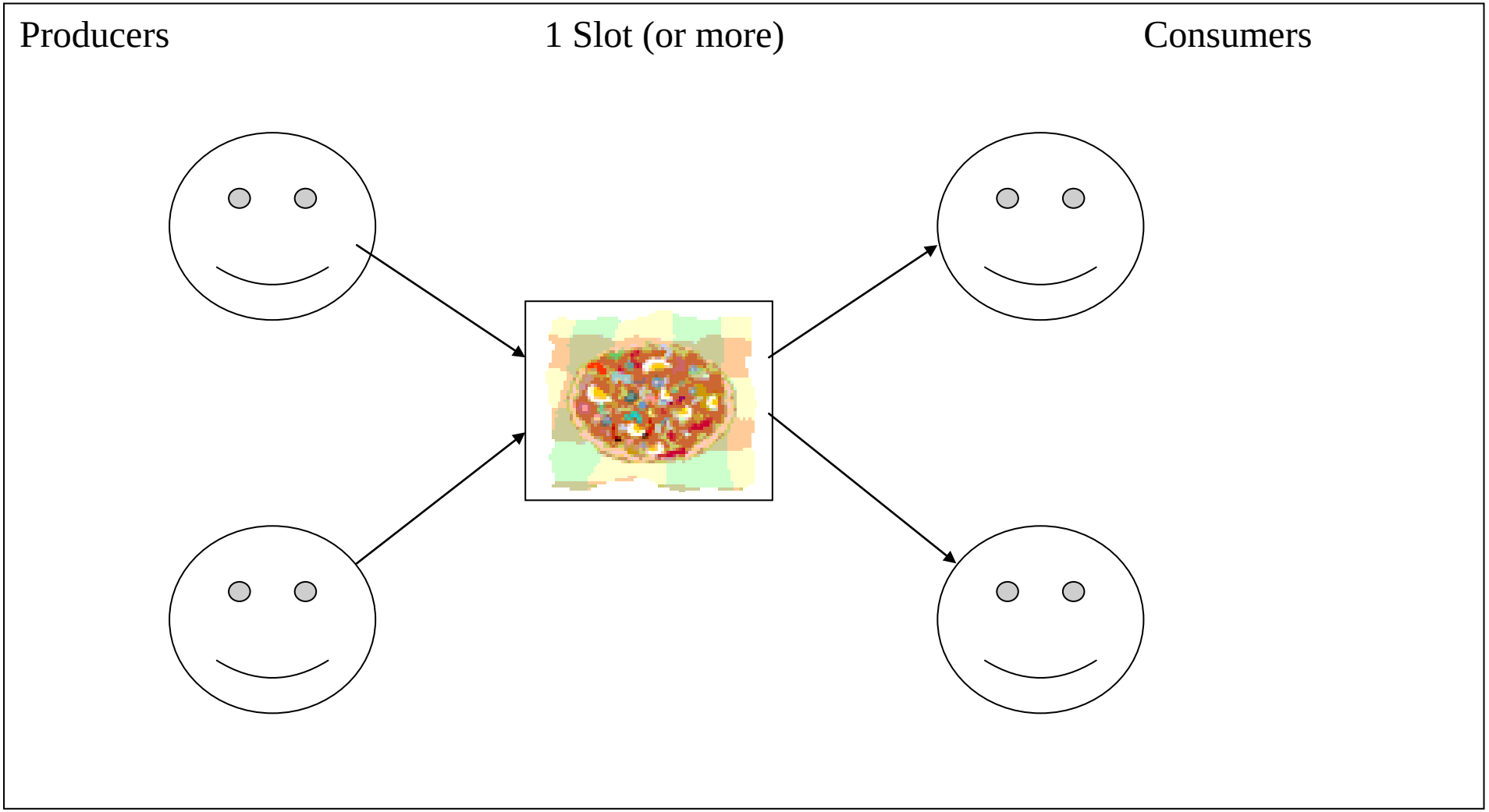
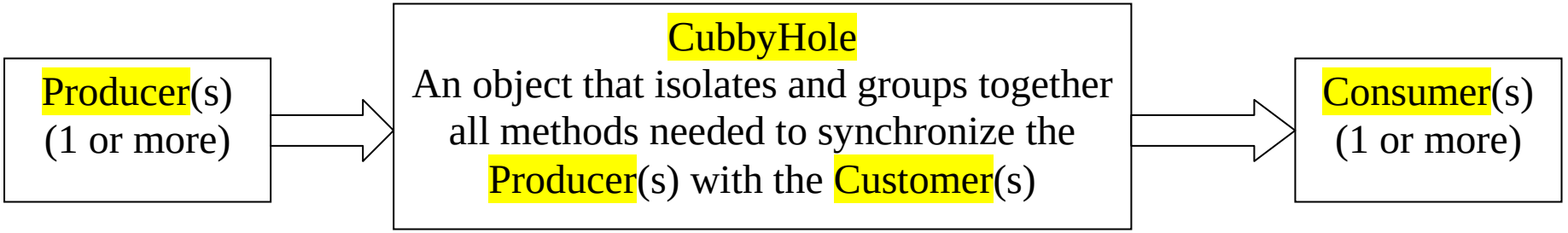
PRODUCERS AND CONSUMERS



OUR BASIC SCENARIO

Consider a producer-consumer scenario, where a Producer thread generates a stream of data that a Consumer thread uses.

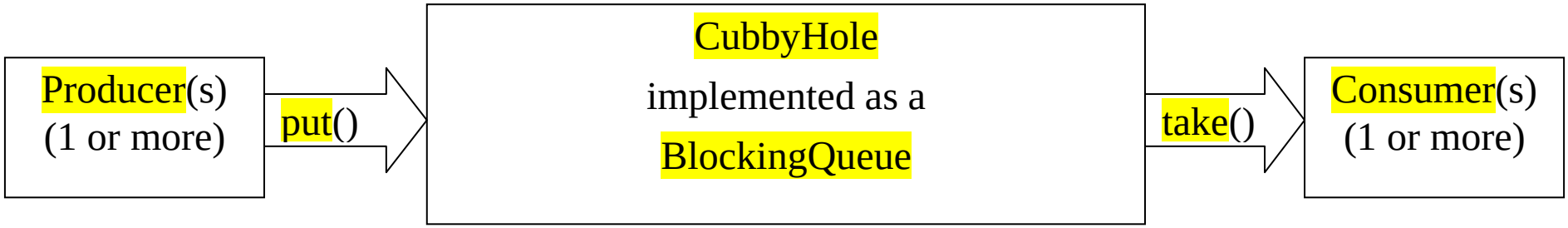
- Each Producer generates an integer between 0 and 9 (inclusive), and then stores it in a CubbyHole object.
- Consumers consume all integers from the CubbyHole (the exact same object into which Producers put the integers in the first place) as quickly as possible.
- Consumers should get each integer produced by Producers exactly once.



HIGH-LEVEL COOPERATION VIA BLOCKING QUEUES

For the CubbyHole, we use an implementation of the generic **interface** BlockingQueue<E> from the java.util.concurrent package

- In this example, we use ArrayBlockingQueue<E>
  - which allows the creation of "queues" with fixed capacities (aka bounded buffers)
  - here we use size 1 queues – equivalent to our required CubbyHole
  - but size N queues are also possible – a larger buffer between Producer and Consumer



- Other options are possible, st.t.:
  - LinkedBlockingQueue<E>: linked queue, variable size, but with an upperbound
  - SynchronousQueue<E>: zero-sized, require direct rendezvous

BLOCKING QUEUES FAIRNESS – SIDE BAR

- ArrayBlockingQueue defines a concurrent "queue" – or so it seems after a quick inspection
- Is this 'queue" really a "fair" queue, i.e. a FIFO structure?
- Surprisingly, this is "sloppy queue" only, which does **not** always respect the FIFO contract, i.e., barging is possible
- This is for efficiency reasons; it is typically hard to enforce fairness, in the context of non-deterministic threads
- Many concurrent data structures and algorithms are equally 'sloppy", they talk about queues, but these are **not** queues, in the strict theoretical sense
- If you are willing to pay the price (give up some performance), you can get a "fair" behaviour using an additional boolean parameter in the constructor:

```
BlockingQueue<Integer> abq = new ArrayBlockingQueue<Integer>(n);
BlockingQueue<Integer> abq = new ArrayBlockingQueue<Integer>(n, true);
```

BLOCKING QUEUES HANDLING OPTIONS – SIDE BAR

ArrayBlockingQueue has 4 sets of options of handling operations that cannot be completed immediately (i.e. if the queue is full or empty):

1. block the current thread indefinitely until the operation can succeed:

abq.put(obj) obj = abq.take()

2. return (immediately) a special value (**false**, for offer, and **null**, for poll) if not possible:

abq.offer(obj) obj = abq.poll()

3. similar to above, but block for a given maximum time limit before giving up:

abq.offer(obj, timeout, units), obj = abq.poll(timeout, units)

4. throw (immediately) an IllegalStateException if not possible:

abq.add(obj)

All the blocking methods (1, 3) are *interruptible*, i.e.

raise InterruptedException if the attempting thread receives an interrupt() signal

BLOCKING QUEUES - CODE

Producer's run method

```
public void run() {
  for (int i = 0; i < 10; i++) {
    cubbyhole.put(i);
    … // sleep?
  }
}
```

Consumer's run method

```
public void run() {
  int value = 0;
  for (int i = 0; i < 10; i++) {
    value = cubbyhole.take();
    … // sleep?
  }
}
```

In this scenario, our CubbyHole is a predefined BlockingQueue.

```
BlockingQueue<Integer> cubbyhole = new ArrayBlockingQueue<Integer>(1);
```

- Here, we use an array of size 1, but we could have used a larger number.
  - This gives the fixed capacity of our CubbyHole buffer
- We use put() and take(), which are blocking calls, but we could have used other options (see previous slide)

CRITICAL SECTIONS AND MONITORS

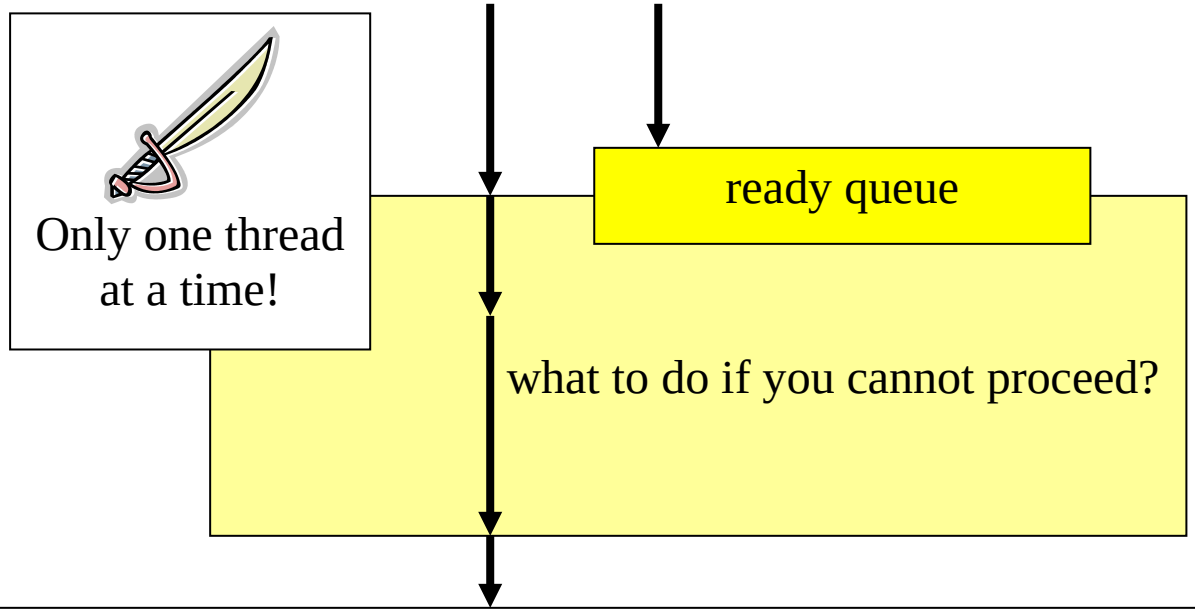
Recall that critical sections can be defined via:

- Intrinsic locks, introduced by **synchronized** keywords
- Explicit locks, introduced by ReentrantLock's or other classes that implement Lock

Critical sections offer:

- Mutual exclusion : a lock that only one thread can hold (at a given time)
- A ready queue for threads blocked to obtain the lock

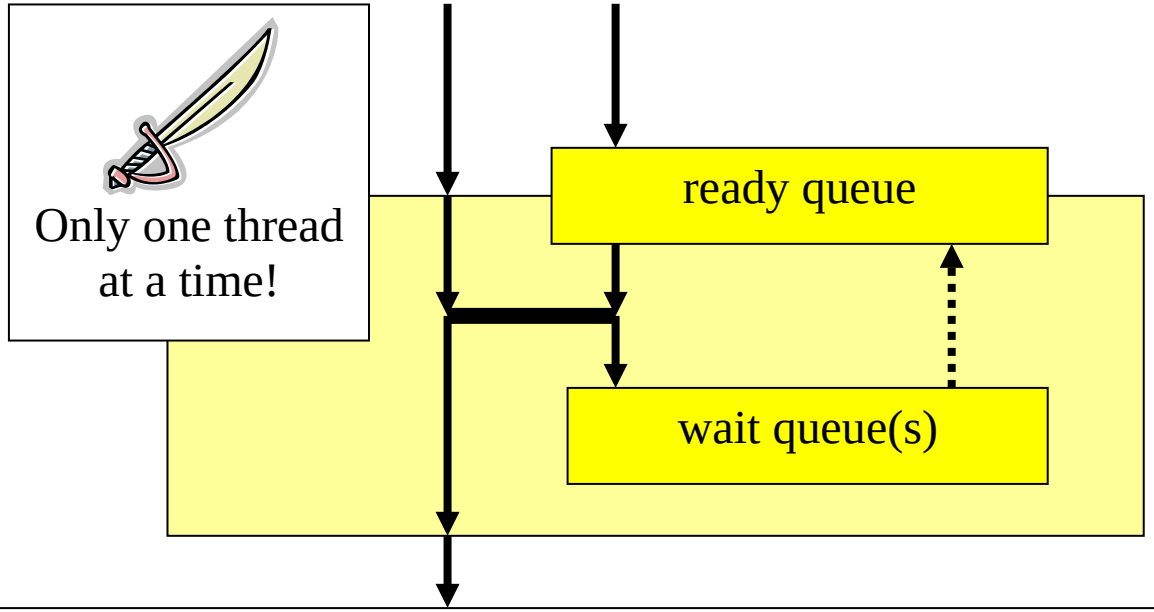
Monitors are an extension of the critical sections and offer additional features…



LOWER-LEVEL COOPERATION VIA MONITORS

Monitors are an extension of the already seen critical sections and offer:

- Mutual exclusion : a lock that only one thread can hold (at a given time)
- A ready queue for threads blocked to (re)obtain the lock
- One or more wait (await) queues for threads that did once hold the lock but have released it and are waiting for a signal (notification), and, optionally, a timeout - to be moved to the ready queue
- The number of queues depend on the API: Java implicit monitors (synchronized) have one queue for all, other API, e.g., ReentrantLock, offer more than one wait queue (which leads to performance and logic improvements)



CUBBYHOLE WITH MONITOR LOCKS

Producer's run method (same as before)

```
public void run() {
  for (int i = 0; i < 10; i++) {
    cubbyhole.put(i);
    … // sleep?
  }
}
```

Consumer's run method (same as before)

```
public void run() {
  int value = 0;
  for (int i = 0; i < 10; i++) {
    value = cubbyhole.get();
    … // sleep?
  }
}
```

- In this scenario we have to implement the CubbyHole object and its put and get methods.
- The bodies of get and put together (see next slides) form a critical region associated with a custom CubbyHole object
- Only one thread at a time is allowed, either one Producer or one Consumer
- … *more cooperation will be needed* …

CUBBYHOLE WITH EXPLICIT MONITOR LOCKS– HERE REENTRANTLOCK

```
public void put(int value) {
  aLock.lock();
  try {
    while (available) {
      try { condVar.await(); }
      catch (InterruptedException e){}
    }
    contents = value;
    available = true;
    condVar.signalAll();
  } finally {
    aLock.unlock();
  }
}
```

```
public int get() {
  aLock.lock();
  try {
    while (! available) {
      try { condVar.await(); }
      catch (InterruptedException e){}
    }
    available = false;
    condVar.signalAll();
    return contents;
  } finally {
    aLock.unlock();
  }
}
```

CUBBYHOLE WITH EXPLICIT MONITOR LOCKS– HERE REENTRANTLOCK

```
public void put(int value) {
  aLock.lock();
  try {
    while (available) {
      try { condVar.await(); }
      catch (InterruptedException e){}
    }
    contents = value;
    available = true;
    condVar.signalAll();
  } finally {
    aLock.unlock();
  }
}
```

```
public int get() {
  aLock.lock();
  try {
    while (! available) {
      try { condVar.await(); }
      catch (InterruptedException e){}
    }
    available = false;
    condVar.signalAll();
    return contents;
  } finally {
    aLock.unlock();
  }
}
```

We can get some help from a Lock variable, here aLock (or from the **synchronized** keyword)

CUBBYHOLE WITH EXPLICIT MONITOR LOCKS– HERE REENTRANTLOCK

```
public void put(int value) {
  aLock.lock();
  try {
    while (available) {
      try { condVar.await(); }
      catch (InterruptedException e){}
    }
    contents = value;
    available = true;
    condVar.signalAll();
  } finally {
    aLock.unlock();
  }
}
```

```
public int get() {
  aLock.lock();
  try {
    while (! available) {
      try { condVar.await(); }
      catch (InterruptedException e){}
    }
    available = false;
    condVar.signalAll();
    return contents;
  } finally {
    aLock.unlock();
  }
}
```

However, in your put and get code:

- The Consumer needs to wait in a queue until the Producer puts something in the CubbyHole
- and the Producer needs to notify the Consumer when it's done so... and the other way round

CUBBYHOLE WITH EXPLICIT MONITOR LOCKS– HERE REENTRANTLOCK

```
public void put(int value) {
  aLock.lock();
  try {
    while (available) {
      try { condVar.await(); }
      catch (InterruptedException e){}
    }
    contents = value;
    available = true;    // free = false
    condVar.signalAll();
  } finally {
    aLock.unlock();
  }
}
```

```
public int get() {
  aLock.lock();
  try {
    while (! available) {
      try { condVar.await(); }
      catch (InterruptedException e){}
    }
    available = false; // free = true
    condVar.signalAll();
    return contents;
  } finally {
    aLock.unlock();
  }
}
```

- You need to provide an available flag (or a free flag)
  - available=**true** means content **is** available (i.e., the cubby hole is **not** free)
- You also need to test this available flag and wait in a queue, if necessary...

CUBBYHOLE WITH EXPLICIT MONITOR LOCKS– HERE REENTRANTLOCK

```
public void put(int value) {
  aLock.lock();
  try {
    while (available) {
      try { condVar.await(); }
      catch (InterruptedException e){}
    }
    contents = value;
    available = true;
    condVar.signalAll();
  } finally {
    aLock.unlock();
  }
}
```

```
public int get() {
  aLock.lock();
  try {
    while (! available) {
      try { condVar.await();}
      catch (InterruptedException e){}
    }
    available = false;
    condVar.signalAll();
    return contents;
  } finally {
    aLock.unlock();
  }
}
```

Here we assume a single queue, condVar

Use **while** (not **if**) and signalAll (not signal) because of

- (1) barging, (2) potential interruptions (await/wait) and
- (3) possible lost or spurious signals (more details later)

CUBBYHOLE WITH EXPLICIT MONITOR LOCKS– HERE REENTRANTLOCK

```
public class CubbyHole2 {
    private int contents;
    private boolean available = false;  // false : "empty" contents
                                        // true : "full", contents are meaningful

    private Lock aLock = new ReentrantLock();  // (true) for a fair lock
    private Condition condVar = aLock.newCondition();
      // condVar indicates an internal "queue" for threads that need to wait:
      // * producers, if the cubby hole is full
      // * consumers, if the cubby hole is empty
      // unfortunately, because of barging, may contain mixed producers and consumers

    public void put(int value) { … }
    public int get() { … }
```

Recall that, in your put and get code:

- The Consumer needs to wait in a queue until the Producer puts something in the CubbyHole
- and the Producer needs to notify the Consumer when it's done so... and the other way round

CUBBYHOLE WITH EXPLICIT MONITOR LOCKS– HERE REENTRANTLOCK

```
public void put(int value) {
  aLock.lock();
  try {
    while (available) {
      try { condVar.await(); }
      catch (InterruptedException e){ ? }
    }
    contents = value;
    available = true;
    condVar.signalAll();
  } finally {
    aLock.unlock();
  }
}
```

- It is essential that we lock on the same monitor object, same aLock, on both sides.
- Otherwise we will get two distinct critical sections.

```
public int get() {
  aLock.lock();
  try {
    while (! available) {
      try { condVar.await(); }
      catch (InterruptedException e){ ? }
    }
    available = false;
    condVar.signalAll();
    return contents;
  } finally {
    aLock.unlock();
  }
}
```

- We could also create two separate queues: e.g.,
  - condProd, for producers and
  - condCons, for consumers – (later)!

DO NOT DO THIS (UNLESS YOU ARE AN EXPERT)

```
public void put(int value) {
  aLock.lock();
  try {
    if (available) {          use while
      try { condVar.await(); }
      catch (InterruptedException e){ ? }
    }
    contents = value;
    available = true;
    condVar.signal();         use signalAll
  } finally {
    aLock.unlock();
  }
}
```

```
public int get() {
  aLock.lock();
  try {
    if (! available) {        use while
      try { condVar.await(); }
      catch (InterruptedException e){ ? }
    }
    available = false;
    condVar.signal();         use signalAll
    return contents;
  } finally {
    aLock.unlock();
  }
```

- Also, avoid sending interrupt signals in this scenario, or treat these exceptions properly
- Best easy bet (to fill ?): re-throw the interruption: `throw e;`
- This will reraise `e` in the producer or consumer code, *after* releasing the lock (via the **finally**)

LOCK API

Class ReentrantLock implements interface Lock

new ReentrantLock () *or* (**false**) : faster but unfair (allows barging)

new ReentrantLock (**true**) : slower but fair (does not allow barging)

Lock API (excerpts):

void lock()	Acquires the lock
void lockInterruptibly()	Acquires the lock, interruptibly
Condition newCondition()	Returns a new Condition (queue) for this Lock
boolean tryLock()	Acquires the lock only if it is free at the time of invocation.
boolean tryLock(**long** time, TimeUnit unit)	Acquires the lock, timeout
void unlock()	Releases the lock

CUBBYHOLE WITH REENTRANTLOCK AND TWO SEPARATE QUEUES

```
public int get() throws InterruptedException {
   aLock.lock();

   try {
      while (! available) {
         try {
            consCond.await();
         } catch (InterruptedException e) {
            throw e;
         }
      }

      available = false;
      prodCond.signalAll();
      return contents;

   } finally {
      aLock.unlock();
   }
}
```

```
public void put() throws InterruptedException {
   aLock.lock();

   try {
      while (available) {
         try {
            prodCond.await();
         } catch (InterruptedException e) {
            throw e;
         }
      }

      contents = value;
      available = true;
      consCond.signalAll();

   } finally {
      aLock.unlock();
   }
}
```

With two separate queues, and if there are no lost signals, we could also use signal instead of signalAll:

- Producers and consumers wait in separate queues
- Interrupts that may cancel signals are thrown again

CUBBYHOLE WITH IMPLICIT MONITOR LOCKS – SYNCHRONIZED

CubbyHole skeleton – here with one slot

```
public class CubbyHole {
  private int contents;  // size 1
  private boolean available = false;

  public synchronized int get() { … }

  public synchronized void put(int value) { … }
}
```

- Get scenario

Consumer locks the CubbyHole

……

Consumer unlocks the CubbyHole

- Put scenario

Producer locks the CubbyHole

……

Producer unlocks the CubbyHole

CUBBYHOLE WITH IMPLICIT MONITOR LOCKS– SYNCHRONIZED PUT

```
public synchronized void put(int value) {
  while (available) {
    try {
      //Wait for a Consumer to get the value.
      //put the thread into a hidden wait queue
      this.wait();
    } catch (InterruptedException e) { }
  }

  contents = value;
  available = true;
  // Notify all waiting threads (incl any waiting Consumer)
  // that the value has been set.
  this.notifyAll();
}
```

- **synchronized** on **this**
- wait() on **this**
- notifyAll() on **this**

CUBBYHOLE WITH IMPLICIT MONITOR LOCKS– SYNCHRONIZED GET

```
public synchronized int get() {
  while (available == false) {
    try {
      // Wait for a Producer to put a value.
      // put the thread into a hidden wait queue
      this.wait();
    } catch (InterruptedException e) { }
  }

  available = false;
  // Notify all waiting (incl any waiting Producer),
  // that the value has been retrieved.
  this.notifyAll();
  return contents;
}
```

- **synchronized** on **this**
- wait() on **this**
- notifyAll() on **this**

- In this model, producers and consumers end up all in the same hidden wait queue
  - Could that be any problems?

WAIT, NOTIFY AND NOTIFYALL

- technically, these are methods of the Object class
- conceptually, these belong to the hidden Monitor type
- consequence: in Java any object can play part of a Monitor role
- these methods may only appear in a synchronized critical section of the any object, i.e.,

```
synchronized (mylock) {          often:      mylock == this
    …
    mylock.wait();               typically:  while (…) { … mylock.wait(); … }
    …
    mylock.notifyAll()
}
```

- wait(): waits indefinitely for notification in the hidden wait queue associated with the current object
- wait(**long** timeout): waits for notification or until the timeout period has elapsed (millis).
- notifyAll() method wakes up all threads waiting in the hidden wait queue associated with the current object
  - all awaked threads are put back into the hidden ready queue – and when run have to test if the conditions are actually met – that's why it typically needs a **while** around the wait
- notify() method wakes up one arbitrary thread from those waiting in the hidden wait queue

SPURIOUS NOTIFICATIONS , LOST NOTIFICATIONS AND BARGING CREATE PROBLEMS

- You've seen that we always have a **while** loop around our wait/await calls
- This is required when several threads of the same kind (producers OR consumers) are in the wait queue and only one of them should be allowed to continue (the others should go back to the waiting state)
- The problem is worse when we have mixed queues, where we have both producers AND consumers
- But the problem remains even if we have two separate queues, one for producers and another for consumers
- Or even when we only have one single producer and one single consumer
- For example, in some operating systems (some Linux versions using the pthreads library), spurious notifications may occur
- Threads in the waiting queue may receive a false wakeup signal, even if **no** notifications have been raised
- Therefore, always wrap you wait/await calls into loops that test your logical condition
- Doug Lea (a Java concurrency guru) points to scenarios in some JVMs where a single notification may get lost, if it happens concurrently with an interruption...
- Barging at wrong time may also create problems...
- Unless you are an expert, use notifyAll and **while** loops, even on systems that do not raise spurious notifications or do not lose any notification

Deadlocks and related problem

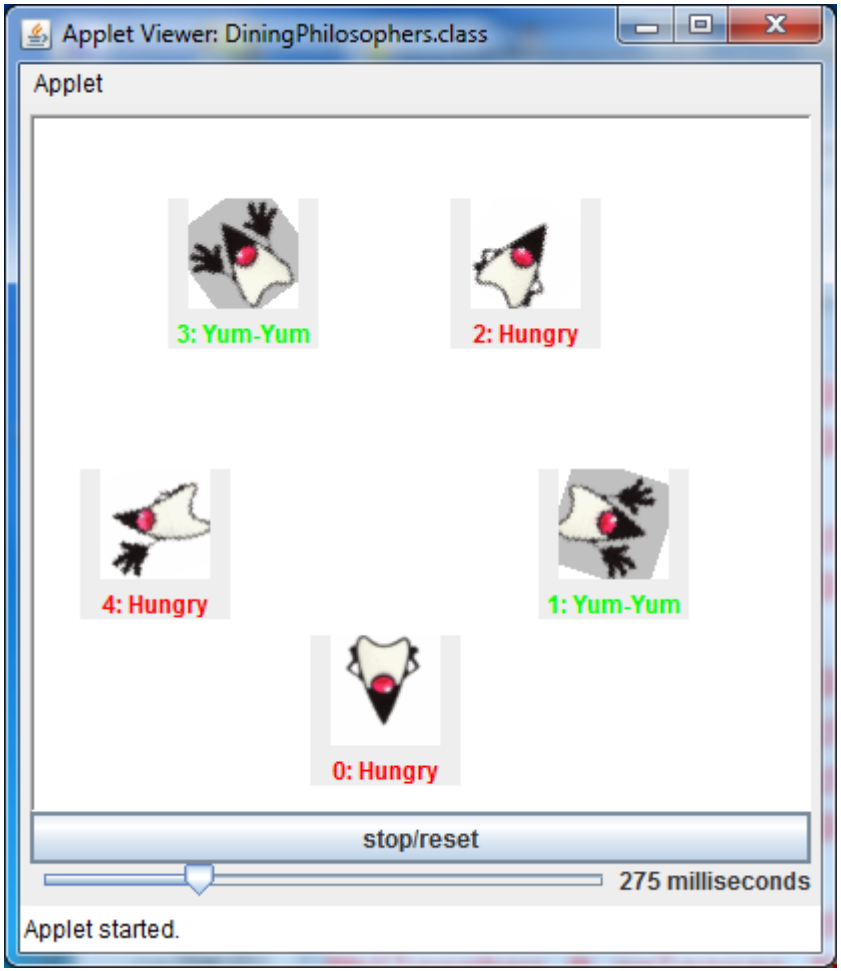
- Concurent design issues
- Deadlock: all thread in a set are blocked forever, typically in a circular chain

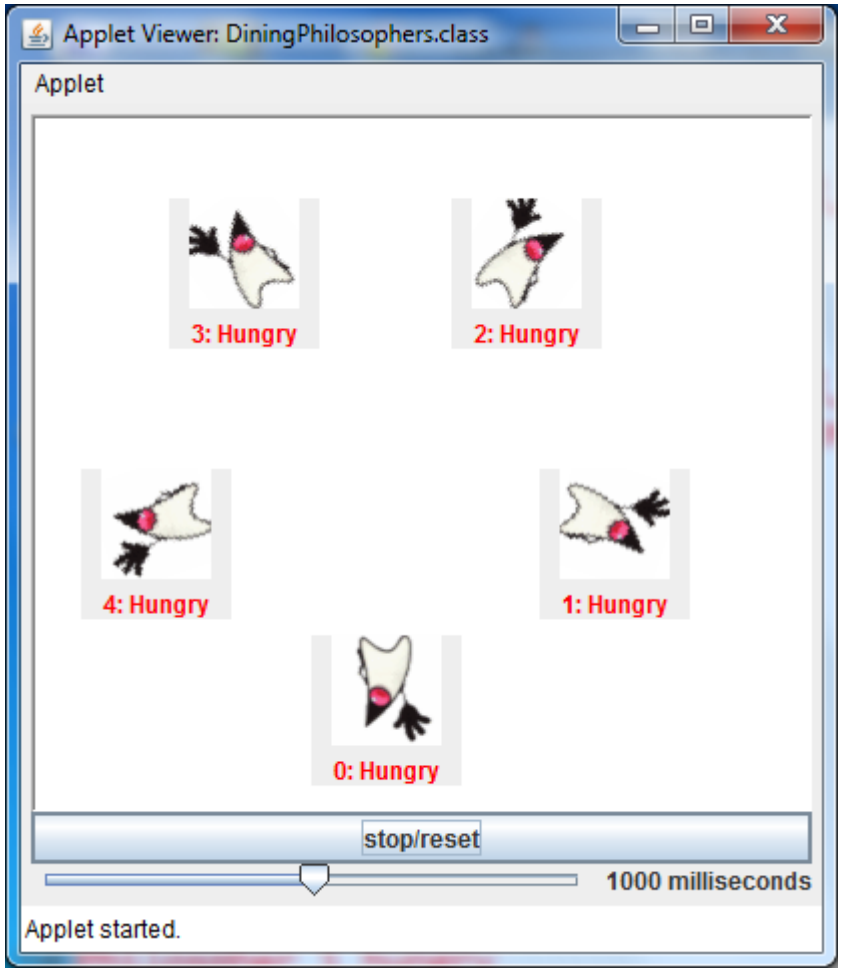
  WIKI: A real world analogical example would be an illogical statute passed by the Kansas legislature in early 20th century, which stated:

> "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

- Livelock: threads are not blocked (or not completely blocked), they may attempts to move on, but they are not able to agree how to progress
- Starvation: some threads cannot progress because other "greedy" threads "conspire" around them to take all their chances
- More details in the Java Tutorial – please **study** that section

Dining Philosophers Problem





Dining Philosophers Problem

- A naive implementation can result in deadlock (sooner or later), if
  - All philosophers can sit at the table concurrently
  - Each philosopher picks first its left fork and then its right fork
- Various strategies to avoid deadlock:
  - Restrict the number of philosophers that can sit at the table concurrently
    - Can be achieved with a Semaphore with 4 (=5-1) permits (where 5 is the number of philosophers)
  - Number forks and always pick them in a predefined global order (e.g. 1, 2, 3, 4, 5)
  - Philosophers alternate their fork picking order: 1 left, 2 right; 1 right 2. left
- More details in the Java Tutorial

SEMAPHORES

Semaphores:

- Can be thought as an extension of the monitors used in the producers/consumers scenario
- No distinction between producers/consumers (or readers/writers)
- Each semaphore has a number of *permits*, say N
- A generalized critical section, which allows access to up to N threads at any given time
- A *binary* semaphore is a semaphore where N=1
  - Up to 1 thread at any given time
  - Essentially, a binary semaphore is equivalent to the standard critical section
- A *counting* semaphore is the more general case, when $N \geq 1$
- API similar to Locks, but without condition queues
- Usual fairness issues

Semaphores API (excerpts):

Semaphore sem;	declaration
sem = **new** Semaphore(N);	constructor (default, unfair), N permits
sem = **new** Semaphore(N, fair);	constructor (with fairness option), N permits
sem.acquire();	acquires one permit, blocking, interruptible
bool ok = sem.tryAcquire();	acquires one permit, only if available, boolean result
bool ok = tryAcquire(timeout, TimeUnit.SECONDS)	acquires one permit, if available within the given time limit, boolean result
sem.release()	releases one permit

Note that there is no concept of lock/permit ownership

- A thread can release the permit acquired by another thread
- Requires strict coding discipline

Semaphores Example (excerpts):

```
try {
        sem.acquire();
        ... critical area – at most N threads at any given time
        sem.release();
} catch (InterruptedException ex) {
        ...
}
```

Dining Philosophers Problem - revisted

See sample code using semaphores

READERS AND WRITERS

Producers/consumers:

- Producers *generate* values, for the consumers
- Consumers *pick* values, generated by producers
- *Picking* a value is assumed to be a *destructive* act (no other consumers should pick it)
- The critical section has *exclusive* access for *all* threads, producers and consumers
  - At any given time, at most one producer or one consumer
- Condition queues available
- Fairness issues

Readers/writers:

- Can be thought as an extension of the monitors used in the producers/consumers scenarios
- Writers *write* values, for the readers
- Readers *read* values, written by writers
- *Reading* a value is assumed to be a *non-destructive* act (other readers can still read it)
- The critical section has *exclusive* access for writers, but *shared* access for readers
  - At any given time:
    - Only one writer and no readers
    - No writer, but any number of readers
- Lock API, condition queues available
- Usual fairness issues
  - E.g., can readers starve writers (will a newly arrived reader get access before an already blocked writer)?

Reader/writers API (excerpts):

ReentrantReadWriteLock rwl;	declaration
rwl = **new** ReentrantReadWriteLock();	constructor (default, *unfair*)
rwl = **new** ReentrantReadWriteLock(**boolean**);	constructor (*fairness* option)
Lock r = rwl.readLock();	get the *shared read* lock
Lock w = rwl.writeLock();	get the *exclusive write* lock

Locks r, w can be further used as we used RentrantLock (which also implements Lock), e.g.:

```
r.lock();
try { ...
} finally { r.unlock(); }
```

R/W Example (excerpts from a r/w wrapper around a Map Integer key → String value):

Method get – access can be *shared* between several readers

```
public String get(Integer key) {
        r.lock();

        try {
                …
                String value = …;
                return value;

        } finally {
                r.unlock();
        }
}
```

Method put – access is *exclusive* for at most one writer

```
public String put(Integer key, String value) {
        w.lock();

        try {
                …
                String oldvalue = …;
                …
                return oldvalue;

        } finally {
                w.unlock();
        }
}
```

Writer starvation is possible with un-fair ReaderWriters (scenario):

- R1 acquires the shared lock
  - W1 asks the exclusive lock, but blocks (coz' one reader, R1, is currently executing)
- R2 acquires the shared lock (possible with un-fair R/W)
- R1 releases the shared lock
  - W1 is still blocked (coz' one reader, R2, is currently executing)
- R3 acquires the shared lock (possible with un-fair R/W)
- R2 releases the shared lock
  - W1 is still blocked (coz' one reader, R3, is currently executing)
- R1 acquires the shared lock (possible with un-fair R/W)

...

Almost as readers "conspire" to starve the writer

SAMPLE PRODUCERS AND CONSUMERS SCENARIOS

Cubbyhole state = (*entry-queue*, lock-owner, internal-value, *wait/condition-queue*)

- the "*entry-queue*" is NOT a strict fair queue, *barging* may happen
- the "*wait-queue*" is NOT a strict fair queue, it is more like a *wait-set*

Examples:

- (–, –, –, –) no threads blocked on entry, no owner (no lock), empty, no threads waiting
- $P_1 \rightarrow$ (–, –, –, –) P_1 attempts to enter, rest as above
- (P_4, P_3, √, P_2) P_4 blocked in the entry-queue, monitor locked by P_3, full, P_2 waiting

Such scenarios can ilustrate that, in general (even without *spurious wakeups*):

- After wakeup, threads need to *recheck* the condition (loop)
- Before leaving, threads need to *notify all* (*signal all*) threads in the waiting queue

Straightforward scenario: 1 producer - 1 consumer

1.	$C_1 \rightarrow (-, -, -, -)$	C_1 enters the monitor
2.	$(-, C_1, -, -)$	C_1 locks the monitor, checks the cubbyhole condition (*empty*)
3.	$(-, -, -, C_1)$	C_1 enters the condition's wait queue (*and* unlocks)
4.	$P_1 \rightarrow (-, -, -, C_1)$	P_1 enters the monitor
5.	$(-, P_1, -, C_1)$	P_1 locks the monitor, checks the cubbyhole condition (*empty*)
6.	$(C_1, -, v_1, -) \rightarrow P_1$	P_1 fills the cubbyhole, signals all (*releases* C_1), unlocks, leaves
7.	$(-, C_1, v_1, -)$	C_1 locks the monitor, re-checks the cubbyhole condition (*full*)
8.	$(-, -, -, -) \rightarrow C_1$	C_1 empties the cubbyhole, signals all (*to nobody*), unlocks, leaves

N producers - N consumers : the need to recheck in a loop

1.	$P_1 \rightarrow (-, -, -, -)$	P_1 enters the monitor
2.	$(-, P_1, -, -)$	P_1 locks the monitor, checks the cubbyhole condition (*empty*)
3.	$(-, -, v_1, -) \rightarrow P_1$	P_1 fills the cubbyhole, signals all (*nobody*), unlocks, leaves
4.	$P_2 \rightarrow (-, -, v_1, -)$	P_2 enters the monitor
5.	$(-, P_2, v_1, -)$	P_2 locks the monitor, checks the cubbyhole condition (*full*)
6.	$(-, -, v_1, P_2)$	P_2 enters the condition's wait queue (*and* unlocks)
7.	$P_3 \rightarrow (-, -, v_1, P_2)$	P_3 enters the monitor
8.	$(-, P_3, v_1, P_2)$	P_3 locks the monitor, checks the cubbyhole condition (*full*)
9.	$(-, -, v_1, P_2\, P_3)$	P_3 enters the condition's wait queue (*and* unlocks)

... (*next slide*)

10.	$C_1 \rightarrow (–, –, v_1, P_2\, P_3)$	C_1 enters the monitor
11.	$(–, C_1, v_1, P_2\, P_3)$	C_1 locks the monitor, checks the cubbyhole condition (*full*)
12.	$(P_2\, P_3, –, –, –) \rightarrow C_1$	C_1 empties the cubbyhole, signals all (*releases* $P_2\, P_3$), unlocks, leaves
13.	$(P_3, P_2, –, –)$	P_2 locks the monitor, re-checks the cubbyhole condition (*empty*)
14.	$(P_3, –, v_2, –) \rightarrow P_2$	P_2 fills the cubbyhole, signals (*nobody*), unlocks, leaves
15.	$(–, P_3, v_2, –)$	P_3 locks the monitor, re-checks the cubbyhole condition (*full*)
16.	$(–, –, v_2, P_3)$	P_3 enters the condition's wait queue (*and* unlocks)

- Observe how important is that P_3 *rechecks* the conditions (i.e. uses a *loop*)!
- Even without *spurious* notifications... which may unfortunately happen on some platforms

N producers - N consumers : notify() vs notifyAll()

Doug Lea points to scenarios in some JVMs

- where a single notification may get lost, if it happens concurrently with an interruption...

Challenge: find a slightly more elaborate scenario

- where there is a crucial difference between notify() and notifyAll(),
- even without interruptions

Hint (1): two type of consumers (which ask different “pizza” toppings or crusts)

- Some take only even numbers (say “mild pizzas”)
- Others only odd numbers (say “hot pizzas”)
- This scenario is easier to implement and test

Another hint (2): just one type of consumers but barging at wrong times

- This scenario is difficult to implement and test

Possible deadlock scenario (1, see demo) because of notify() instead of notifyAll()

- C_1 wants odd numbers (*hot pizzas*)
- C_2 wants even numbers (*mild pizzas*)
- At this stage, assume an *empty* cubbyhole (*no pizza*)

- $C_1 \rightarrow (–, –, –, –)$ — C_1 comes
- $(–, –, –, C_1)$ — C_1 has to wait (because there is *no pizza at all*)
- $C_2 \rightarrow (–, –, –, –)$ — C_2 comes
- $(–, –, –, C_1\ C_2)$ — C_2 has to wait (because there is *no pizza at all*)
- P_1 comes, puts 103 (a *hot pizza*), notifies-one, which happens to be C2 (who asks *mild pizzas*)
- $(–, C_2, h_1, C_1)$

- (–, C_2, h_1, C_1)
- C_2 re-checks, but has to wait again (does NOT like *hot pizzas*)
- (–, –, h_1, C_1 C_2)
- P_1 comes back with 104 (another *hot pizza*), but has to wait (slot is full with the 103 *hot pizza*)
- (–, –, h_1, C_1 C_2 P_1)
- P_2 comes with 205 or 206 (a *mild pizza*), but has to wait (slot is still full with 103)
- (–, –, h_1, C_1 C_2 P_1 P_2)

- Summary: C_1, C_2, P_1, P_2 are all waiting while there is a *pizza* in the slot (103, a *hot pizza*)
- Everybody is frozen (although we have all the ingredients to move on)

This will **not** happen if:

- We use notifyAll() instead of notify() one, *or*
- We have two separate queues, one for *hot* and another for *mild pizzas*

Possible deadlock scenario (2, no demo) : notify() can NOT solve barging issues

Assume three producers and three consumers

- P_1 arrives and puts its value v_1 (but there is nobody to notify)
- P_2, P_3 and P_1 (again) arrive and wait (because the slot is full with v_1)
- $(-, -, v_1, P_1\ P_2\ P_3)$
- C_1 arrives, picks v_1, notifies one ... say P_1, which competes on re-entry
- $(P_1, -, -, P_2\ P_3) \rightarrow C_1$
- C_2, C_3 and C_1 (again) arrive fast, barge past P_1, and wait (because the slot is empty)
- $(P_1, -, -, P_2\ P_3\ C_1\ C_2\ C_3)$
- Finally, P_1 re-enters ...

- P_1 puts its value v'_1, and notifies one ... which happens to be another producer P_2 !
- $(P_2, -, v'_1, P_3\ C_1\ C_2\ C_3) \rightarrow P_1$
- P_2 re-enters and waits again (because the slot is full with v'_1)
- $(, -, v'_1, P_2\ P_3\ C_1\ C_2\ C_3) \rightarrow$
- P_1 arrives once more and waits (because the slot is full with v'_1)
- $(, -, v'_1, P_1\ P_2\ P_3\ C_1\ C_2\ C_3) \rightarrow$

All six threads, P_1, P_2, P_3, C_1, C_2, C_3, are now all blocked in the internal waiting queue

Also, the slot is full, with v'_1

The threads could continue with a better policy, e.g., notifyAll()...

Exercises

BlockingQueues, Locks and Semaphores are equally powerful, conceptually

Exercises (not easy to these properly, but you will learn a lot)

- Try to re-implement a ReentrantLock using Semaphores
- Try to re-implement a Semaphore using ReentrantLocks
- ...
- Try to re-implement all of them using atomic integers, e.g., compare-and-swap

See also the references, esp. *Alfonse...*, on the difficulty of implementing a Semaphore on top of the Java intrinsic monitor (**synchronized**)

Required Readings

- Handouts, samples
- JDK 6 Documentation (Java Reference)
- The Java Tutorial
  - Essential Classes: Concurrency
  - Creating a GUI with JFC/Swing: Concurrency in Swing
- CHM versions seems better (search, index)

Recommended Readings (but not required)

Stephen J. Hartley

Alfonse, wait here for my signal!

(how to implement Semaphores on top of Java intrinsic monitors)

In Proceedings of SIGCSE'1999. pp.58~62

http://portal.acm.org/ft_gateway.cfm?id=364590&type=pdf

Brian Goetz

Stick a fork in it (intro to Java 7 parallel extensions)

Java theory and practice

IBM developerWorks (PFDF available)

http://www.ibm.com/developerworks/java/library/j-jtp11137/index.html

http://www.ibm.com/developerworks/java/library/j-jtp03048/index.html

More Advanced Recommended Readings (not required)

Brian Goetz

Talk (about Java 7 parallel extensions)

http://www.parleys.com/#st=5&id=25&sl=2

Might also be useful for your assignment (see also handout 04)

Doug Lea

A Java Fork/Join Framework (design concepts Java 7 parallel extensions)

http://gee.cs.oswego.edu/dl/papers/fj.pdf

Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea

Java Concurrency in Practice

Addison-Wesley Professional

More Advanced Recommended Readings (not required)

Maged M. Michael, Michael L. Scott (efficient implementations, including lock-free)

Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms

http://www.cs.rochester.edu/u/scott/papers/1996_PODC_queues.pdf

Base for java.util.concurrent.ConcurrentLinkedQueue<E>

Last but not least – recommended but not required

A slightly different and more advanced view (but .NET not Java)

Joseph Albahari

C# 5.0 in a Nutshell: The Definitive Reference

O'Reilly Media

Free Chapter: Threading in C#

http://www.albahari.com/threading/

http://www.albahari.info/threading/threading.pdf

Colin Campbell, Ralph Johnson, Ade Miller, Stephen Toub

Design Patterns for Decomposition and Coordination on Multicore Architectures

http://msdn.microsoft.com/en-us/library/ff963553.aspx