Stop. Go. No, Stop!
How many ways are there to block a thread in Java?

Aidon Jennery
Senior Architect
Enterprise Content Management Solutions Development
IBM Corporation, Silicon Valley Lab
San Jose, CA, USA
What Is This All About?

- JSR166 / java.util.concurrent
  - New classes for Java 5 *(aka Java 1.5, aka Tiger)*
  - Based on Doug Lea’s concurrent classes
    - Contains both frameworks / building blocks & useful implementations
  - Overcomes several issues with previous multi-thread support *(synchronized *et. al.)*
- Multi-threaded synchronization focus
  - NOT going to cover every method of every class, or even every class, interface, *etc.*
What’s Wrong? (Java 1.4 & Prior)

- General Thread Problems
  - Fully asynchronous, cannot return a value
  - Cannot have exceptions caught
    - Unless using Thread Groups

- Synchronization Problems
  - Scope and Order
  - Too low level
  - Synchronized is not interruptible, no timeout

- Other problems too
Prerequisite Knowledge

- Previous multi-threaded programming experience with Java 1.4 (or earlier)
  - Threads, threading concepts
  - Thread state, methods, creation and destruction
  - Synchronization, signaling, semaphore concepts
  - (Understanding of collections, etc.)
- Or attend “Taking Advantage of Multi-Threading in Java”
  - by Kimberly Jennery
Agenda

- Concurrent
- Locks
- Semaphores
- Barriers
- Countdown Latches
- Collections Enhancements
- Thread Pools
- Miscellaneous

Points to think about!
Concurrent

- Merriam-Webster
  - “Operating or occurring at the same time”
  - “Running parallel”

- Wikipedia.org
  - “…concerned with the sharing of common resources between computations that are running in parallel”
  - “Generally, the tighter the coupling between computations, the more difficult the concurrency problems”
Locks

- `java.util.concurrent.locks.*`
- A mutex lock on an “entity”
- May count
- May be owned by locking thread
- May support conditions
- “Scope free” - allows for “chain locking”
- Has same memory semantics as synchronized and volatile
Lock Interfaces

- `java.util.concurrent.locks.Lock`
  - Controls access from multiple threads
- `java.util.concurrent.locks.Condition`
  - “Replaces” Object monitor methods
  - Provides for multiple “wait-sets” per object
- `java.util.concurrent.locks.ReadWriteLock`
  - A pair of locks, one for reading, one for writing, allowing for multiple readers but only one exclusive writer
ReentrantLock

- java.util.concurrent.locks.ReentrantLock
- Implementation of Lock
- A true mutex lock on an “entity”, thread “owns” entity
- A ReentrantLock counts
- Supports “fairness”
- Supports Conditions
- Interruptible
ReentrantLock Use

// Create reentrant lock
final boolean FAIRNESS = true;
ReentrantLock dataLock =
    new ReentrantLock( FAIRNESS );

// Obtain the lock and do work
dataLock.lock();
try {
    ...
    // Do stuff in try/finally
    ...
    // block just in case!
} finally {
    // Release the lock
    // dataLock.unlock();
}
Important ReentrantLock Methods

- **ReentrantLock()** - Creates an instance of ReentrantLock.
- **ReentrantLock(boolean fair)** - Creates an instance of ReentrantLock with the given fairness policy.
- **lock()** - Acquires the lock.
- **lockInterruptibly()** - Acquires the lock unless the current thread is interrupted.
- **newCondition()** - Returns a Condition instance for use with this Lock instance.
- **tryLock()** - Acquires the lock only if it is not held by another thread at the time of invocation.
- **tryLock(long timeout, TimeUnit unit)** - Acquires the lock if it is not held by another thread within the given waiting time and the current thread has not been interrupted.
- **unlock()** - Attempts to release this lock.
  ...

Conditions

- Condition Queues or Condition Variables
- Similar to wait(), notify() and notifyAll()
  - Uses await(), signal() and signalAll()
  - Lock is unlocked while awaiting
  - Lock is locked after the await returns
  - They are interruptible

```java
ReentrantLock bufferLock = new ReentrantLock( FAIRNESS );
Condition empty = bufferLock.newCondition();
Condition full = bufferLock.newCondition();
```
Condition Use

```java
object getContent() {
    bufferLock.lock();
    try {
        if (!content) full.await();
        ...
        empty.signal();
        return o;
    } finally {
        bufferLock.unlock();
    }
}

void putContent( Object o ) {
    bufferLock.lock();
    try {
        if (content) empty.await();
        ...
        full.signal();
    } finally {
        bufferLock.unlock();
    }
}
```
ReadWrite Locks

- ReadWrite locks are a very important mechanism to promote concurrency and improve performance
  - Maintained read/write structures
  - Far more reads than writes
  - Not an insignificant maintenance overhead

- Most useful for
  - Large collections, caches, etc.
ReentrantReadWriteLock

- `java.util.concurrent.locks.ReentrantReadWriteLock`
- Implementation of `ReadWriteLock`
- No acquisition order but does have “fairness”
- Writer can acquire the read lock and then “downgrade” to just the read lock
- Interruptible
- Counts
- Supports Conditions (write lock only)
ReentrantReadWriteLock Use

// Create reentrant read/write lock
ReentrantReadWriteLock tableLock =
    new ReentrantReadWriteLock( FAIRNESS );

// Write access
tableLock.writeLock().lock();
...
tableLock.writeLock().unlock();

// Read access
tableLock.readLock().lock();
...
tableLock.readLock().unlock();
Important

ReentrantReadWriteLock Methods

- **ReentrantReadWriteLock()** - Creates a new ReentrantReadWriteLock with default ordering properties.
- **ReentrantReadWriteLock(boolean fair)** - Creates a new ReentrantReadWriteLock with the given fairness policy.
- **readLock()** - Returns the lock used for reading.
- **writeLock()** - Returns the lock used for writing.
- ...
**Semaphores**

- `java.util.concurrent.Semaphore`
- Simple counting semaphore
- Allows n threads access to an “entity”
  - Threads acquire a “permit”, then release it
- No concept of ownership
  - Any thread can release a “permit” even if it has not acquired one
- Supports “fairness”
Semaphore Use

```java
// Create semaphore
final int PERMITS = 20
final boolean FAIRNESS = false;
Semaphore semaphore =
    new Semaphore( PERMITS, FAIRNESS );

// Acquire semaphore
semaphore.acquire();
...
// Release semaphore
semaphore.release();
```
Important Semaphore Methods

- **Semaphore(int permits)** - Creates a Semaphore with the given number of permits and nonfair fairness setting.
- **Semaphore(int permits, boolean fair)** - Creates a Semaphore with the given number of permits and the given fairness setting.
- **acquire(), acquire(int permits), acquireUninterruptibly(), acquireUninterruptibly(int permits)** - Acquires permit(s) in a blocking manner.
- **release(), release(int permits)** - Releases permit(s)
- **tryAcquire(), tryAcquire(int permits)** - Attempt acquisition instantly.
- **tryAcquire(int permits, long timeout, TimeUnit unit), tryAcquire(long timeout, TimeUnit unit)** - Acquire permit(s) with timeout.

...
Barriers

- `java.util.concurrent.CyclicBarrier`
- A “sync point” allowing a set of threads to ALL wait for each other at that point before ANY can continue execution
- Cyclic = reusable
- Optional “barrier action”
  - A Runnable command started after last thread arrives and before the threads are released
CyclicBarrier Use

// Create barrier
final int PARTIES = 20
CyclicBarrier barrier =
   new CyclicBarrier( PARTIES );

// Create barrier with barrier action
Runnable barrierAction = new BarrierActionCommand();
CyclicBarrier barrier =
   new CyclicBarrier( PARTIES, barrierAction );

// Wait at barrier
barrier.await();
Important CyclicBarrier Methods

- **CyclicBarrier(int parties)** - Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and does not perform a predefined action upon each barrier.

- **CyclicBarrier(int parties, Runnable barrierAction)** - Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and which will execute the given barrier action when the barrier is tripped, performed by the last thread entering the barrier.

- **await()** - Waits until all parties have invoked await on this barrier.

- **await(long timeout, TimeUnit unit)** - Waits until all parties have invoked await on this barrier.

- **reset()** - Resets the barrier to its initial state.
Countdown Latches

- `java.util.concurrent.CountDownLatch`
- A “sync point” allowing one or more threads to ALL wait until a number of operations have been performed by other threads
- Initialized with a count (the number of ops.)
- When count reaches zero awaiting threads are released
- Not reusable
CountDownLatch Use

// Create countdown latch
final int WAIT_FOR = 20
CountDownLatch counter =
    new CountDownLatch( WAIT_FOR );

// Wait for the latch
counter.await();

// Decrement the latch count
// When count goes to zero threads are unblocked
counter.countDown();
**Important CountDownLatch Methods**

- **CountDownLatch(int count)** - Constructs a CountDownLatch initialized with the given count.
- **await()** - Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted.
- **await(long timeout, TimeUnit unit)** - Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted, or the specified waiting time elapses.
- **countDown()** - Decrements the count of the latch, releasing all waiting threads if the count reaches zero.
- ...
Collections Enhancements

- Java 5 provides enhancements to the use of collections

- Language:
  - Generics, Enhanced for loop, Autoboxing

- Interfaces:
  - Queue, BlockingQueue, ConcurrentHashMap

- And a bunch of new/changed classes, some provided by java.util.concurrent
ConcurrentLinkedQueue

- `java.util.concurrent.ConcurrentLinkedQueue`
- Unbounded, FIFO, non-blocking, linked node implementation with high concurrency
- Implements Serializable, Iterable, Collection, and Queue
- The `size()` method is a time variable operation (it has to walk the list!)
// Create queue
ConcurrentLinkedQueue<String> queue =
    new ConcurrentLinkedQueue<String>();

// Put element on the queue
queue.offer( "This is from a producer" );

// Take element off the queue
String element = queue.poll();
// null returned if queue is empty
Important

ConcurrentLinkedQueue Methods

- **ConcurrentLinkedQueue()** - Creates a ConcurrentLinkedQueue that is initially empty.
- **ConcurrentLinkedQueue(Collection<? extends E> c)** - Creates a ConcurrentLinkedQueue initially containing the elements of the given collection, added in traversal order of the collection's iterator.
- **add(E o), offer(E o)** - Adds/inserts the specified element to the tail of this queue.
- **peek()** - Retrieves, but does not remove, the head of this queue, returning null if this queue is empty.
- **poll()** - Retrieves and removes the head of this queue, or null if this queue is empty.
- ...

Aidon Jennery — Stop. No, stop! How many ways are there to block a thread in Java?
### BlockingQueue Interface

- `java.util.concurrent.BlockingQueue`
- `java.util.Queue` does not block
- `BlockingQueue` maintains `Queue` methods but adds blocking methods
  - `void put(E o)` and `E take()`
- Supports bounded and unbounded queues
- Provides “draining” into other collections
**LinkedBlockingQueue**

- `java.util.concurrent.LinkedBlockingQueue`
- Based on `java.util.AbstractQueue`
- Bounded or unbounded, FIFO, blocking, linked node implementation
- Implements `Serializable`, `Iterable`, `Collection`, `Queue`, and `BlockingQueue`
LinkedBlockingQueue Use

// Create queue
final int QUEUE_SIZE = 20
LinkedBlockingQueue<String> queue =
    new LinkedBlockingQueue<String>( QUEUE_SIZE );

// Put element on the queue
queue.put( "This is from a producer" );

// Take element off the queue
String element = queue.take();
Important
LinkedBlockingQueue Methods

- `LinkedBlockingQueue()`, `LinkedBlockingQueue(int capacity)` - Creates a `LinkedBlockingQueue` with a fixed capacity of `Integer.MAX_VALUE` or the given capacity.
- `LinkedBlockingQueue(Collection<? extends E> c)` - Creates a `LinkedBlockingQueue` with a capacity of `Integer.MAX_VALUE`, initially containing the elements of the given collection, added in traversal order of the collection's iterator.
- `offer(E o)`, `offer(E o, long timeout, TimeUnit unit)` - Inserts the specified element at the tail of this queue if possible, returning immediately if this queue is full or waiting if necessary.
- `poll(long timeout, TimeUnit unit)` - Retrieves and removes the head of this queue, waiting if necessary.
- `put(E o)` - Adds the specified element to the tail of this queue, waiting if necessary for space to become available.
- `take()` - Retrieves and removes the head of this queue, waiting if no elements are present on this queue.

...
ArrayBlockingQueue

- `java.util.concurrent.ArrayBlockingQueue`
- Bounded, FIFO, blocking, array based implementation
- Implements Serializable, Iterable, Collection, Queue, and BlockingQueue
ArrayBlockingQueue Use

// Create queue
final int QUEUE_SIZE = 20
final boolean FAIRNESS = true;
ArrayBlockingQueue<String> queue =
    new ArrayBlockingQueue<String>
        ( QUEUE_SIZE, FAIRNESS );

// Put element on the queue
queue.put( "This is from a producer" );

// Take element off the queue
String element = queue.take();
Important ArrayBlockingQueue Methods

- **ArrayBlockingQueue(int capacity)**, **ArrayBlockingQueue(int capacity, boolean fair)** - Creates an ArrayBlockingQueue with the given (fixed) capacity and the supplied access policy if specified.
- **ArrayBlockingQueue(int capacity, boolean fair, Collection<? extends E> c)** - Creates an ArrayBlockingQueue with the given (fixed) capacity, the specified access policy and initially containing the elements of the given collection.
- **offer(E o)**, **offer(E o, long timeout, TimeUnit unit)** - Inserts the specified element at the tail of this queue if possible, returning immediately if this queue is full or waiting if necessary.
- **poll()**, **poll(long timeout, TimeUnit unit)** - Retrieves and removes the head of this queue, or null if this queue is empty, or waiting if necessary.
- **put(E o)** - Adds the specified element to the tail of this queue, waiting if necessary for space to become available.
- **take()** - Retrieves and removes the head of this queue, waiting if no elements are present on this queue.

...
PriorityBlockingQueue

- `java.util.concurrent.PriorityBlockingQueue`
- Unbounded, priority, blocking, heap based implementation
- Implements Serializable, Iterable, Collection, Queue, and BlockingQueue
- Uses same ordering rules as `java.util.PriorityQueue`
- Can take a Comparator
PriorityBlockingQueue Use

```java
// Create queue
final int INIT_SIZE = 20
LinkedBlockingQueue<String> queue =
    new LinkedBlockingQueue<String>( INIT_SIZE );

// Put element on the queue
queue.put( "This is from a producer" );

// Take element off the queue
String element = queue.take();
```
PriorityBlockingQueue

Constructors

- `PriorityBlockingQueue()`, `PriorityBlockingQueue(int initialCapacity)` - Creates a `PriorityBlockingQueue` with the default initial capacity (11) or the specified initial capacity that orders its elements according to their natural ordering (using `Comparable`).

- `PriorityBlockingQueue(Collection<? extends E> c)` - Creates a `PriorityBlockingQueue` containing the elements in the specified collection.

- `PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator)` - Creates a `PriorityBlockingQueue` with the specified initial capacity that orders its elements according to the specified comparator.
Important PriorityBlockingQueue Methods

- **add(E o)** - Adds the specified element to this queue.
- **offer(E o), offer(E o, long timeout, TimeUnit unit)** - Inserts the specified element into this priority queue.
- **poll(), poll(long timeout, TimeUnit unit)** - Retrieves and removes the head of this queue, or null if this queue is empty or waiting if necessary.
- **put(E o)** - Adds the specified element to this priority queue.
- **take()** - Retrieves and removes the head of this queue, waiting if no elements are present on this queue.

...
DelayQueue

- java.util.concurrent.DelayQueue
- Unbounded, blocking implementation with time based ordering of Delayed elements
- Implements Iterable, Collection, Queue, and BlockingQueue
- Elements are only obtainable after they have “expired”
- “Oldest” expired returned first
Delayed Interface

- java.util.concurrent.Delayed
- Must implement 2 methods
- long getDelay(TimeUnit unit);
  - Must return the time left until expiration in the specified units (NANOSECONDS) at the time it is called
- int compareTo(Delayed other);
  - Called by DelayQueue during insertion and extraction
DelayQueue Use

// Create queue
DelayQueue<MyDelayed> queue =
    new DelayQueue<MyDelayed>();

// Put element on the queue
queue.put( new MyDelayed( TargetNanoseconds ) );

// Take element off the queue
String element = queue.take();
Important

DelayQueue Methods

- **DelayQueue()** - Creates a new DelayQueue that is initially empty.
- **DelayQueue(Collection<? extends E> c)** - Creates a DelayQueue initially containing the elements of the given collection of Delayed instances.
- **add(E o)** - Adds the specified element to this queue.
- **offer(E o), offer(E o, long timeout, TimeUnit unit)** - Inserts the specified element into this delay queue.
- **poll(), poll(long timeout, TimeUnit unit)** - Retrieves and removes the head of this queue, or null if this queue has no elements with an unexpired delay or waiting if necessary.
- **put(E o)** - Adds the specified element to this delay queue.
- **take()** - Retrieves and removes the head of this queue, waiting if no elements with an unexpired delay are present on this queue.
- ...

...
SynchronousQueue

- java.util.concurrent.SynchronousQueue
- A blocking queue with a capacity of zero!!!
- Used for transferring an object from one thread to another in a synchronous manner
- Producers and consumers block, waiting for a consumer and producer respectively
- Optional “fairness” policy for thread ordering
SynchronousQueue Use

```java
// Create queue
SynchronousQueue<String> queue =
    new SynchronousQueue<String>();

// Put element on the queue
queue.put( "This is from a producer" );

// Take element off the queue
String element = queue.take();
```
Important

SynchronousQueue Methods

- **SynchronousQueue()** - Creates a SynchronousQueue with nonfair access policy.
- **SynchronousQueue(boolean fair)** - Creates a SynchronousQueue with specified fairness policy.
- **offer(E o), offer(E o, long timeout, TimeUnit unit)** - Inserts the specified element into this queue, if another thread is waiting to receive it, or waiting if necessary.
- **poll(), poll(long timeout, TimeUnit unit)** - Retrieves and removes the head of this queue, if another thread is currently making an element available or waiting if necessary.
- **put(E o)** - Adds the specified element to this queue, waiting if necessary for another thread to receive it.
- **take()** - Retrieves and removes the head of this queue, waiting if necessary for another thread to insert it.
- ...

Aidon Jennery — Stop. Go. No, stop! How many ways are there to block a thread in Java?
ConcurrentMap Interface

- java.util.concurrent.ConcurrentMap
- Interface based on java.util.Map interface
- Added atomic methods:
  - V putIfAbsent(K key, V value);
  - boolean remove(Object key, Object value);
  - V replace(K key, V value);
  - boolean replace(K key, V oldValue, V newValue);
ConcurrentHashMap

- `java.util.concurrent.ConcurrentHashMap`
- Hash table implementing `ConcurrentMap`
- Supports full retrieval concurrency
- Supports selectable update concurrency
  - `concurrencyLevel` used as hint for “internal sizing” (relates to number of hash buckets)
  - Updates can occur concurrently provided elements belong to different hash buckets
CopyOnWriteArray Classes

- `java.util.concurrent.CopyOnWriteArrayList`
  - Thread-safe version of `java.util.ArrayList`
  - Mutative operations cause new copy of array
  - Possibly useful when reads vastly outnumber writes, otherwise probably not

- `java.util.concurrent.CopyOnWriteArraySet`
  - An implementation of Set that uses `CopyOnWriteArrayList`
  - Similar attributes to `CopyOnWriteArrayList`
Thread Pools

- Why Thread Pools?
  - Creating a new thread for each asynchronous task is expensive
  - Need to be able to manage resources, impose limits etc

- Thread Pools are vital to scalable, high performance multi-threaded systems
  - Before Java 5, one had to roll their own or use third party package
ExecutorService Interface

- `java.util.concurrent.ExecutorService`
- Expansion of Executor interface
  - `void execute(Runnable command);`
- Provides shutdown & termination
- Provides bulk submission of tasks
- Provides for the return of values from executed tasks
- `AbstractExecutorService` implements it
ThreadExecutor

- `java.util.concurrent.ThreadPoolExecutor`
- Based on `AbstractExecutorService`
- Many configuration parameters and extensibility hooks, so the best bet is to...
- Use `java.util.concurrent.Executors` methods
  - `Executors.newCachedThreadPool();`
  - `Executors.newFixedThreadPool(int);`
  - `Executors.newSingleThreadExecutor();`
ThreadPoolExecutor Use

```java
// Create thread pool
final int THREADS = 20
ExecutorService pool =
    Executors.newFixedThreadPool( THREADS );

// Submit a Runnable task
pool.execute( new MyRunnable( param ) );

// Shutdown the pool
threadpool.shutdown();

// Wait for termination
threadpool.awaitTermination();
```
Callable & Future Interfaces

- java.util.concurrent.Callable
  - Similar to Runnable but call() returns a value

- java.util.concurrent.Future
  - Represents the result of an asynchronous task
  - Allows the result to be obtained
  - Allows querying of the task completion state
  - Allows the task to be cancelled / interrupted
  - Catches task exceptions
Callable and Future Use

// Create thread pool
final int THREADS = 20
ExecutorService pool =
    Executors.newFixedThreadPool( THREADS );

// Submit a Callable task that returns a String
Future<String> future =
    pool.submit( new MyCallable( param ) );

// Wait for termination
boolean Terminated = future.isDone();
// Get result, waits if necessary
String result = future.get();
Future Exceptions

- Future.get() can throw exceptions relating to the termination of the Callable task

  ➢ CancellationException
    • If the task was cancelled during execution

  ➢ ExecutionException
    • If the task threw an exception during execution
    • May encapsulate original exception

  ➢ InterruptedException
    • If the task was interrupted during execution
ScheduledExecutorService interface

- `java.util.concurrent.ScheduledExecutorService`
- Based on `ExecutorService`
- Schedules commands
  - To execute after a delay
  - To execute periodically
  - No absolute time execution
ScheduledThreadPoolExecutor

- `java.util.concurrent.ScheduledThreadPoolExecutor`
- Based on `ThreadPoolExecutor`
- Implements `ScheduledExecutorService`
- Delayed tasks do not execute before they are “enabled” (*i.e.* delay expired)
  - No guarantee when they will execute after they are enabled
- Tasks set to be enabled for the same execution time are enabled in FIFO order
Atomic Variables

- java.util.concurrent.atomic.*
- Lock-free and thread-safe
- Extends volatile concept to increment, decrement and compare-and-set operations, where applicable
- A must for complex, high-performance multi-threaded systems
- Not replacements for primitive type wrapper classes
Atomic Variable Classes

- Single Value Classes
  - For Boolean, Integer, Long and Reference
- Updater Classes (for use on volatile fields)
  - For Integer, Long and Reference
- Array Classes (also apply volatile to arrays)
  - For Integer, Long and Reference
- Markable and Stamped Reference Classes
  - Associate a single boolean and integer with a reference, respectively
Aidon Jennery — Stop. Go. No, stop! How many ways are there to block a thread in Java?

**Exchanger**

- `java.util.concurrent.Exchanger`
- A “sync point” at which two threads can exchange objects

```java
Exchanger<String> exchanger = new Exchanger<String>();
...
String send = “A message from this side!”
String receive = exchanger.exchange( send );

...
String send = “A message from the other side!”
String receive = exchanger.exchange( send );
```
Nanosecond Timing

- `java.lang.System.nanoTime()`
- Returns a long with the current value of the “most precise available system timer”
- Only for elapsed time
  - Up to ~ 292 years (2^63 nanoseconds)
- Nanosecond precision, but not necessarily nanosecond accuracy
- Useful for profiling and instrumentation
TimeUnit Enum

- `java.util.concurrent.TimeUnit`
- Represents time duration units
  - `NANOSECONDS, MICROSECONDS, MILLISECONDES, SECONDS`
- Utility methods to convert between units
- Most blocking methods covered here have a variant that takes a timeout

```java
// Wait 10 nanoseconds
boolean locked = lock.try(10, TimeUnit.NANOSECONDS);
```
Points to Think About

- Why do we multi-thread?
  - Simplification & Performance

- Concurrency itself is not a goal, but it is vital to good multi-threaded systems
  - Aids in the simplification of task management
  - Aids in the achievement of “performance”

- Multi-threaded != Thread-safe

- Thread-safe != Concurrency
Concurrency Building Blocks

- Multiple Threads
- Volatile keyword
- Synchronized, wait(), notify(), notifyAll()
- Locks and Conditions, Semaphores, Barriers, CountDown Latches
- Queues, Exchanger, Map, CopyOnWrite
- ThreadPools, Executors, Callable, Future
- AtomicVariables
Synchronized

- Just because access to data members is synchronized, doesn’t make it thread-safe

> “Conditional” thread-safety

```java
private int count = 0;
public synchronized int getCount(){ return count; };
public synchronized void setCount( int value )
    { count = value; };

if ( getCount() == MAX_VALUE )
    { throwMaxValueException(); } else
else
    { setCount( getCount() + 1 ); }
```

© Copyright 2005, IBM Corporation
Stopping Threads

- Thread.stop() deprecated
  - Along with Thread.suspend(), Thread.resume()
  - All are “inherently unsafe”
- Use Thread.interrupt()
  - Need to code for it
    - Catch the exception etc.
  - Will not interrupt
    - Thread waiting on synchronized method or block
    - Blocking I/O operation
Collection Scalability

- There are many choices when it comes to collections - choices become more critical when scalability (large collections) and performance are important
  - Array based
  - Link based
  - Others

- Important: different operations have different scalability profiles
Using “Fairness”

- Locks, Semaphores and Queues support the concept of “fairness”
- Basically a FIFO queuing of requesting threads
- Performance penalty for being fair
- Using “fairness”
  - Reduces variability
  - Helps avoid starvation
UP, MP & Multi-core Systems

- Never Assume Anything!!!
- While uniprocessor systems can (effectively) only do one thing at once, multi-processor and multi-core systems do multiple things at the same time, i.e. concurrently!
- Never rely on the order of thread execution, concurrency of thread execution or any other “feature” of a JVM
Debugging

- Debugging multi-threaded / concurrent problems is difficult
- Debugging timing dependent problems in a multi-threaded / concurrent system is harder
- Avoid problems with good, thorough design
- Still going to have problems
  - Use a good debugger
  - Instrument your code
Thank You!  Questions?

Email Aidon Jennery:
aidon@us.ibm.com
aidon@jennery.com