java.lang.Thread

- `start()` – **housekeeping, becomes runnable**
- `run()` – **where your code is executed**
- `interrupt()` – **sets interruption status**
  - **If blocked on** `wait`, `sleep` **or** `join` `InterruptedException` **is thrown and the status cleared**
- `isInterrupted()` – **returns interruption status**
- `interrupted()` – **returns interruption status of current thread and resets the status**
- `join()` – **waits till the thread terminates**
- `sleep()` – **sleeps for some time**
- `yield()` – **pause and let another execute**
- `set(Default)UncaughtExceptionHandler()` – **installs a handler for uncaught exception**

# java.lang.Thread

- `resume()` - **resumes a suspended thread**
- `stop()` - **stops the thread**
- `suspend()` - **suspends the thread**

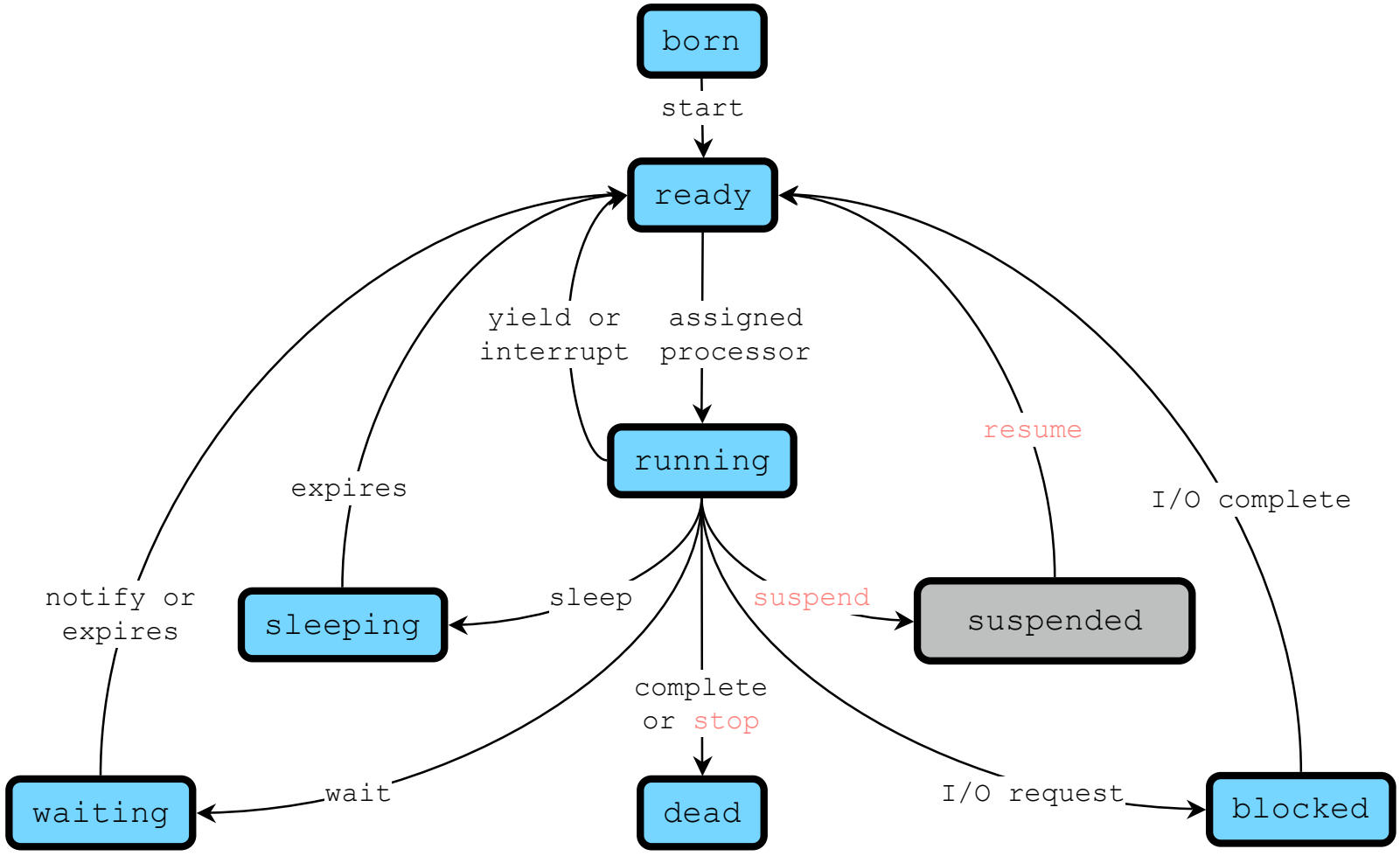
# How Threads Run

- Each thread needs time to execute
  - Pre-emptive time-slicing
    - Performed at process level
  - Scheduling
    - No scheduling guarantees
    - No fairness guarantees
    - No guarantee threads will make progress

# Thread Priority

- **Only thread scheduler uses priority**
  - **Priority heuristically influences scheduler**
    - **10 Crisis management**
    - **7-9 Interactive, event driven**
    - **4-6 IO - bound**
    - **2-3 Background computation**
    - **1 Run only if nothing else can**
- `setPriority()`
  - `MIN_PRIORITY`, **1**
  - `MAX_PRIORITY`, **10**
  - `NORM_PRIORITY`, **5**

# Thread Lifecycle



# Controlling Thread State

- **Use flag to indicate if thread is running**
- **Don't be selfish**
- **Do not use**
  - `stop()`
  - `suspend()`
  - `resume()`
- **Implement stop using flags,** `interrupt()`, `isInterrupted()` **and** `interrupted()`
- **Name threads (useful for debugging)**

# Controlling Thread State

```
volatile boolean running = true;

public void run()
{
   while( running && !Thread.interrupted() )
   {
      // run and do something in a loop
   }
}

public void stopThisThread()
{
   running = false;
   interrupt();
}
```

# Non-portable Thread Features

- The use of priorities and `yield()` to "tune" concurrency should be avoided
- Behavior of `yield()` and impact of priorities vary from VM to VM.

# java.lang.Runnable

- **Interface specifying the `run()` operation**
  - **Implemented by `Thread`**
- **Using `Runnable`**
  - **Do not need to extend from `Thread`**
  - **Separates logic from concurrency**
- **Implementation**
  1. **Define implementing class**
  2. **Provide an instance to the `Thread` constructor**
  3. **Start the thread**

# Synchronization

# Reasons for Synchronization

- **Example**

```
int totalAmount = price * shares;
// 1. report total amount to the user
// 2. get money equal to total amount
// 3. buy shares
```

***Price can be changed by other threads in between operations***
***Synchronize access to price***

# Monitor and Synchronization

- **Monitor** (one per object)
  - **Implemented in** `Object`
  - **Only one thread may have ownership of objects monitor at a time**
- **Synchronization**
  - **An objects monitor is used to restrict access to guarded statements**
  - `synchronized` **keyword**
    - **Method modifier**
    - **Controls access to a block**

# `synchronized`

- **A thread may not execute synchronized statements until it obtains ownership of controlling monitor**
  - **Class methods**
    - `Class` **object for the method's class**
  - **Instance methods**
    - **The object the method was invoked on**
  - **Synchronized block**
    - **The object specified in the synchronized block**

# `synchronized` Block

```
Object m_Lock = new Object();
public void doSomething()
{
   synchronized(m_Lock)
   {
      // current thread now has ownership of m_Lock
   }
   // current thread releases ownership of m_Lock
   ...
}
```

# Atomic Assignment

- Assignment cannot be interrupted

```
foo = 50;
foo = 42;
```

  - Any thread will see either `42` or `50`, not garbage
  - No guarantee value is result of most recent assignment
    - `volatile` – value is "flushed" immediately after written
- Except... `long` and `double`
  - Unless `volatile`

# Assignment & Synchronization

```
private Object mFoo;
...
public void set(Object foo)
{
   mFoo = foo;
}
```

- **Doesn't need to be synchronized**

# Assignment & Synchronization

```
private Object mFoo;
private boolean mFooAssigned;
...
public synchronized void set(Object foo)
{
   mFoo = foo;
   mFooAssigned = true;
   Syetem.out.println( foo );
}
```

- Needs to be synchronized, because:
  - Multiple related variables
  - Multiple operations

# Assignment & Synchronization

```
public void set(Object foo)
{
   mFoo = foo;

   Iterator it = colListeners.iterator();
   while(it.hasNext())
   {
      ((IEventListner)it.next()).processEvent(mFoo);
   }
}
```

- **Do you see a problem here?**
  - **Notification of listeners can be late**
  - **Member `mFoo` can be changed while updating listeners**

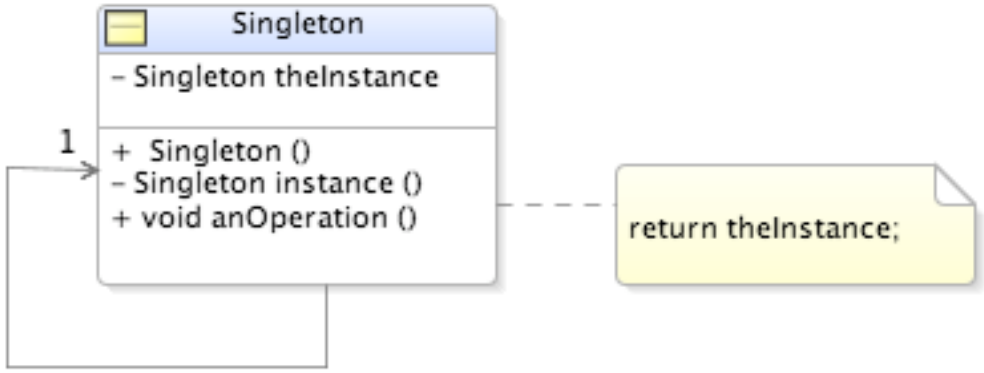
# Assignment & Synchronization

```
public void set(Object foo)
{
   mFoo = foo;

   Iterator it = colListeners.iterator();
   while(it.hasNext())
   {
      ((IEventListner)it.next()).processEvent(foo);
   }
}
```

- **Local variables can resolve some problems**

# Singleton Pattern




- Insures there is one instance of the class
  - Constructor is private

# Singleton enum Implementation

- Use of an enum is the preferred way to implement singletons

```
public enum EnumSingleton {
    INSTANCE;

    public void anyMethod() {
        // implementation
    }
}
```

# Singleton vs. Class Methods

- Benefits over a class having only class (static) methods
  - Constructors apply to instances
  - Only instances can be serialized
  - Interfaces can't specify static methods
  - Inheritance is not useful
  - May insert an object pool if a single instance is not sufficient (non-enum implementation)

# Lazy Initialization Class Holder Idiom

- Lazily initializes a static field
  - Classes are guaranteed not to be initialized until used
  - Useful for singleton implementation pre 1.5

```
public class MySingleton {
    private static class InstanceHolder {
        static final MySingleton singleton = new MySingleton();
    }

    private MySingleton() {
    }

    public static MySingleton instance() {
        return InstanceHolder.singleton;
    }
}
```

# Lazy Initialization Double-check Idiom

- **Double-check idiom**
  - **Broke prior to 1.5, due to weak `volatile` symantics**

```
public class LazyInitialization {
    private volatile LazyType lazyValue;

    public LazyType getLazyValue() {
        LazyType result = lazyValue;
        if (result == null) {
            synchronized(this) {
                if (result == null) {
                    result = lazyValue = computeInitValue();
        }   }   }
        return result;
    }
    ...
}
```

# Cooperating Threads

# Monitor Methods

- `java.lang.Object`
  - `wait()`
    - **Thread must have ownership of object's monitor prior to calling**
    - **Thread releases ownership, and waits, until…**
  - `notify()`
    - **Thread must own object's monitor**
    - **Thread wakes up any one of the threads that called `wait()` on the object**
    - **Awakened thread must obtain ownership of the object's monitor**

# Synchronization Example

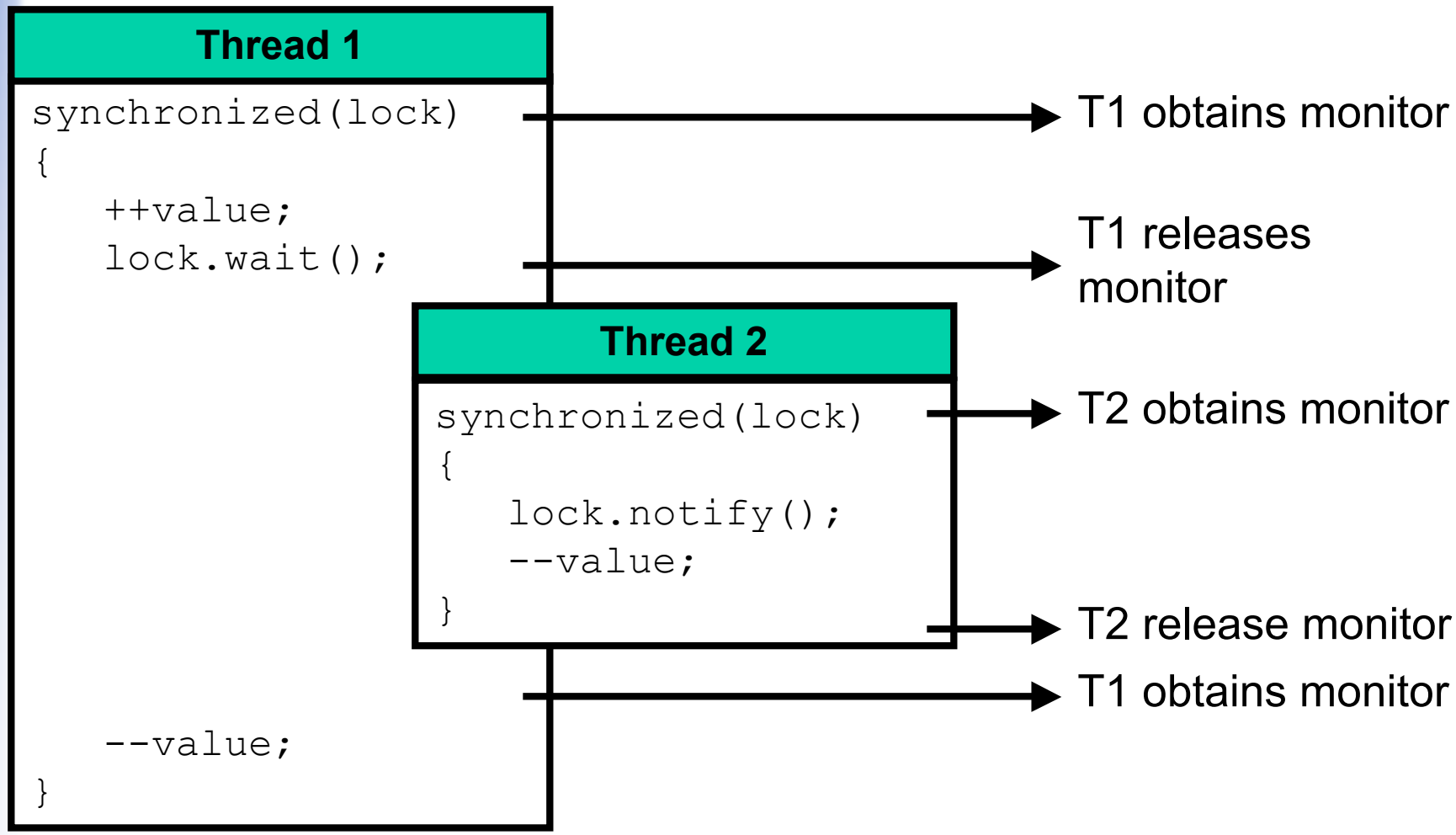
```
int value = 0;
Object lock = new Object;
void method1()
{
   synchronized(lock)
   {
      ++value;
      try {
         lock.wait();
      }
      catch(InterruptedException ie) {
         // means something called interrupt()
      }
      finally {
         --value;
      }
   }
}
```

# Synchronization Example

```
void method2()
{
    synchronized(lock)
    {
        lock.notify();
        --value;
    }
}
```

**What is a value of `value` going to be?**

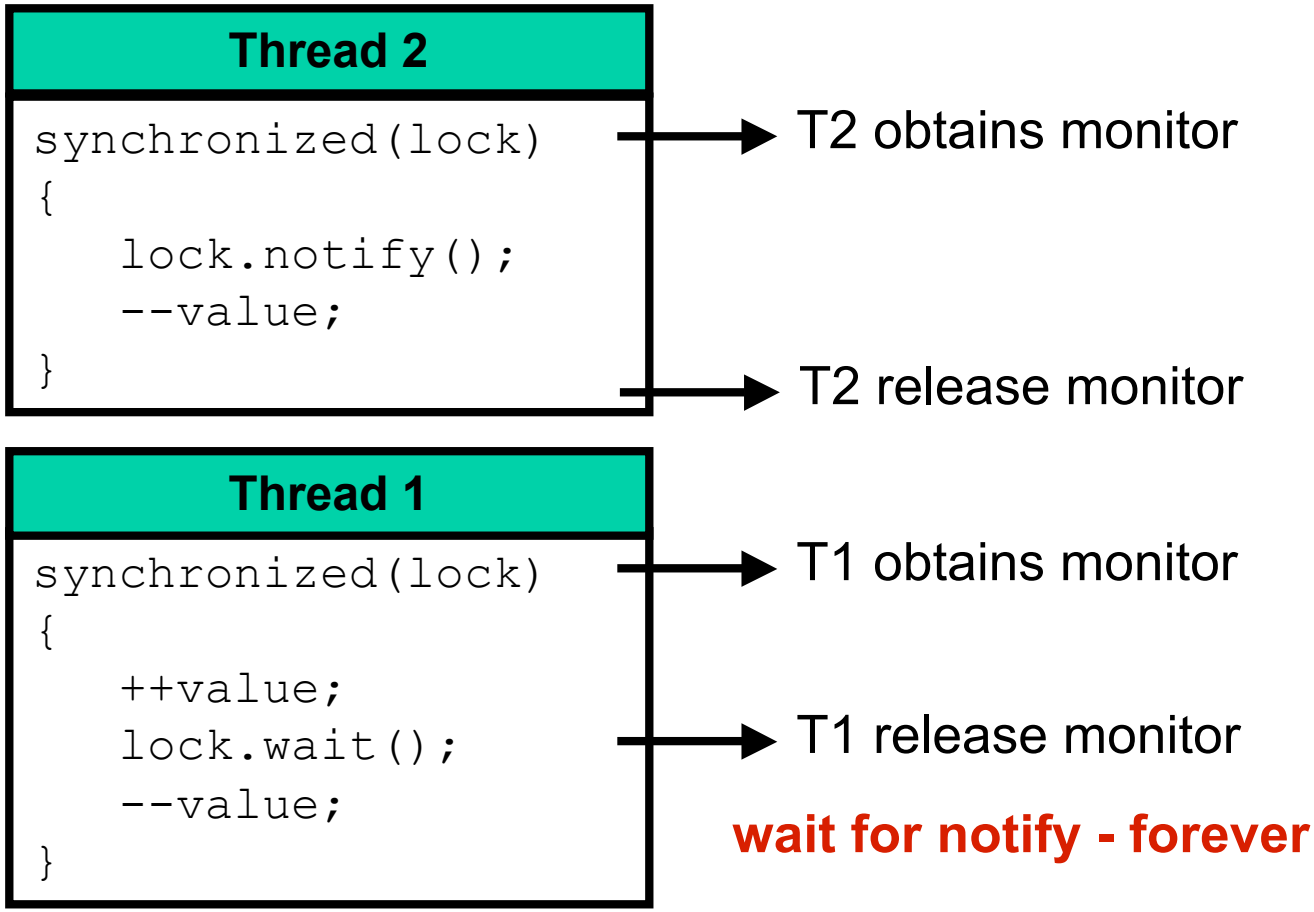
# How it Works


Thread 1
synchronized(lock)
{
++value;
lock.wait();
--value;
}
Thread 2
synchronized(lock)
{
lock.notify();
--value;
}
T1 obtains monitor
T1 releases monitor
T2 obtains monitor
T2 release monitor
T1 obtains monitor

# More Monitor Methods

- `java.lang.Object`
  - `notifyAll()` **- awakens all** *`wait()`***ing threads**
    - **Easy but more expensive**
  - `wait(long timeout)`
    - **Equivalent of `wait()`, but will try to re-obtain the lock and resume execution**
  - `wait(long timeout, int nanos)`
    - **Same as above with "higher precision"**

# Race Condition



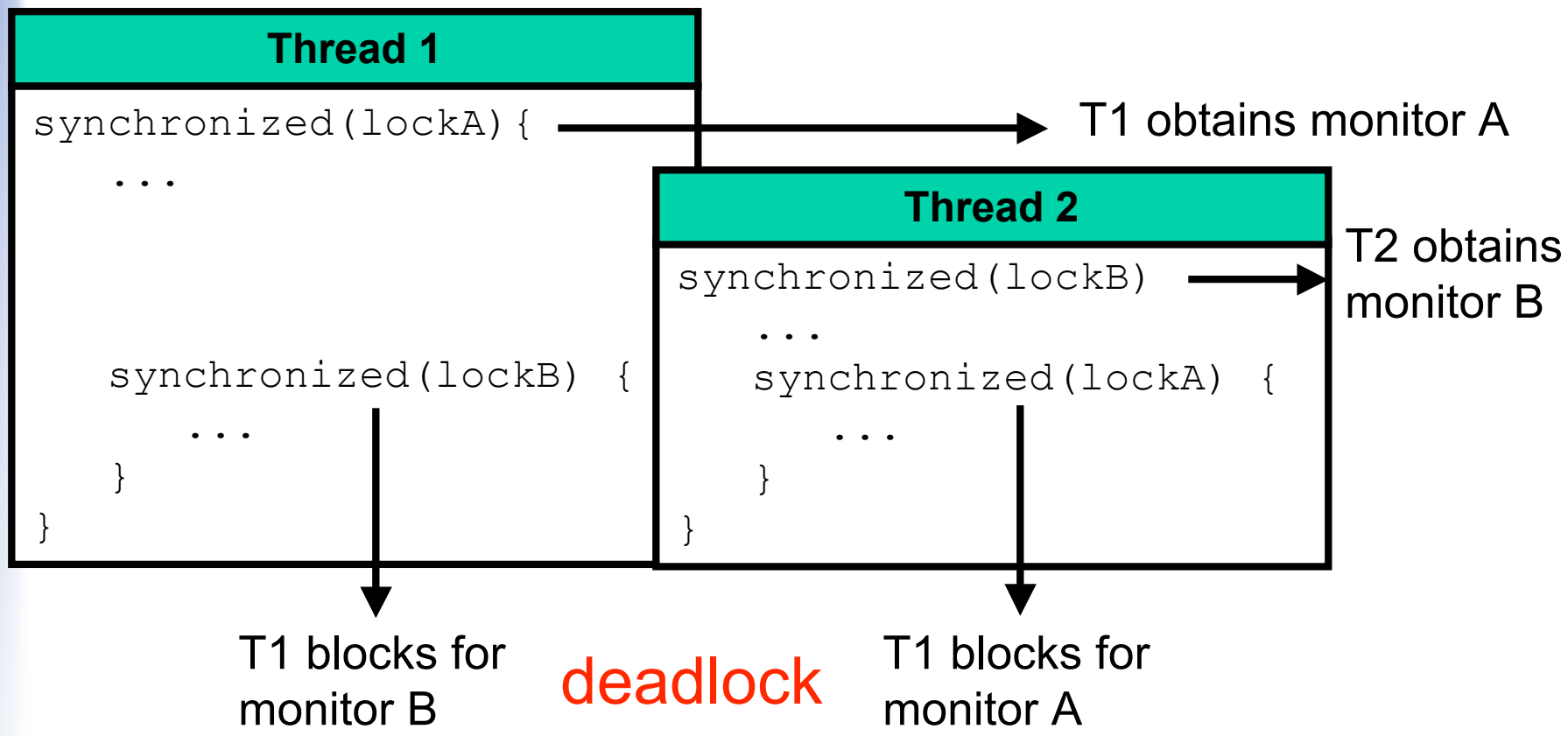
# Race Condition

- **Call `wait()` carefully, without appropriate `notify()` we have hung thread**
- **Calling `notifyAll()` can help avoid hung threads**
- **Only good design will save you! :-)**

# Lock Inversion

```
Object lockA = new Object();
Object lockB = new Object();
void method1() {
   synchronized(lockA) {
      ...
      synchronized(lockB) {
         ...
      }
   }
}
void method1() {
   synchronized(lockB) {
      ...
      synchronized(lockB) { {
         ...
      }
   }
}
```

# Lock Inversion - Deadlock



# Coordinating Worker Threads Producer - Consumer

- Both threads work on a specific task
  - Producer thread populates queue
  - Consumer picks the results

```
// consumer thread
public void run()
{
   while( true )
   {
      synchronized(list)
      {
         while(list.isEmpty())
            list.wait()
         Object task = list.getFirst();
      }
   }
```

# Coordinating Worker Threads Producer - Consumer

- **Producer thread**
  - **add**
  - **notify**

```
// producer thread
public void run()
{
   synchronized(list)
   {
      list.addLast(task);
      list.notify();
   }
}
```

# Coordinating Worker Threads AND-style

- Workers need to coordinate their work
  - Both tasks must complete prior to proceeding
  - AND-style joiner

```
Thread th1 = new Thread(runnable1);
Thread th2 = new Thread(runnable2);

th1.start();
th2.start();
// both threads running

// wait for first thread to terminate
th1.join();

// wait for second thread to terminate
th2.join();
```

# Coordinating Worker Threads OR-style

- Using callback
  - Each thread knows about others
  - When thread completes notifies other threads
  - Notified threads terminate
  - All joins cease blocking
- Use same `join()` scheme

# Shutting Down a Program

- To clean up and shut down a running Java program from the outside (from the command line or a script file)
- Use a *shutdownHook*
  - Simply an initialized but unstarted thread
  - ***Must*** extend Thread, Runnable doesn't work here
  - All shutdownHooks will be run when the Java runtime is told to stop; by a Cntl-C from the command line, for example

# Shutting Down a Program (cont'd)

- **Instantiate the** shutdownHook
- **Get the current runtime object by calling**

  Runtime.getRuntime();
- **Register the shutdownHook by calling**

  runtime.addShutdownHook(shutdownHook);
- **When the runtime shuts down, all the registered** shutdownHook**s will run**