Concurrent Algorithms
Lecture 8 - Synchronisation in Java

Eugene Kenny

eugkenny.lit@gmail.com

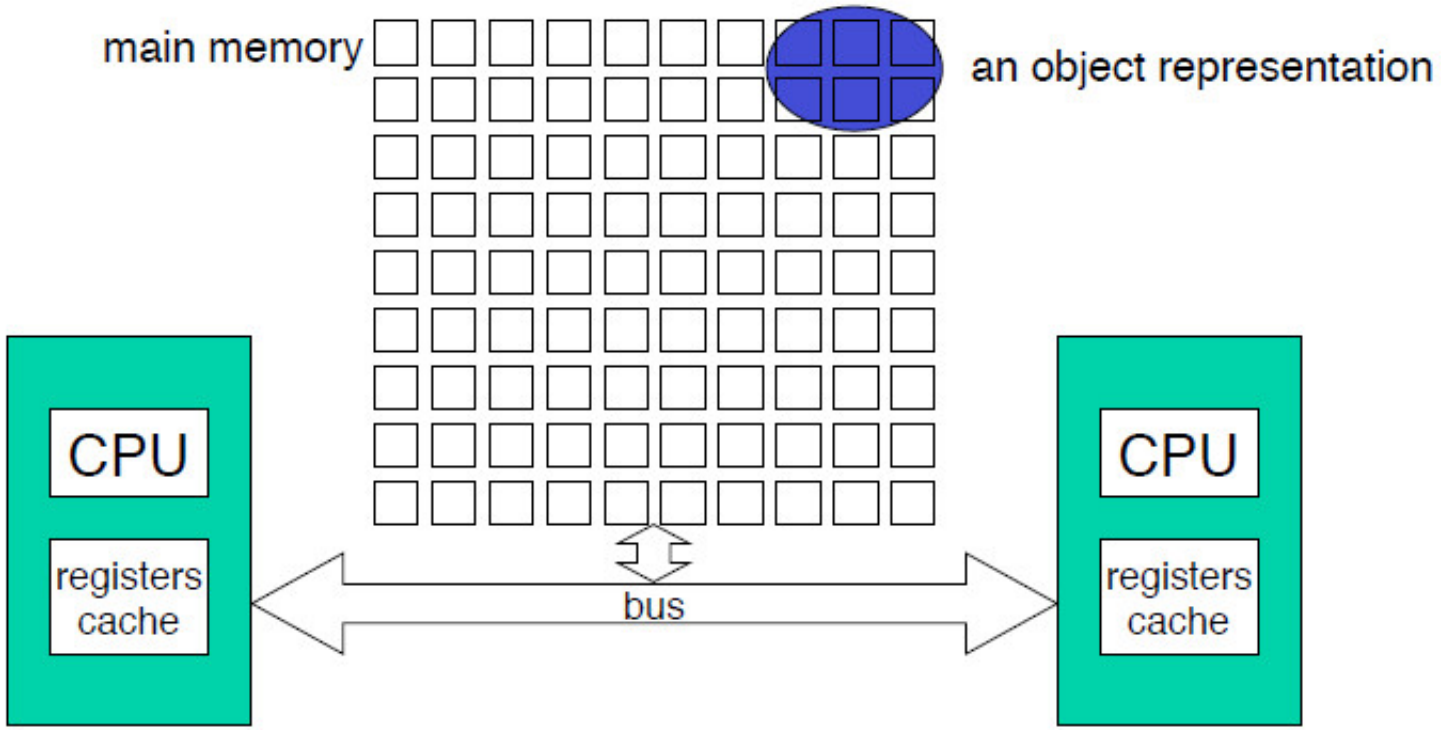
Limerick Institute of Technology

November 27th 2014

Outline

- mutual exclusion in Java
- condition synchronisation in Java
- monitors
  - example: `BoundedBuffer` monitor and Java
- problems with `synchronized`
  - example: backing off from lock attempts
  - example: non-block structured locking
  - example: read-write locks
- problems with `notifyAll()`
  - example: improving the `BoundedBuffer` solution
- deadlock revisited
- resource ordering
- acquiring multiple locks in Java
  - example: swapping values in synchronized objects

Java Memory Model



Working Memory

Java allows threads that access shared variables to keep private working copies of variables:

- each thread is defined to have a *working memory* (an abstraction of caches and registers) in which to store values;
- this allows a more efficient implementation of multiple threads

Model Properties

The Java Memory Model specifies when values must be transferred between main memory and per-thread *working memory*:

- **atomicity**: which instructions must have indivisible effects
- **visibility**: under what conditions are the effects of one thread visible to another; and
- **ordering**: under what conditions the effects of operations can appear out of order to any given thread.

Unsynchronized Code

- **atomicity**: reads and writes to memory cells corresponding to fields of any type *except* **long** or **double** are guaranteed to be atomic
- **visibility**: changes to fields made by one thread are not guaranteed to be visible to other threads
- **ordering**: from the point of view of other threads, instructions may appear to be executed out of order

volatile Fields

If a field is declared **volatile** , a thread must reconcile its working copy of the field with the master copy every time it accesses the variable.

- reads and writes to a **volatile** field are guaranteed to be atomic (even for **long**s and **double**s);
- new values are immediately propagated to other threads; and
- from the point of view of other threads, the relative ordering of operations on **volatile** fields are preserved.

However the ordering and visibility effects surround only the single read or write to the **volatile** field itself, e.g, ++ on a **volatile** field is not atomic.

Mutual Exclusion in Java

Java provides built-in support for mutual exclusion with the `synchronized` keyword:

- both methods and blocks can be `synchronized`
- each object has a *lock* (inherited from class `Object`)
- when a `synchronized` method (or block) is called, it waits to obtain the lock, executes the body of the method (or block) and then releases the lock
- allows the implementation of coarse grained atomic actions

Invoking `synchronized` Methods

When a thread invokes a `synchronized` method `foo()` on an object `x`, it tries to obtain the lock on `x`

- if another thread already holds the lock on `x`, the thread invoking `foo()` blocks
- when it obtains the lock, it executes the body of the method and then releases the lock, even if the exit occurs due to an exception
- when one thread releases a lock, another thread may acquire it (perhaps the same thread, if it invokes another `synchronized` method) – there are no guarantees about which thread will acquire a lock next or if a thread will ever acquire a lock
- locks are *reentrant*, i.e., per thread, not per invocation – a `synchronized` method can invoke another `synchronized` method on the same object without deadlocking

`synchronized` Methods

- the `synchronized` keyword is not part of a methods signature, and is not automatically inherited when subclasses override superclass methods
- methods in interfaces and class constructors cannot be declared `synchronized`
- `synchronized` instance methods in subclasses use the same lock as methods in their superclasses

`synchronized` Blocks

Block synchronization is lower-level than method synchronization:

- synchronized methods synchronize on an instance of the methods class (or the `Class` object for **static** methods)
- block synchronization allows synchronization on *any* object
- this allows us to narrow the scope of a lock to only part of the code in a method
- also allows us to use a different object to implement the lock

`synchronized` Code

- **atomicity**: changes made in one `synchronized` method (or block) are atomic with respect to other `synchronized` methods (blocks) on the *same* object
- **visibility**: changes made in one `synchronized` method (or block) are visible with respect to other `synchronized` methods (blocks) on the *same* object
- **ordering**: order of `synchronized` calls is preserved from the point of view of other threads

A Simple Example: `ParticleApplet`

`ParticleApplet` creates *n* `Particle` objects, sets each particle in autonomous continuous motion, and periodically updates the display to show their current positions:

- each `Particle` runs in its own Java `Thread` which computes the position of the particle; and
- an additional `ParticleCanvas` `Thread` periodically checks the positions of the particles and draws them on the screen.
- in this example there are at least 12 threads and possibly more, depending on how the browser handles applets.

ParticleApplet

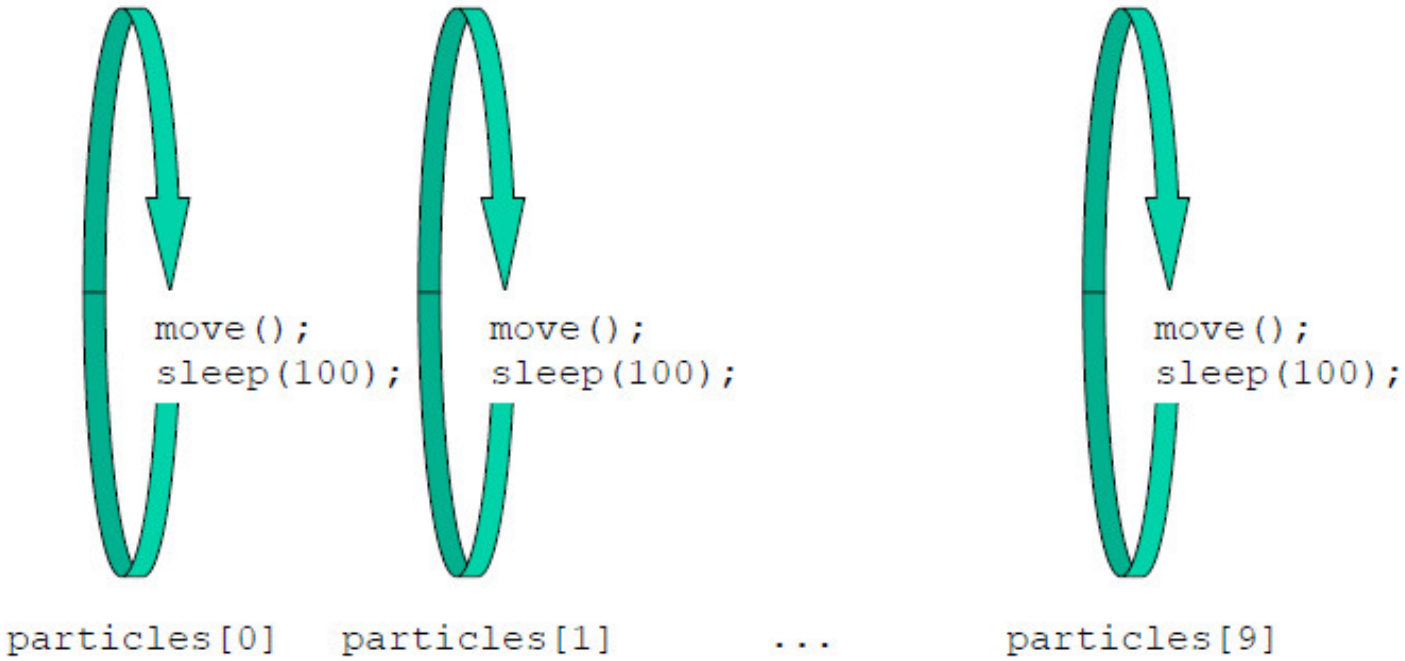
There are three classes:

- `Particle`: represents the position and behaviour of a particle and can draw the particle at its current position;
- `ParticleCanvas`: provides a drawing area for the Particles, and periodically asks the `Particles` to draw themselves; and
- `ParticleApplet`: creates the `Particles` and the canvas and sets the `Particles` in motion.

Particle.run()

```
class Particle extends Thread {
  // fields, constructor etc
  public void run() {
    try {
      for(;;) {
        move();
        sleep(100);
      }
    }
    catch (InterruptedException e) { return; }
  }
    // other methods
}
```

Particle Threads



ParticleCanvas.run()

```
class ParticleCanvas extends Canvas implements Runnable {
  // fields, constructor etc
  public void run() {
    try {
      for(;;) {
        repaint();
        Thread.sleep(100);
      }
    }
    catch (InterruptedException e) { return; }
  }
  // other methods
}
```

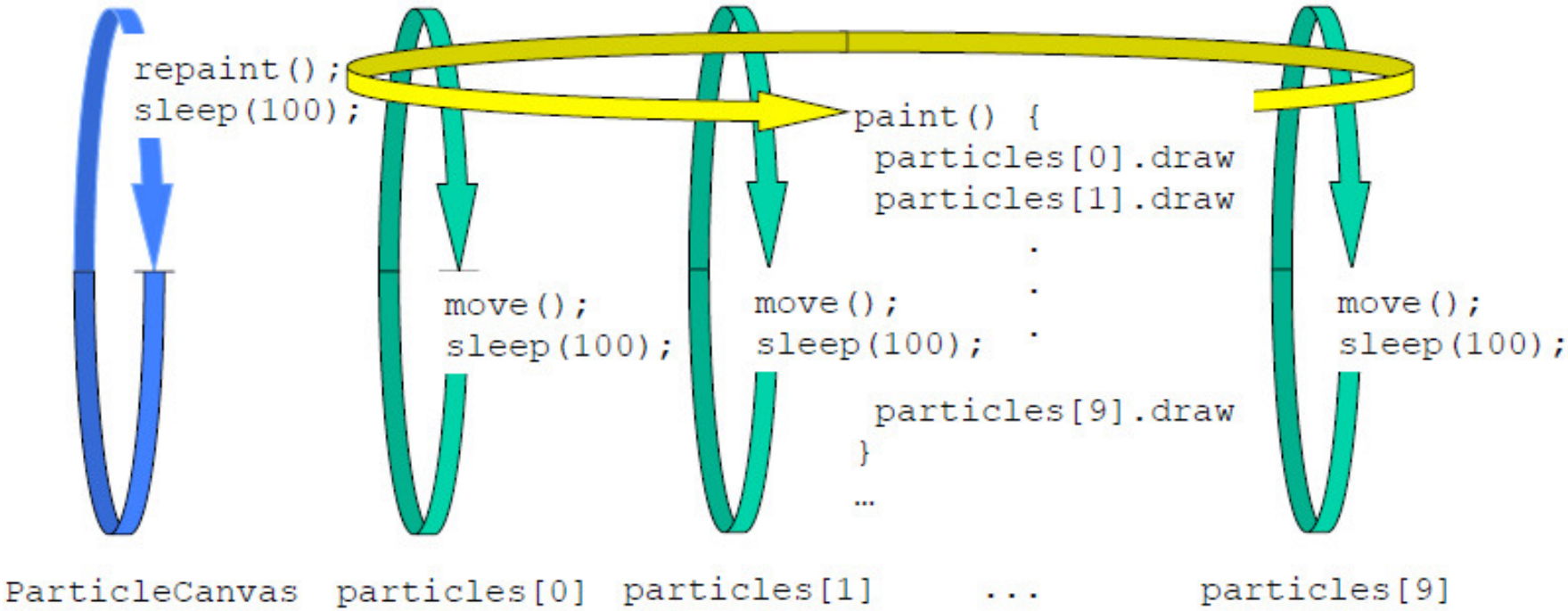
ParticleCanvas Class Continued

```
  protected synchronized void getParticles() {
    return particles;
  }

  // called by Canvas.repaint();
  public void paint(Graphics g) {
    Particle[] ps = getParticles();

    for (int i = 0; i < ps.length(); i++)
      ps[i].draw(g);
  }
}
```

ParticleCanvas & AWT Event Threads



Interference in the `ParticleApplet`

- need to ensure that `draw()` doesnt see an incompletely updated `x`, `y` position of a particle, e.g.
  - JVM runs a `Particle` thread which invokes `move()` which increments `x`
  - JVM then switches to running the `ParicleCanvas` thread which invokes `draw()`— which sees the updated `x` position but the *old* `y` position
  - JVM switches back to running the `Particle` thread – `move()` completes, updating the `y` position of the particle
- `Particle` object is drawn in a position it never occupied

Synchronising Particle.move

- we can avoid interference by making access to the x, y position of a particle a critical section
- we use synchoronized to enforce mutual exclusion

```
// Particle move method
public synchronized void move() {
  x += (rng.nextInt() % 10);
  y += (rng.nextInt() % 10);
}
```

Synchronising Particle.draw

```
// Particle draw method
public void draw(Graphics g) {
  int lx, ly;
  synchronized (this) {
    lx = x;
    ly = y;
  }
  g.drawRect(lx, ly, 10, 10);
}
```

Condition Synchronisation in Java

Java provides built-in support for condition synchronisation with the methods `wait()`, `notify()` and `notifyAll()`:

- to delay a thread until some condition is true, write a loop that causes the thread to `wait()` (block) if the delay condition is false
- ensure that every method which changes the truth value of the delay condition notifies threads waiting on the condition (using `notify()` or `notifyAll()`), causing them to wake up and re-check the delay condition.
- Java uses a Signal and Continue signalling discipline

`wait()`, `notify()` and `notifyAll()`

- `wait()`, `notify()` and `notifyAll()` must be executed within `synchronized` methods or blocks
- `wait()` releases the lock on the object held by the calling thread – the thread blocks and is added to the wait set for the object
- `notify()` wakes up *a thread* in the wait set (if any); `notifyAll()` wakes up *all threads* in the wait set (if any)
  - the thread that invoked `notify()` / `notifyAll()` continues to hold the objects lock
  - the awakened thread(s) remain blocked and execute at some future time when they can reacquire the lock

notify() vs notifyAll()

- notify() can be used to increase performance when only *one* thread needs to be woken:
  - all threads in the wait set are waiting on the *same* delay condition and
  - each notification enables at most one thread to continue and
  - the possibility of an interrupt() during notify() is handled (pre Java 5)
- notifyAll() is required when:
  - the threads in the wait set are waiting on different conditions; or
  - a notification can satisfy multiple waiting threads

Waking the Right Process

```
class TicketCounter {
  long ticket = 0;
  long turn = 0;

  synchronised takeTicket() throws InterruptedException {
    long myTurn = ticket++;
    while (myTurn != turn)
      wait();
  }

  synchronised nextTurn() {
    turn++;
    notifyAll(); // wakes all processes
  }
}
```

Loss of Notification

```
class TicketCounter { // broken - do not use ...
  long ticket = 0;
  long turn = 0;

  synchronised takeTicket() throws InterruptedException {
    long myTurn = ticket++;
    while (myTurn != turn)
      wait();
  }

  synchronised nextTurn() {
    turn++;
    notify(); // wakes an arbitrary process
  }
}
```

Interrupting a Thread

Each Thread object has an associated boolean interruption status:

- `interrupt()`: sets the thread's interrupted status to *true*
- `isInterrupted()`: returns *true* if the thread has been interrupted by `interrupt()`

A thread can periodically check its interrupted status, and if it is *true*, clean up and exit.

Handling Interrupts

Threads which are blocked in calls `wait()`, `sleep()` and `join()` aren't runnable, and can't check the value of the interrupted flag

- interrupting a thread which is not runnable aborts the thread, throws an `InterruptedException` and sets the thread's interrupted status to *false*
- calls to `wait()`, `sleep()`, or `join()` are often enclosed in a **try catch** block:

```
synchronized <method or block>
  try {
    wait() | sleep(millis) | join(millis)
  } catch (InterruptedException e) {
    // clean up and return
  }
```

Approaches to Mutual Exclusion

- **Mutual exclusion algorithms**: pre- and post-protocols are implemented using special machine instructions or atomic memory accesses (e.g., Test-and-Set, Peterson's algorithm)
- **Semaphores**: pre- and post-protocols can be implemented using atomic **P** and **V** operations
- **Monitors**: mutual exclusion is *implicit* – pre- and post protocols are executed automatically on entering and leaving the monitor to ensure that monitor procedures are not executed concurrently.

Implementing Mutual Exclusion in Java

- it is difficult to implement mutual exclusion algorithms in Java, due to problems of visibility, ordering, scheduling and efficiency
- we can implement semaphores as a Java class with methods which implement the **P** and **V** operations (see `java.util.concurrent`)
- monitors are the basis of Javas synchronisation primitives – there is a straightforward mapping from designs based on monitors to solutions using synchronized classes

Approaches to Condition Synchronisation

- **Busy-waiting**: the process sits in a loop until the condition is true
- **Semaphores**: **P** and **V** operations can be used to wait for a condition and to signal that it has occurred
- **Monitors**: condition synchronisation is explicitly programmed using *condition variables* and monitor operations, e.g., wait and signal.

Implementing Condition Synchronisation in Java

- it is difficult to implement busy waiting in Java, due to problems of visibility, scheduling and efficiency
- we can implement semaphores as a Java class with methods which implement the **P** and **V** operations (see `java.util.concurrent`)
- while monitors are the basis of Javas synchronisation primitives, each object in Java has a only a single condition variable and delay queue – monitor operations can be implemented using `wait()` and `notify()`, and `notifyAll()`

Example: Monitors

A *monitor* is an abstract data type representing a shared resource. Monitors have four components:

- a set of *private variables* which represent the state of the resource;
- a set of *monitor procedures* which provide the public interface to the resource;
- a set of *condition variables* and associated *monitor operations* (`wait()`, `signal()`, `signal_all()`) used to implement condition synchronisation; and
- *initialisation code* which initialises the private variables.

Monitors in Java

Monitors can be implemented as Java classes:

- the monitor's private variables are **private** fields in a class
- the monitor procedures are implemented using `synchronized` methods – `synchronized` methods are executed under mutual exclusion with all other `synchronized` methods on the same object
- condition synchronisation is implemented using `wait()`, `notify()`, `notifyAll()`
  - each object has a single (implicit) condition variable and delay queue, the *wait set*
  - Java uses a *signal and continue* signalling discipline

Bounded Buffer with Monitors

```
monitor BoundedBuffer {
    // Private variables …
    Object buf = new Object[n];
    integer out = 0,        // index of first full slot
            in = 0,         // index of first empty slot
            count = 0;      // number of full slots

    // Condition variables ...
    condvar not_full,       // signalled when count < n
            not_empty;      // signalled when count > 0

    // continued ...
```

Bounded Buffer with Monitors

```
// Monitor procedures ...
//(signal & continue signalling discipline)
procedure append(Object data) {
    while(count == n) {
        wait(not_full);
}
buf[in] = data;
in = (in + 1) % n;
count++;
signal(not_empty);
}

// continued ...
```

Bounded Buffer with Monitors

```
        procedure remove(Object &item) {
            while(count == 0) {
                wait(not_empty);
            }
            item = buf[out];
            out = (out + 1) %n;
            count--;
            signal(not_full);
        }
}
```

Bounded Buffer in Java

```
class BoundedBuffer {
  // Private variables
  private Object buf;
  private int out = 0, // index of first full slot
  private int in = 0, // index of first empty slot
  private int count = 0; // number of full slots

  public BoundedBuffer(int n) {
    buf = new Object[n];
  }

  // continued ...
```

Bounded Buffer in Java

```
// Monitor procedures
public synchronized void append(Object data) {
  try {
    while(count == n)
      wait();
    }
  catch (InterruptedException e) {
    return;
  }
  buf[in] = data;
  in = (in + 1) % n;
  count++;
  notifyAll();
}
```

Bounded Buffer in Java

```
  public synchronized Object remove() {
    try {
      while(count == 0) {
        wait();
    }
    catch (InterruptedException e) {
      return null;
    }
    Object item = buf[out];
    out = (out + 1) % n;
    count--;
    notifyAll();
    return item;
  }
}
```

Fully Synchronised Objects

The safest (but not necessarily the best) design strategy based on mutual exclusion is to use *fully synchronized objects* in which:

- all methods are `synchronized`
- there are no public fields or other encapsulation violations
- all methods are finite (i.e., no infinite loops after acquiring a lock)
- all fields are initialised to a consistent state in constructors
- the state of the object is consistent at both the beginning and end of each method (even in the presence of exceptions).

Synchronization Wrappers

- you can turn a `Collection` into a fully synchronized object using a *synchronization wrapper*
- each of the core collection interfaces, `Collection`, `Set`, `List`, `Map`, `SortedSet`, and `SortedMap`, has a static factory method

```
public static <T> Collection<T> synchronizedCollection(Collection
    <T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T>
     s);
public static <K,V> SortedMap<K,V> synchronizedSortedMap(
    SortedMap<K,V> m);
```

Ensuring Synchronization

- the factory methods returns a synchronized `Collection` backed by the collection passed as argument
- to ensure mutual exclusion, *all* access to the backing collection must be through the synchronized collection
- simplest approach is not to keep a reference to the backing collection, e.g.:

```
List<T> list = Collections.synchronizedList(new ArrayList<
    T>());
```

- note that wrappers only synchronize interface methods

Wrappers and Iterators

- a synchronized wrapper does not make a collection thread safe for iteration
- iteration involves multiple calls into the collection, and you must manually synchronize on the wrapped collection when iterating over it, e.g.:

```
List<T> list = Collections.synchronizedList(new ArrayList<
    T>());
synchronized(list) {
  for (T e : list)
    foo(e);
}
```

- note that when iterating over a `Collection` view of a synchronized `Map` you should synchronize on the `Map` *not* the `Collection` view

Problems with `synchronized`

`synchronized` methods and blocks have a number of limitations:

- there is no way to back off from an attempt to acquire a lock, e.g., to timeout or cancel a lock attempt following an `interrupt`
- synchronisation within methods and blocks limits use to strict block structured locking
- there is no way to alter the semantics of a lock, e.g., read vs write protection

One way these problems can be overcome is by using *utility classes* to control locking.

Problems with `synchronized`

`synchronized` methods and blocks have a number of limitations:

- there is no way to back off from an attempt to acquire a lock, e.g., to timeout or cancel a lock attempt following an `interrupt`
- synchronisation within methods and blocks limits use to strict block structured locking
- there is no way to alter the semantics of a lock, e.g., read vs write protection

One way these problems can be overcome is by using *utility classes* to control locking.

Example: Semaphores

A semaphore s is an integer variable which can take only non-negative values. Once it has been given its initial value, the only permissible operations on s are the atomic actions:

> $P(s)$: if $s > 0$ then $s = s1$, else suspend execution of the process that called $P(s)$
>
> $V(s)$: if some process p is suspended by a previous $P(s)$ on this semaphore then resume p, else $s = s + 1$

A *general semaphore* can have any non-negative value; a *binary semaphore* is one whose value is always 0 or 1.

Example: Semaphores in Java

```
class GeneralSemaphore {
  private long resource;

  public GeneralSemaphore (long r) {
    resource = r;
  }

  // method to implement the P operation

  // method to implement the V operation
}
```

Example: Semaphores in Java

```
// Sample implementation  not the way its done in
// java.util.concurrent.Semaphore
class GeneralSemaphore {
  private long resource;

  public GeneralSemaphore (long r) {
    resource = r;
  }

  public synchronized void V() {
    ++resource;
    notify();
  }
```

Example: Semaphores in Java

```
    public synchronized void P() throws InterruptedException {
      while (resource <= 0)
        wait();
      --resource;
    }
}
```

Example: GeneralSemaphore

The `GeneralSemaphore` class is fully synchronized:

- when the `P()` method is invoked on an instance of the `GeneralSemaphore` class, `s`, the invoking thread attempts to obtain the lock on `s`
- there is no way to back off if the lock is already held by another thread, to give up after waiting for a specified time, or to cancel the lock attempt following an interrupt
- this can make it difficult to recover from liveness problems.

Semaphores in Java 2

```
  public void P() throws InterruptedException {
    if (Thread.interrupted())
      throw new InterruptedException();
    synchronized(this) {
      while (resource <= 0)
        wait();
      --resource;
    }
  }
}
```

Handling Interrupts

Threads should periodically check their interrupt status, and if interrupted, shut down:

- a good place to check is before calling a `synchronized` method, e.g.:

```
// other code ...
s.P();
```

  as we may spend a long time contending for the lock on `s`
- this can result in threads being unresponsive to interrupts

Backing Off from Lock Attempts

Even if we check for interrupts before attempting to acquire a lock on a `synchronized` method or block

- a thread which is trying to acquire the lock must still be prepared to wait indefinitely
- deadlocks are fatal – the only way to recover is to restart the application
- however, we can implement more flexible locking protocols using utility classes

The Lock Interface

java.util.concurrent defines a Lock interface and a number of utility classes (e.g., ReentrantLock) which implement the interface:

```
public interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long timeout, TimeUnit unit) throws
      InterruptedException;
  void unlock();
  Condtion newCondition();
}
```

Replacing synchronized Blocks

A Lock can be used to replace blocks of the form:

```
synchronized(this) { /* body */}
```

with a before/after construction, e.g.:

```
Lock lock = new ReentrantLock();
lock.lock();
try {
  // body: catch & handle exceptions if necessary
} finally {
  lock.unlock();
}
```

Backing Off from Lock Attempts with `Lock`

Unlike `synchronzied`, the `Lock` interface supports:

- *polled* lock acquisition: `tryLock()` allows control to be regained if all the required locks cant be acquired
- *timed* lock acquisition: `tryLock(timeout)` allows control to be regained if the time available for an operation runs out
- *interruptible* lock acquisition: `lockInterruptibly` allows an attempt to aquire a lock to be interrupted

Problems with `synchronized`

`synchronized` methods and blocks have a number of limitations:

- there is no way to back off from an attempt to acquire a lock, e.g., to timeout or cancel a lock attempt following an `interrupt`
- synchronisation within methods and blocks limits use to strict block structured locking
- there is no way to alter the semantics of a lock, e.g., read vs write protection

One way these problems can be overcome is by using *utility classes* to control locking.

Locking is Block Structured

- `synchronized` methods and blocks limits use to strict block structured locking:
  - a lock is always released in the same block as it was acquired, regardless of how control exits the block
  - e.g., a lock can't be acquired in one method or block and released in another
- this prevents potential coding errors, but can be inflexible
- again we can use utility classes to implement non-block stuctured locking

Example: `ListUsingLocks`

- for example, we can use `Lock` objects to lock the nodes of linked list during operations that traverse the list
- the lock for the next node must be obtained while the lock for the current node is still being held
- after acquiring the next lock, the current lock is released
- this allows extremely fine-grained locking and increases potential concurrency
- only worthwhile in situations where there is a lot of contention.

Example: ListUsingLocks

```
class ListUsingLocks {
  static class Node {
    Object item;
    Node next;
    Lock lock = new ReentrantLock();
    Node(Object x, Node n) { item = x; next = n; }
  }

  protected Node head;

  protected synchronized Node getHead() { return head; }

  public synchronized add(Object x) {
    head = new Node(x, head);
  }
```

Example: ListUsingLocks

```
  boolean search(Object x) throws InterruptedException {
    Node p = getHead();
    if (x == null || p == null) return false;
    Node nextp;
    p.lock.lock();
    for (;;) {
      try {
        if (x.equals(p.item)) return true;
        if ((nextp = p.next()) == null) return false;
        nextp.lock.lock();
      } finally {
        p.lock.unlock();
      }
      p = nextp;
    }
} // other methods omitted ...
```

ListUsingSynchronized

```
// Broken, do not use ...
boolean search(Object x) throws InterruptedException {
  Node p = getHead();
  if (x == null || p == null) return false;

  Node nextp;
  for (;;) {
    synchronized(p) {
      if (x.equals(p.item)) return true;
      if ((nextp = p.next()) == null) return false;
      synchronized(nextp) {
        // cant release the lock on p here ...
      } // lock on nextp will be released here
      p = nextp;
    } // lock on p will be released here ...
  }
}
```

Problems with `synchronized`

`synchronized` methods and blocks have a number of limitations:

- there is no way to back off from an attempt to acquire a lock, e.g., to timeout or cancel a lock attempt following an `interrupt`
- synchronisation within methods and blocks limits use to strict block structured locking
- there is no way to alter the semantics of a lock, e.g., read vs write protection

One way these problems can be overcome is by using *utility classes* to control locking.

Altering the Semantics of a Lock

With `synchronized` there is no way to alter the semantics of a lock, e.g., read vs write protection

- this makes it difficult to solve *selective mutual exclusion* problems, like the Readers and Writers problem
- again, these problems can be overcome is by using utility classes to control locking

The ReadWriteLock Interface

- java.util.concurrent defines a ReadWriteLock interface and a number of utility classes (e.g., ReentrantReadWriteLock) which implement the interface:

```
public interface ReadWriteLock {
  Lock readLock(); // returns the read lock
  Lock writeLock(); // returns the write lock
}
```

ReadWriteLocks

A `ReadWriteLock` maintains a pair of associated Locks, one for readonly operations and one for writing:

- the `readLock` may be held simultaneously by multiple reader threads, so long as there are no writers
- the `writeLock` is exclusive
- since the `readLock` and `writeLock` implement the `Lock` interface, they support polled, timed and interruptible locking

Read-Write Locks

A read-write lock can allow for a greater level of concurrency in accessing shared data than that permitted by a mutual exclusion lock if:

- the methods in a class can be separated into those that only read internally held data and those that read and write
- reading is not permitted while writing methods are executing
- the application has more readers than writers
- the methods are time consuming, so it pays to introduce more overhead in order to allow concurrency among reader threads
- e.g, in accessing a dictionary which is frequently read but seldom modified

Example RWDictionary

```
class RWDictionary {
  private final Map<String, Data> m = new TreeMap<String, Data>()
      ;
  private final ReentrantReadWriteLock rwl = new
      ReentrantReadWriteLock();

  private final Lock r = rwl.readLock();
  private final Lock w = rwl.writeLock();

  // methods follow ...
```

Example RWDictionary Readers

```
// Reader method (does not update the map)
public Data get(String key) {
  r.lock();
  try { return m.get(key); }
  finally { r.unlock(); }
}
```

Example RWDictionary Writers

```
    // Writer method (changes the map)
    public Data put(String key, Data value) {
      w.lock();
      try { return m.put(key, value); }
      finally { w.unlock(); }
    }
}
```

Mutual Exclusion Summary

- all the synchronisation primitives we have looked at are equivalent in the sense that they all have the same expressive power
- while it is often helpful to take advantage of the higher level of abstraction offered by monitors, there are situations when other forms of synchronisation are required and we can implement any of these using any of the primitives.
- more complex forms of locking can and are defined in terms of primitives like `Lock`.
- at each level of abstraction we see this pattern of acquiring and releasing locks.

Condition Synchronisation in Java

Condition Synchronisation can be implemented using the methods `wait()`, `notify()` and `notifyAll()`:

- to delay a thread until some condition is true, write a loop that causes the thread to `wait()` (block) if the delay condition is false
- ensure that every method which changes the truth value of the delay condition notifies threads waiting on the condition (using `notify()` or `notifyAll()`), causing them to wake up and re-check the delay condition.

Context Switching in Java

When a thread blocks and/or another is scheduled, the JVM must perform a *context switch*:

- this involves saving the registers of the suspended thread and loading the registers of the newly scheduled thread
- which takes time
- a concurrent program, runs faster if we can reduce the number of context switches.

Condition Variables in Java

In Java, each object has a single implicit condition variable:

- a `notifyAll()` intended to inform threads about one condition also wakes up threads waiting for unrelated conditions, resulting in large numbers of context switches
- context switching can be minimised by delegating operations with different `wait()` conditions to different *helper objects*
- such helper objects serve as *condition variables* – places to put threads that need to wait on and be notified of a particular condition

Bounded Buffer in Java

```
class BoundedBuffer {
  // Private variables
  Object[] buf;
  int out = 0, // index of first full slot
  int in = 0, // index of first empty slot
  int count = 0; // number of full slots

  public BoundedBuffer(int n) {
    buf = new Object[n];
  }

  // continued ...
```

Bounded Buffer in Java

```
// Monitor procedures
public synchronized void append(Object data) {
  try {
    while(count == n) {
      wait();
    }
  catch (InterruptedException e) {
    return;
  }
  buf[in] = data;
  in = (in + 1) % n;
  count++;
  notifyAll();
}
```

Bounded Buffer in Java

```
  public synchronized Object remove() {
    try {
      while(count == 0) {
        wait();
      }
    catch (InterruptedException e) {
      return null;
    }
    Object item = buf[out];
    out = (out + 1) % n;
    count--;
    notifyAll();
    return item;
  }
}
```

Problems with the `BoundedBuffer`

- there are two different conditions a thread may be waiting on:
  - the buffer being not full (producer threads)
  - the buffer being not empty (consumer threads)
- but a `BoundedBuffer` object has only one wait set, so we must use `notifyAll()`
- e.g., if the buffer is full, a consumer taking one item will wake all waiting producers, even though only one can proceed

Bounded buffer with Semaphores

```
class BoundedBufferWithSemaphores {
  // Private variables
  BufferArray buf; // defined later ...
  GeneralSemaphore empty;
  GeneralSemaphore full;

  public BoundedBufferWithSemaphores(int n) {
    buf = new BufferArray(n);
    empty = new GeneralSemaphore(n);
    full = new GeneralSemaphore(0);
  }

// continued ...
```

Bounded buffer with Semaphores

```
    public void append(Object data) throws InterruptedException {
      empty.P();
      buff.append(data);
      full.V();
    }

    public Object remove() throws InterruptedException {
      full.P();
      Object data = buff.remove();
      empty.V();
    }
}
```

Bounded Buffer with Semaphores

- operations with different `wait()` conditions are delegated to different *helper objects* – the `GeneralSemaphores`
- underlying array operations are isolated in a simple `BufferArray` class
- `BufferArray` uses `synchronized` methods to ensure mutually exclusive access to the underlying array
- only one thread can access the buffer at a time

Bounded Buffer with Semaphores

```
class BufferArray {
  Object[] array; int in = 0; int out = 0;

  BufferArray(int n) { array = new Object[n]; }

  synchronized void append(Object data) {
    array[in] = data;
    in = (in + 1) % array.length;
  }

  synchronized Object remove() {
    Object data = array[out];
    array[out] = null;
    out = (out + 1) % array.length;
    return data;
  }
}
```

Quadratic to Linear

- `BoundedBufferWithSemaphores` is likely to run more efficiently than the `BoundedBuffer` class when many threads are using the buffer
- it uses two different underlying wait sets
- the semaphores only wake one thread on each operation, eliminating the unnecessary context switching caused by using `notifyAll()` instead of `notify()`
- this reduces the worst case number of wakeups from a quadratic function of the number of invocations to linear

The Condition Interface

Condition factors out the Object condition synchronisation methods (wait, notify and notifyAll) into distinct objects to give the effect of having multiple wait-sets per object

```
public interface Condition {
  // Key methods only ...
  void await() throws InterruptedException
  void signal()
  void signalAll()
}
```

- the utility classes ReentrantLock, ReentrantReadWriteLock, etc. implement the Condition interface

Condition Methods

- **void** `await()`: causes the invoking thread to wait until it is signalled or interrupted
- **boolean** `await(`**long** `time, TimeUnit unit)`: causes the invoking thread to wait until it is signalled or interrupted, or the specified waiting time elapses
- **void** `awaitUninterruptibly()`: causes the invoking thread to wait until it is signalled
- **void** `signal()`: wakes one thread waiting on the condition
- **void** `signalAll()`: wakes all threads waiting on the condition

The Condition Interface

- because access to the shared condition occurs in different threads, it must be protected by a `Lock`
- each `Condition` instance is intrinsically bound to a `Lock` – to obtain a `Condition` instance for a particular `Lock` instance use its `newCondition()` method.
- waiting for a `Condition` atomically releases the associated lock and suspends the current thread, just like `Object.wait()`
- supports interruptible, non-interruptible, and timed waits

Example: Bounded Buffer with Conditions

```
class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  Object[] buf;
  int out = 0, // index of first full slot
  int in = 0, // index of first empty slot
  int count = 0; // number of full slots

  public BoundedBuffer(int n) {
    buf = new Object[n];
  }

  // continued ...
```

Example: Bounded Buffer with Conditions

```
public void append(Object data) throws InterruptedException {
  lock.lock();
  try {
    while (count == items.length)
      notFull.await();
    buf[in] = data;
    in = (in + 1) % n;
    count++;
    notEmpty.signal();
  } finally {
    lock.unlock();
  }
}
```

Example: Bounded Buffer with Conditions

```
  public Object remove() throws InterruptedException {
    lock.lock();
    try {
      while (count == 0)
        notEmpty.await();
      Object item = buf[out];
      out = (out + 1) % n;
      count--;
      notFull.signal();
      return item;
    } finally {
      lock.unlock();
    }
  }
}
```

Other `java.util.concurrent` Utility Classes

- utility classes implementing the `Lock` and `Condition` interfaces are used internally in the implementation of other classes in `java.util.concurrent`
- e.g., the `BlockingQueue<E>` interface defines a `Queue` with operations that:
  - wait for the queue to become non-empty when retrieving an element
  - wait for space to become available in the queue when storing an element
- if inserting, removing or examining an element can't proceed, methods may either: throw an exception, return a special value, block, or timeout

The Class ArrayBlockingQueue<E>

`ArrayBlockingQueue` implements a bounded buffer backed by a fixed-sized array

- attempts to put an element into a full queue block;
- attempts to take an element from an empty queue also block;
- supports an optional *fairness policy* for ordering waiting producer and consumer threads – a queue constructed with *fairness* set to true grants threads access in FIFO order;
- fairness generally decreases throughput but reduces variability and avoids starvation.

Condition Synchronisation Summary

- simple condition synchronisation can be implemented using the methods `wait()`, `notify()` and `notifyAll()`
- however a single wait set can be inefficient if threads wait on different conditions
- context switching can be minimised by delegating operations with different `wait()` conditions to different *helper objects*
- `Condition` interface makes it clear that an object is being used as a condition variable, and allows interruptible and timed waits

Multiple Locks

- utility classes implementing `Lock` and `Condition` allow finer-grained locking
- used correctly, this can increase concurrency/reduce latency
- however problems can arise when threads must acquire *multiple* locks
- a particular problem is *deadlock*
- e.g.: two threads must acquire locks on two objects, get one lock each, and wait forever for each other to release the other lock.

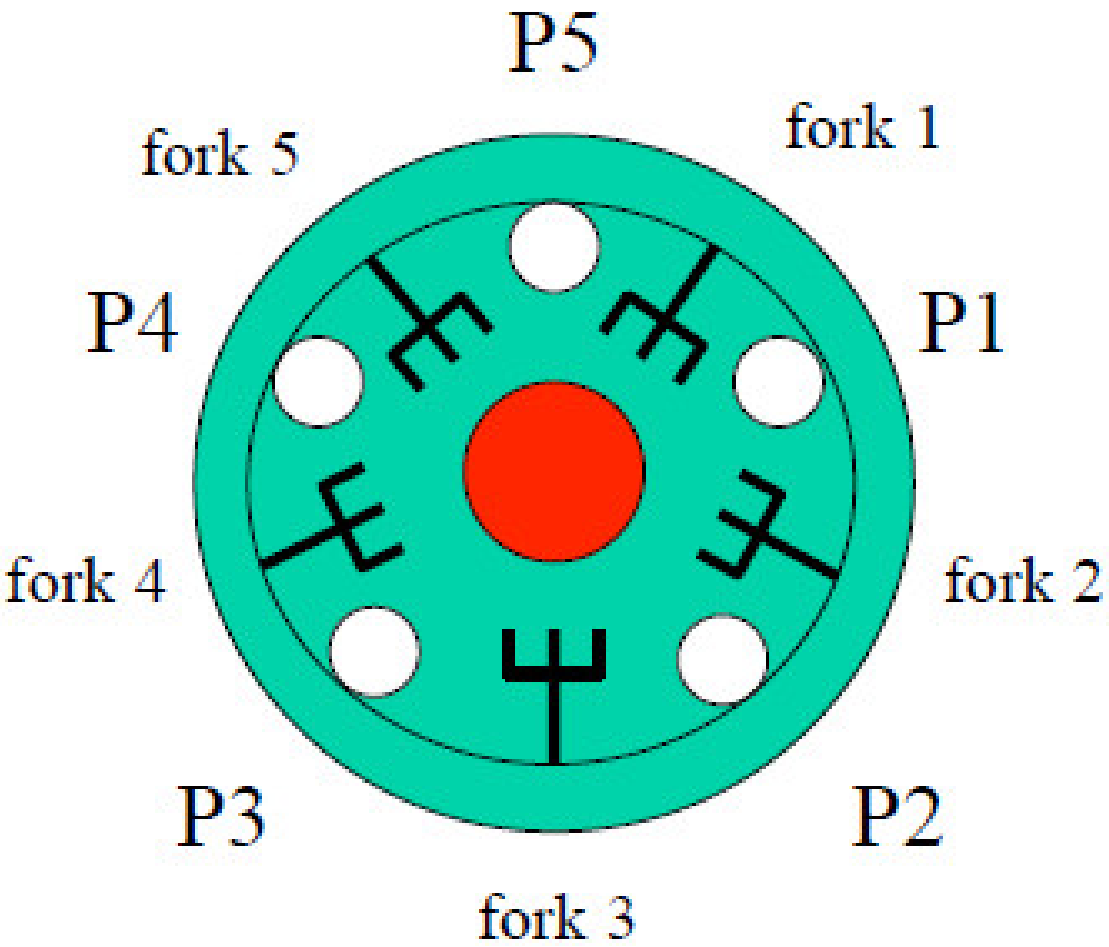
Dining Philosophers Problem

The *Dining Philosophers* problem illustrates mutual exclusion between processes which compete for overlapping sets of shared variables

- five philosophers sit around a circular table
- each philosopher alternately thinks and eats spaghetti from a dish in the middle of the table
- the philosophers can only afford five forks – one fork is placed between each pair of philosophers
- to eat, a philosopher needs to obtain mutually exclusive access to the fork on their left and right

The problem is to avoid *starvation* – e.g., each philosopher acquires one fork and refuses to give it up.

Dining Philosophers Problem



Deadlock in the Dining Philosophers

The key to the solution is to avoid deadlock caused by circular waiting:

- process 1 is waiting for a resource (fork) held by process 2
- process 2 is waiting for a resource held by process 3
- process 3 is waiting for a resource held by process 4
- process 4 is waiting for a resource held by process 5
- process 5 is waiting for a resource held by process 1.

No process can make progress and all processes remain deadlocked.

Semaphore Solution

```
// Philosopher i, i == 1-4
while(true) {
    //get right fork then left
    P(fork[i]);
    P(fork[i+1]);
    // eat ...
    V(fork[i]);
    V(fork[i+1]);
    // think ...
}
```

```
// Philosopher 5
while(true) {
    //get left fork then right
    P(fork[1]);
    P(fork[5]);
    // eat ...
    V(fork[1]);
    V(fork[5]);
    // think ...
}
```

```
// Shared variables
binary semaphore fork[5] = {1, 1, 1, 1, 1};
```

Deadlock

Although fully synchronised objects are always safe, threads using them are not always live

- some `synchronized` actions are multiparty – they acquire locks on multiple objects
- *deadlock* is possible when two or more objects are mutually accessible from two or more threads, and each thread holds one lock while trying to obtain another lock held by another thread

Example: Cell

```
class Cell { // Broken, do not use ...
  private long value;

  synchronized long getValue() { return value; }

  synchronized void setValue(long v) { value = v; }

  synchronized void swapValue(Cell other) {
    long t = getValue();
    long v = other.getValue();
    setValue(v);
    other.setValue(t);
  }
}
```

Example Deadlock Trace

Consider two threads, one of which invokes `a.swapValue(b)` while the other invokes `b.swapValue(a)`

Thread 1	Thread 2
acquire lock for `a` on invoking `a.swapValue(b)`	
pass the lock for `a` (since already held) on invoking `t = getValue()`	acquire lock for `b` on invoking `b.swapValue(a)`
block waiting for lock on `b` on invoking `v = other.getValue()`	pass lock for `b` (since already held) on invoking `t = getValue()`
	block waiting for lock on `a` on invoking `v = other.getValue()`

Resource Ordering

One way to avoid this kind of deadlock is to use *resource ordering*:

- associate a numerical (or any other strictly orderable data type) *tag* with each object that can be an argument to a `synchronized` multiparty action
- if synchronization is always performed in *tag order*, then a situation can never arise in which a thread which has a lock on object `x` and is waiting for a lock on `y` while another thread has a lock on `y` and is waiting for a lock on `x`
- whichever thread locks the resource with the lowest tag first will acquire both locks while the other waits, and then the second thread will acquire both locks

Example: Resource Ordering

- in the case of `Cells`, we can either extend the `Cell` class to add a tag field
- or we can use some existing information about `Cell` objects, e.g., their hash codes
- e.g., we can use `System.identityHashCode` which returns the hash value computed by `Object.hashCode` (even if a class overrides the `hashCode` method)
- while the `identityHashCode` value is not guaranteed to be unique, it is very likely to be unique, which is often good enough

Example: swapValue()

```
public void swapValue(Cell other) {
  if (System.identityHashCode(this) <
      System.identityHashCode(other))
      this.doSwapValue(other);
  else
    other.doSwapValue(this);
}

protected synchronized void doSwapValue(Cell other) {
  long t = getValue();
  long v = other.getValue();
  setValue(v);
  other.setValue(t);
}
```