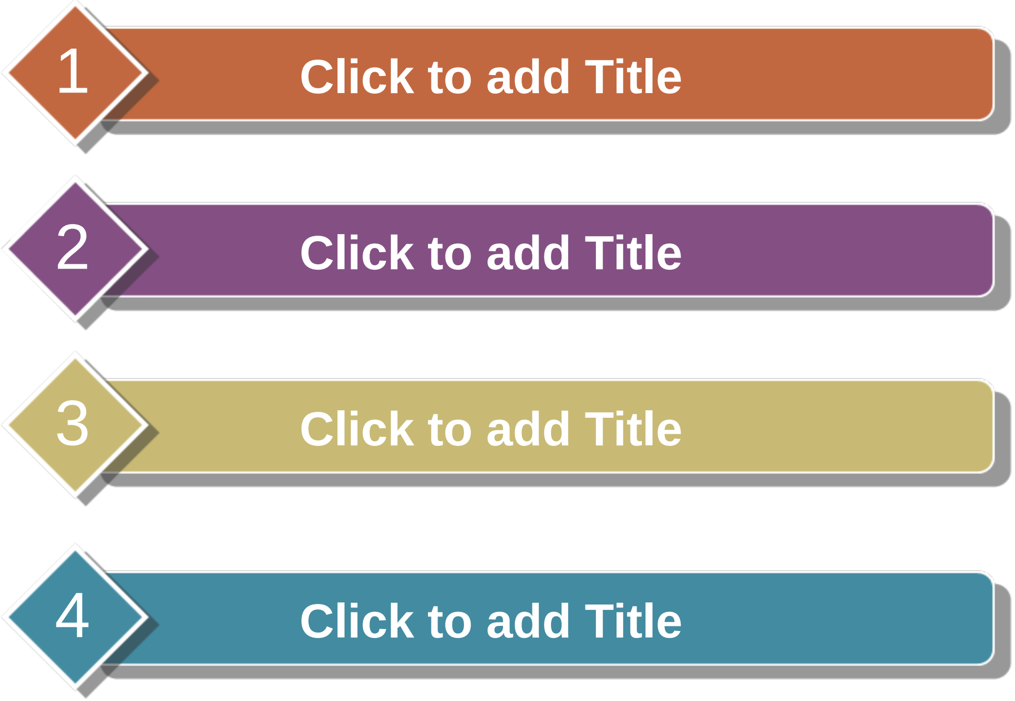
# Concurrent Data Structures

By Mohit Kumar

# Contents



1 Click to add Title

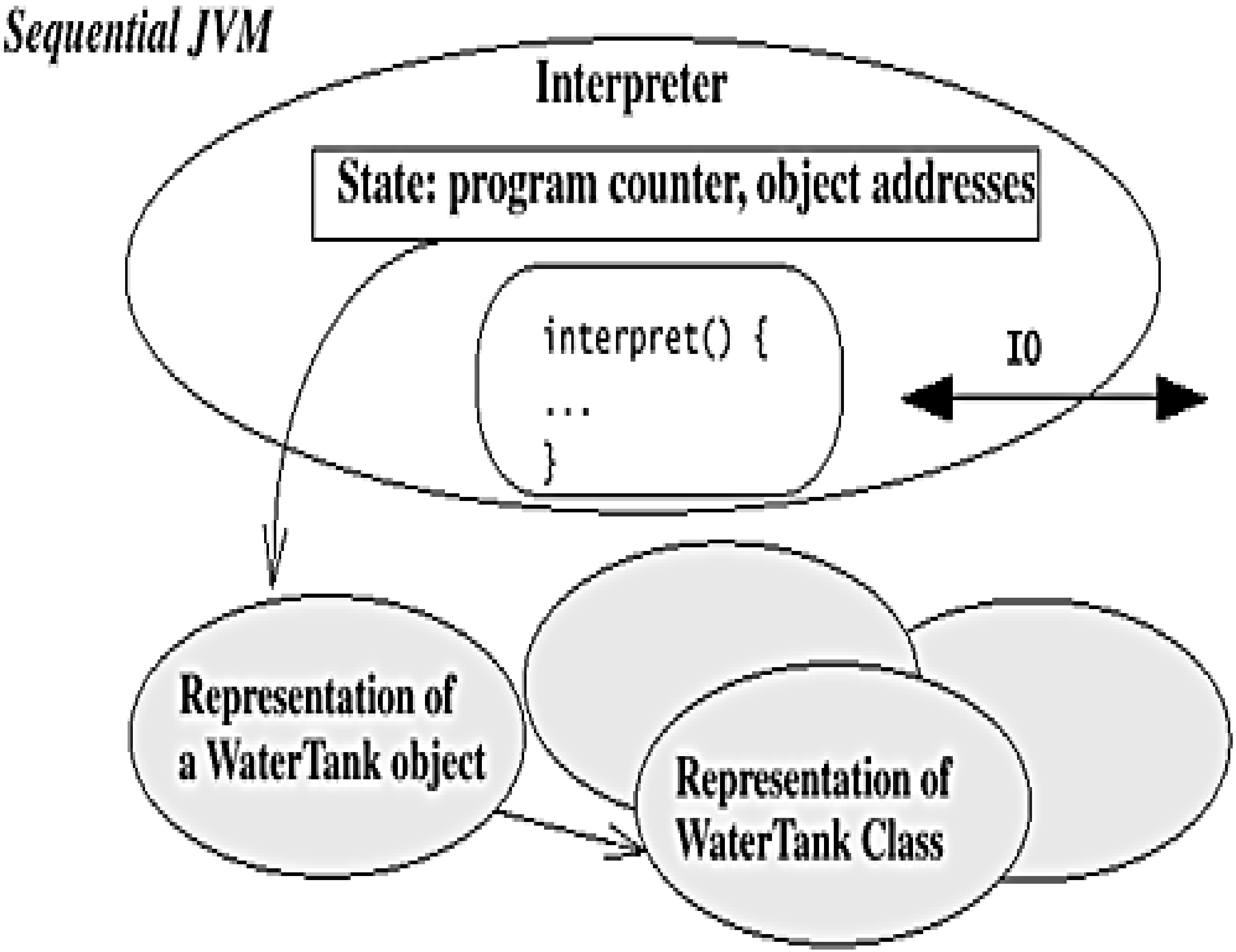
2 Click to add Title

3 Click to add Title

4 Click to add Title

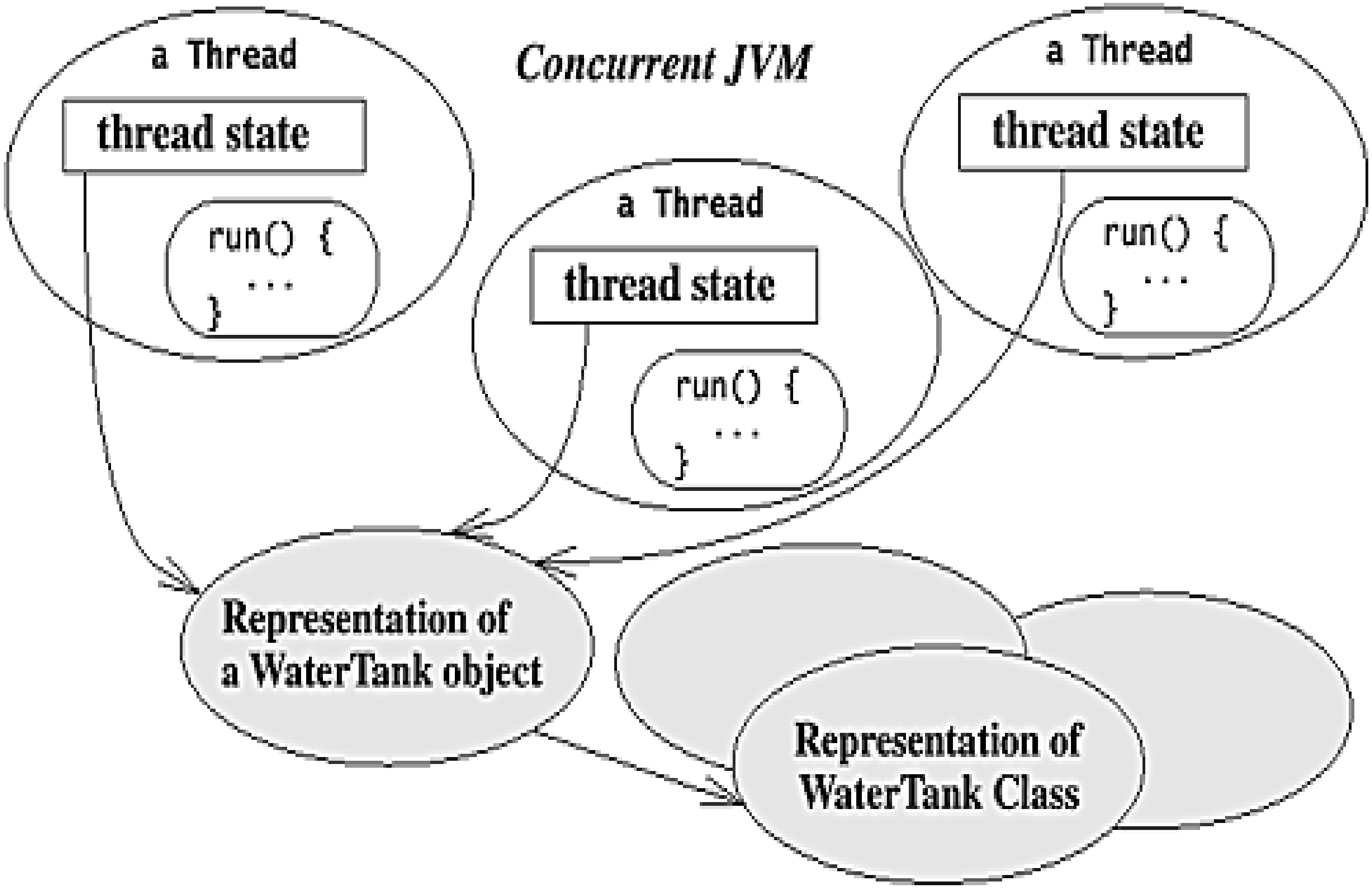
Company Logo

# Sequential JVMs


Sequential JVM
Interpreter
State: program counter, object addresses
interpret() {
...
}
IO
Representation of a WaterTank object
Representation of WaterTank Class

# Concurrent JVMs

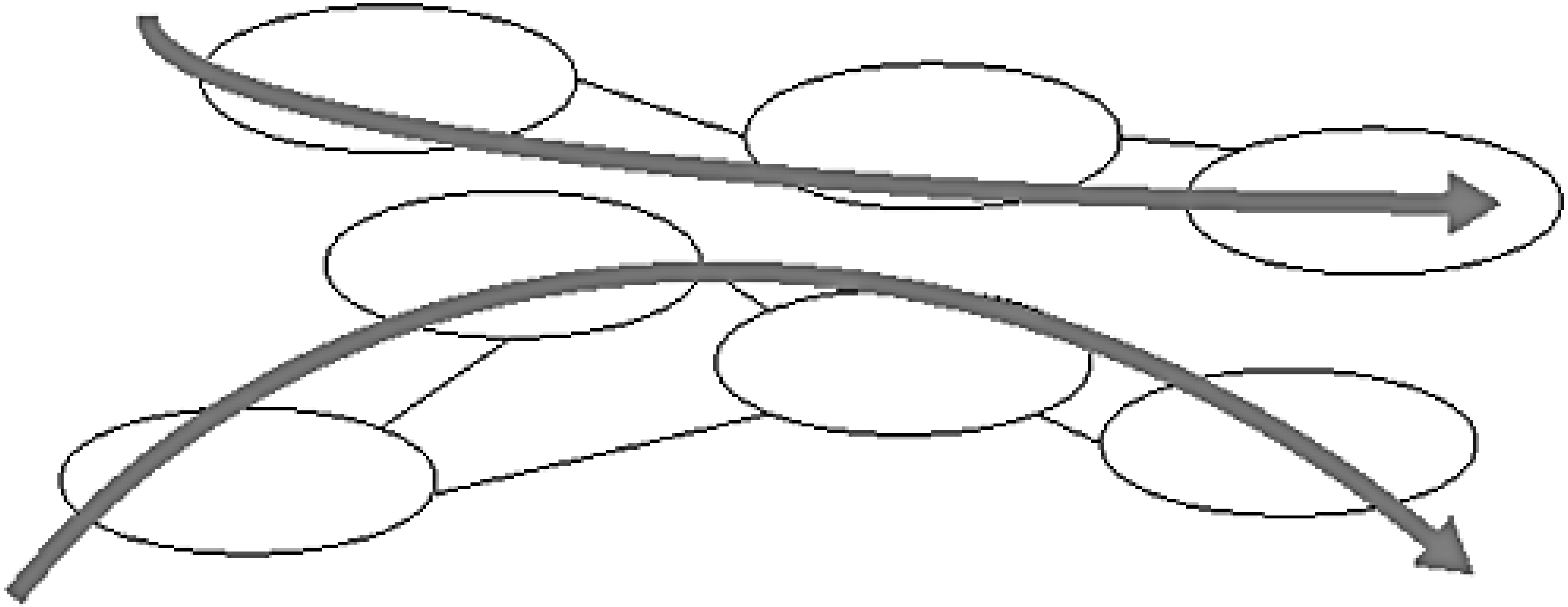
- The multiple threads may running on individual cores or Processors.



# Dealing With Concurrency

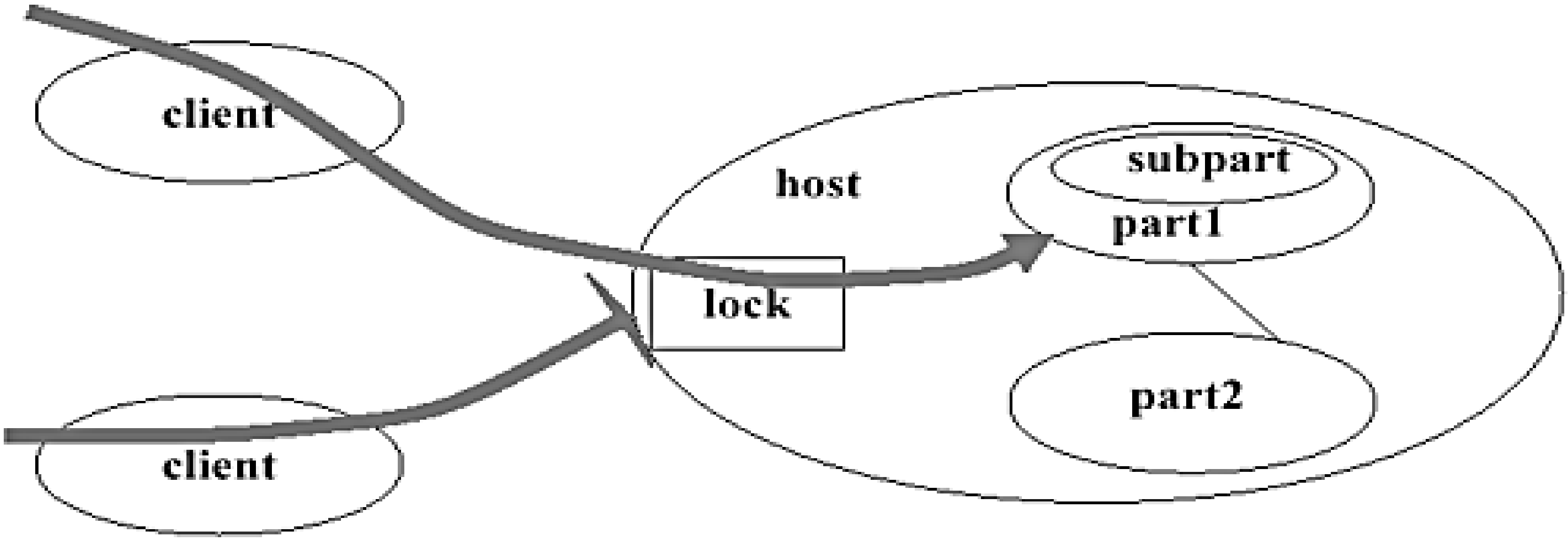
- **Dealing with Concurrency by avoiding it.**
  - Confinement
  - Immutability
  - Almost Immutability
- **Unavoidable shared state may require:**
  - Atomicity
  - Visibility
  - Restructuring and Refactoring and not erring on side of safety(erring on side of performance is not a choice.).

# Confinement



# Confinement-Object

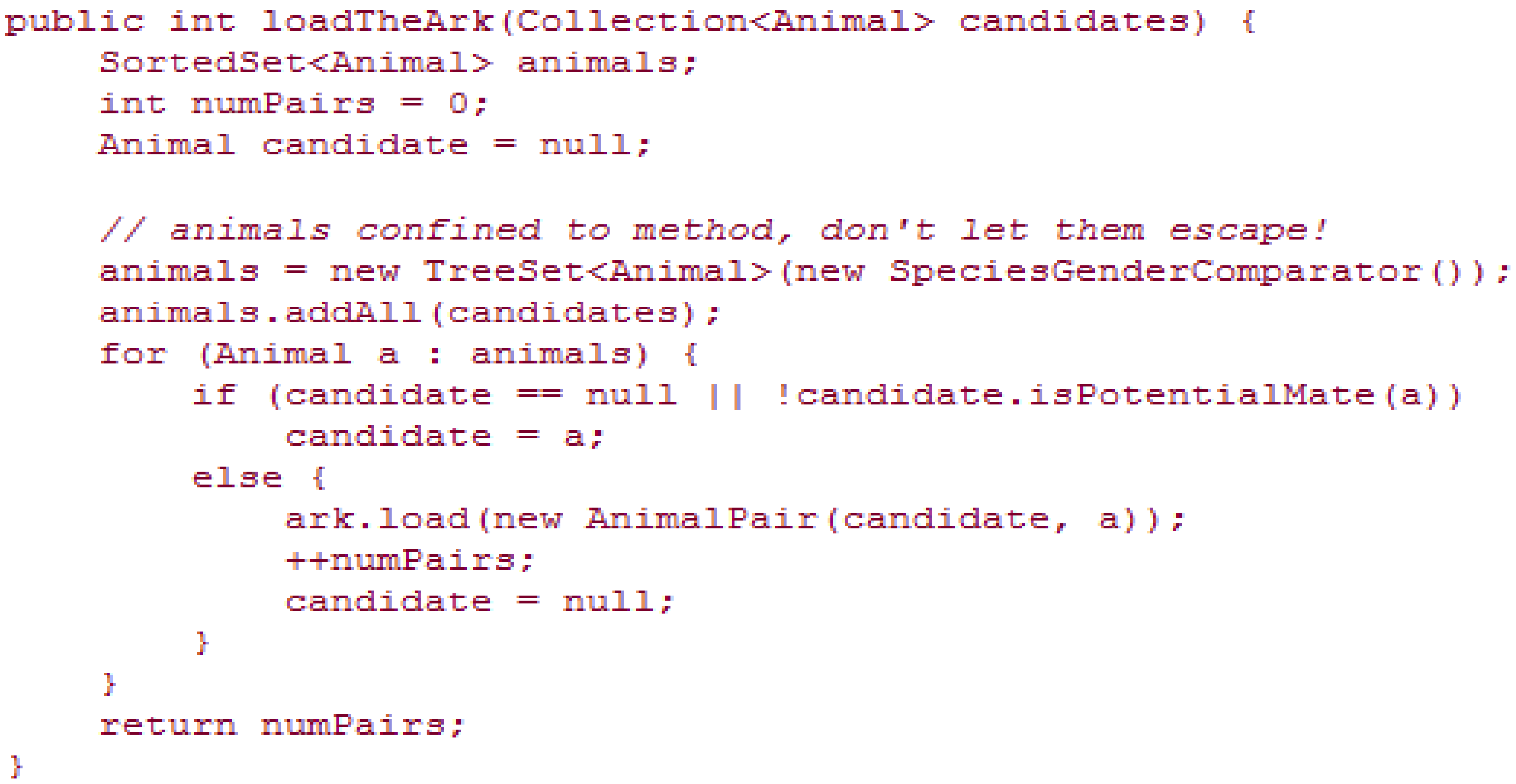
- Confinement with objects



# Confinement-Stack

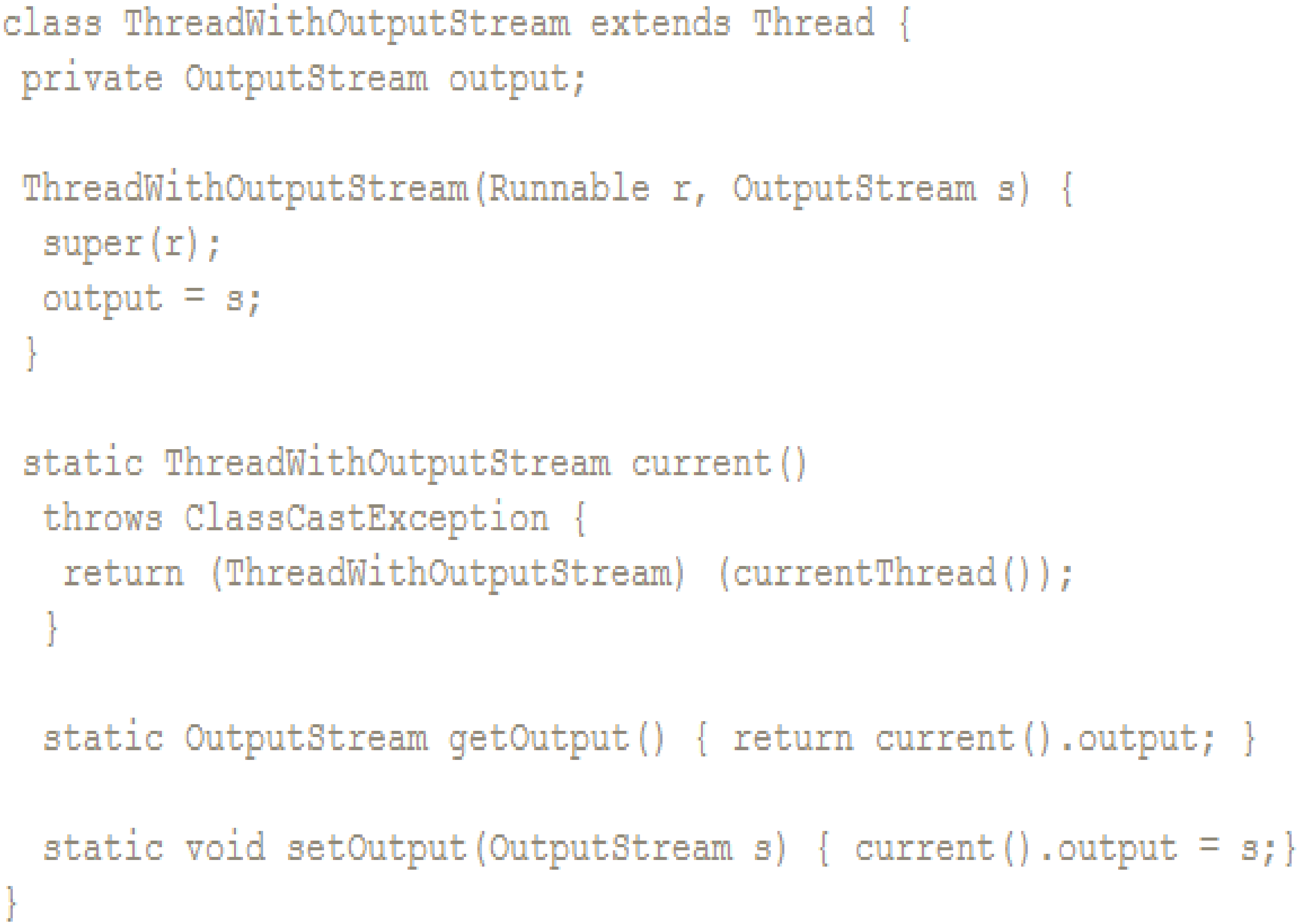
- Stack confinement is a special case of thread confinement in which an object can only be reached through local variables.
- Just as encapsulation can make it easier to preserve invariants, local variables can make it easier to confine objects to a thread.

# Confinement-Stack



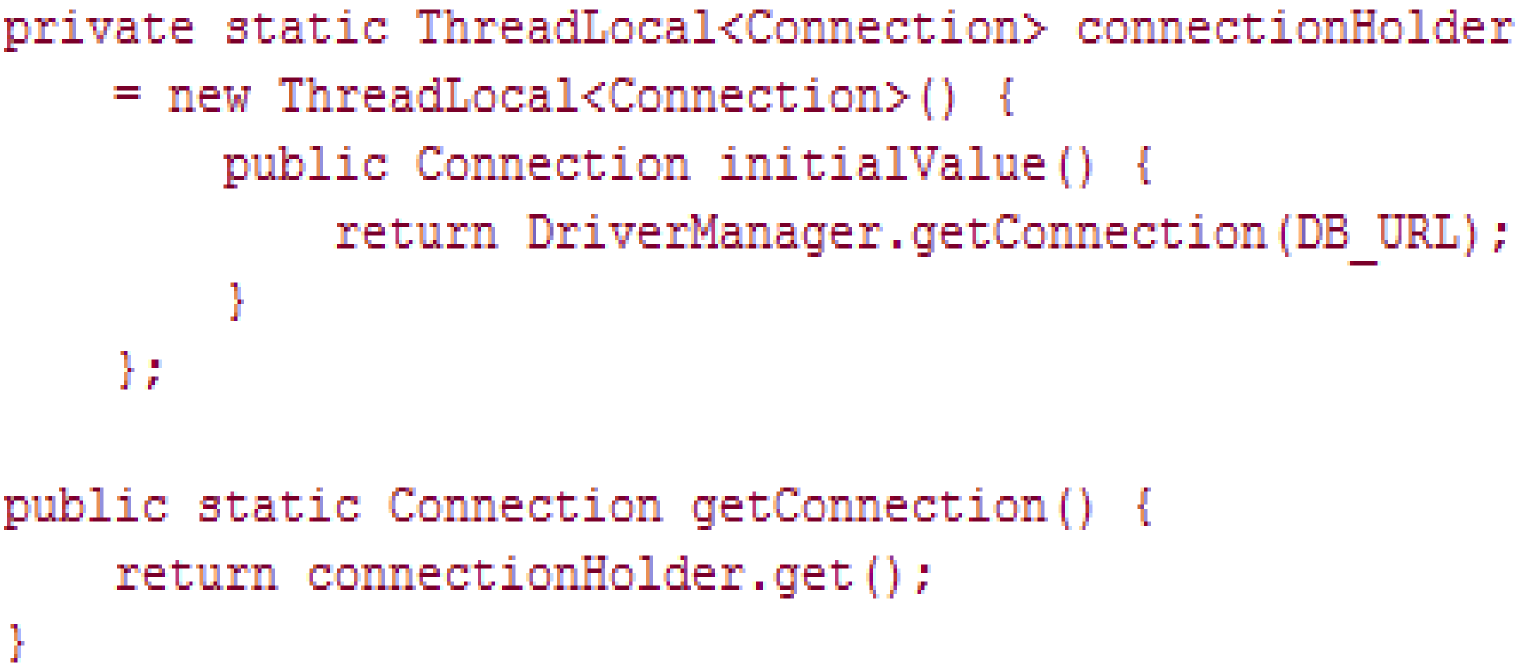
```
public int loadTheArk(Collection<Animal> candidates) {
    SortedSet<Animal> animals;
    int numPairs = 0;
    Animal candidate = null;

    // animals confined to method, don't let them escape!
    animals = new TreeSet<Animal>(new SpeciesGenderComparator());
    animals.addAll(candidates);
    for (Animal a : animals) {
        if (candidate == null || !candidate.isPotentialMate(a))
            candidate = a;
        else {
            ark.load(new AnimalPair(candidate, a));
            ++numPairs;
            candidate = null;
        }
    }
    return numPairs;
}
```

# Confinement-ThreadLocal



```
class ThreadWithOutputStream extends Thread {
 private OutputStream output;

 ThreadWithOutputStream(Runnable r, OutputStream s) {
  super(r);
  output = s;
 }

 static ThreadWithOutputStream current()
  throws ClassCastException {
   return (ThreadWithOutputStream) (currentThread());
  }

  static OutputStream getOutput() { return current().output; }

  static void setOutput(OutputStream s) { current().output = s;}
}
```

# Confinement-ThreadLocal

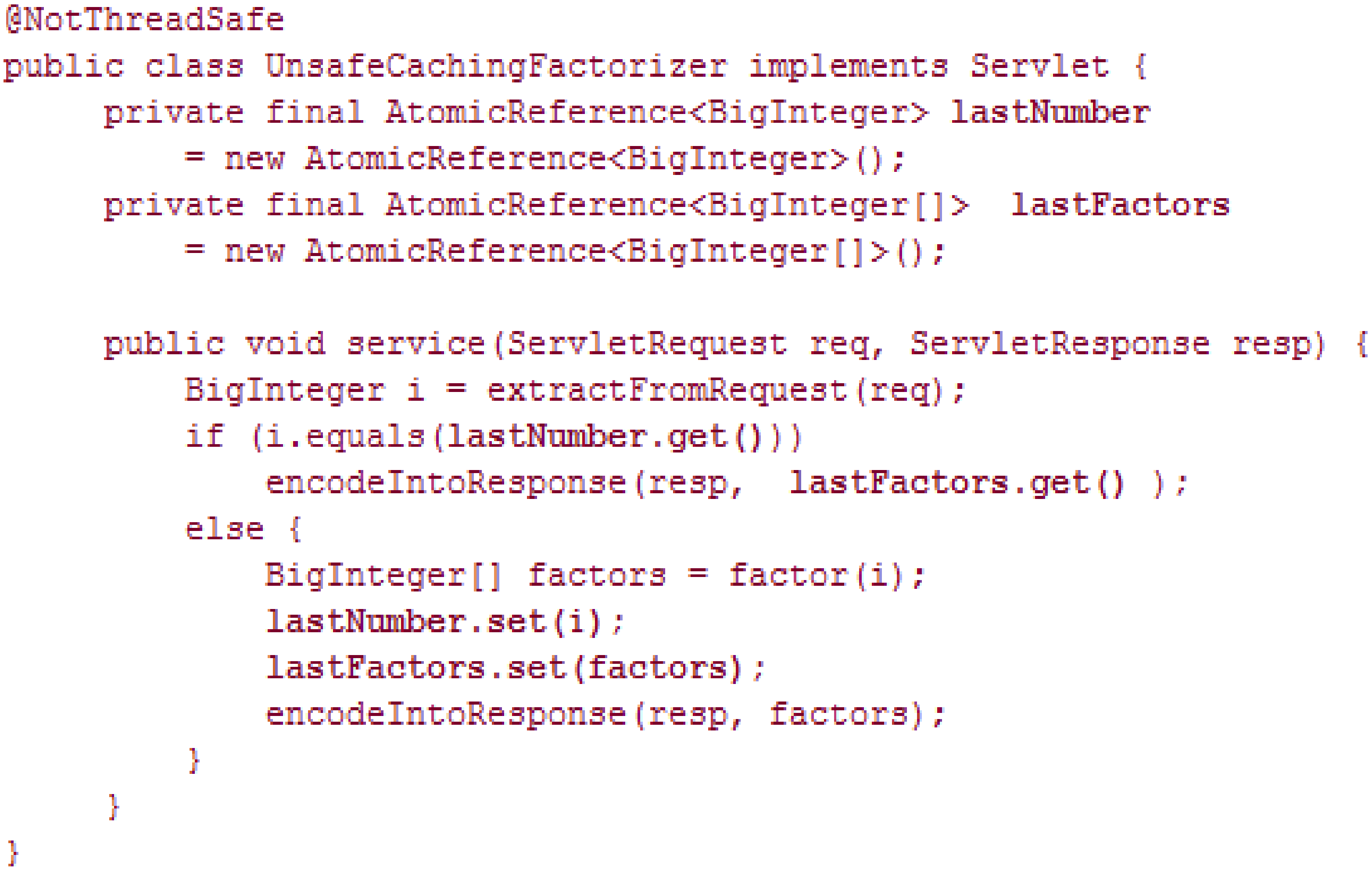


```
private static ThreadLocal<Connection> connectionHolder
    = new ThreadLocal<Connection>() {
        public Connection initialValue() {
            return DriverManager.getConnection(DB_URL);
        }
    };

public static Connection getConnection() {
    return connectionHolder.get();
}
```

# Immutability

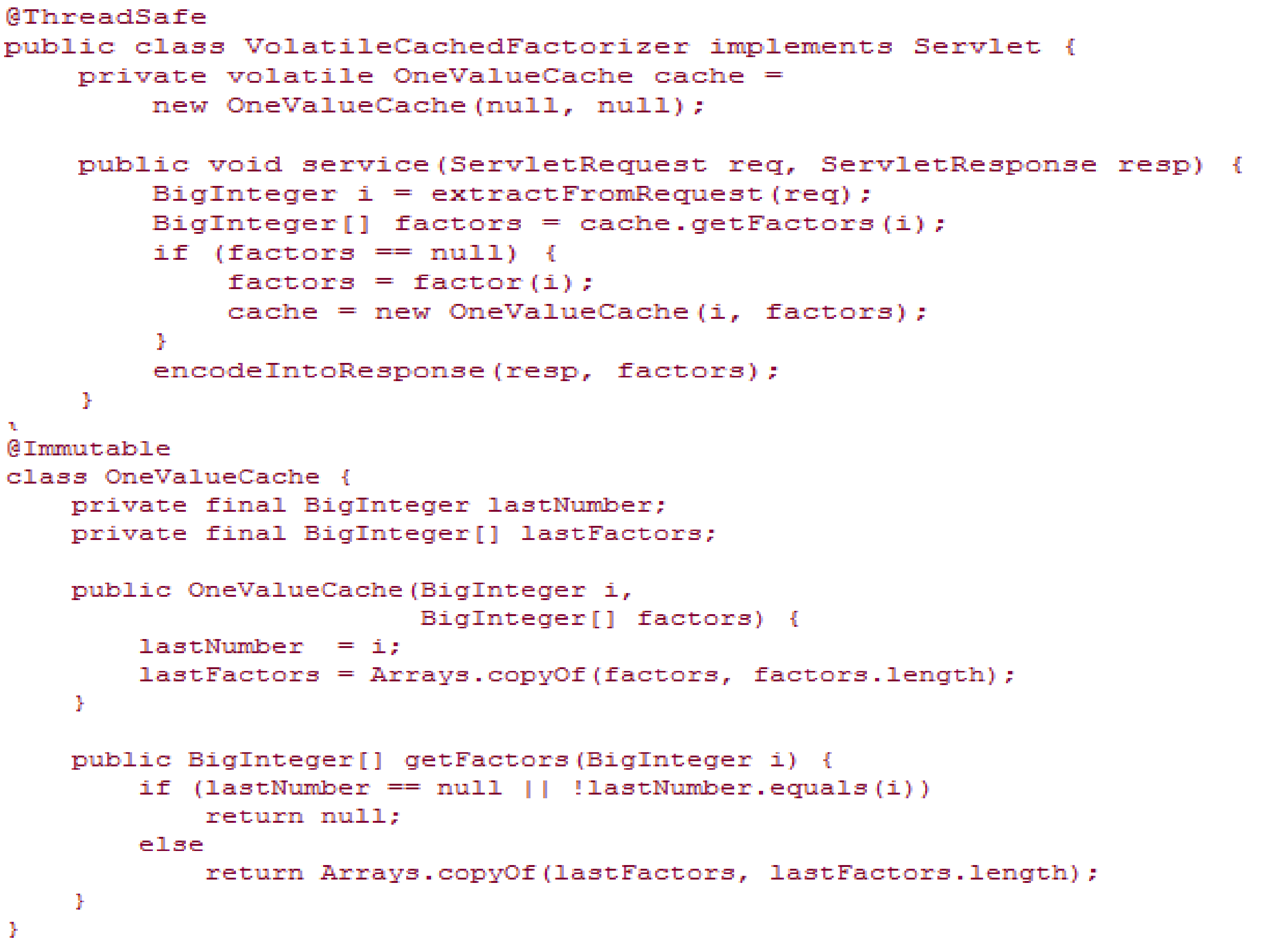
- Private , final field.
- Defensive Copy of mutable constructor parameters
- Defensive Copy mutable objects in assessors.
- No Setters
- Class effectively final.

# Immutability-Example1



```
@NotThreadSafe
public class UnsafeCachingFactorizer implements Servlet {
     private final AtomicReference<BigInteger> lastNumber
         = new AtomicReference<BigInteger>();
     private final AtomicReference<BigInteger[]>  lastFactors
         = new AtomicReference<BigInteger[]>();

     public void service(ServletRequest req, ServletResponse resp) {
         BigInteger i = extractFromRequest(req);
         if (i.equals(lastNumber.get()))
             encodeIntoResponse(resp,  lastFactors.get() );
         else {
             BigInteger[] factors = factor(i);
             lastNumber.set(i);
             lastFactors.set(factors);
             encodeIntoResponse(resp, factors);
         }
     }
}
```

# Immutability-Example1



```
@ThreadSafe
public class VolatileCachedFactorizer implements Servlet {
    private volatile OneValueCache cache =
        new OneValueCache(null, null);

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = cache.getFactors(i);
        if (factors == null) {
            factors = factor(i);
            cache = new OneValueCache(i, factors);
        }
        encodeIntoResponse(resp, factors);
    }
}
@Immutable
class OneValueCache {
    private final BigInteger lastNumber;
    private final BigInteger[] lastFactors;

    public OneValueCache(BigInteger i,
                         BigInteger[] factors) {
        lastNumber  = i;
        lastFactors = Arrays.copyOf(factors, factors.length);
    }

    public BigInteger[] getFactors(BigInteger i) {
        if (lastNumber == null || !lastNumber.equals(i))
            return null;
        else
            return Arrays.copyOf(lastFactors, lastFactors.length);
    }
}
```

# Almost Immutability

- **These are normal mutable object that are not thread safe, but their usage patterns allow us to relax the level of locking.**
  - These object are initialized by a single thread and accessed by other threads.
  - The accessing threads make no changes to the object.
  - These objects must be safely published.

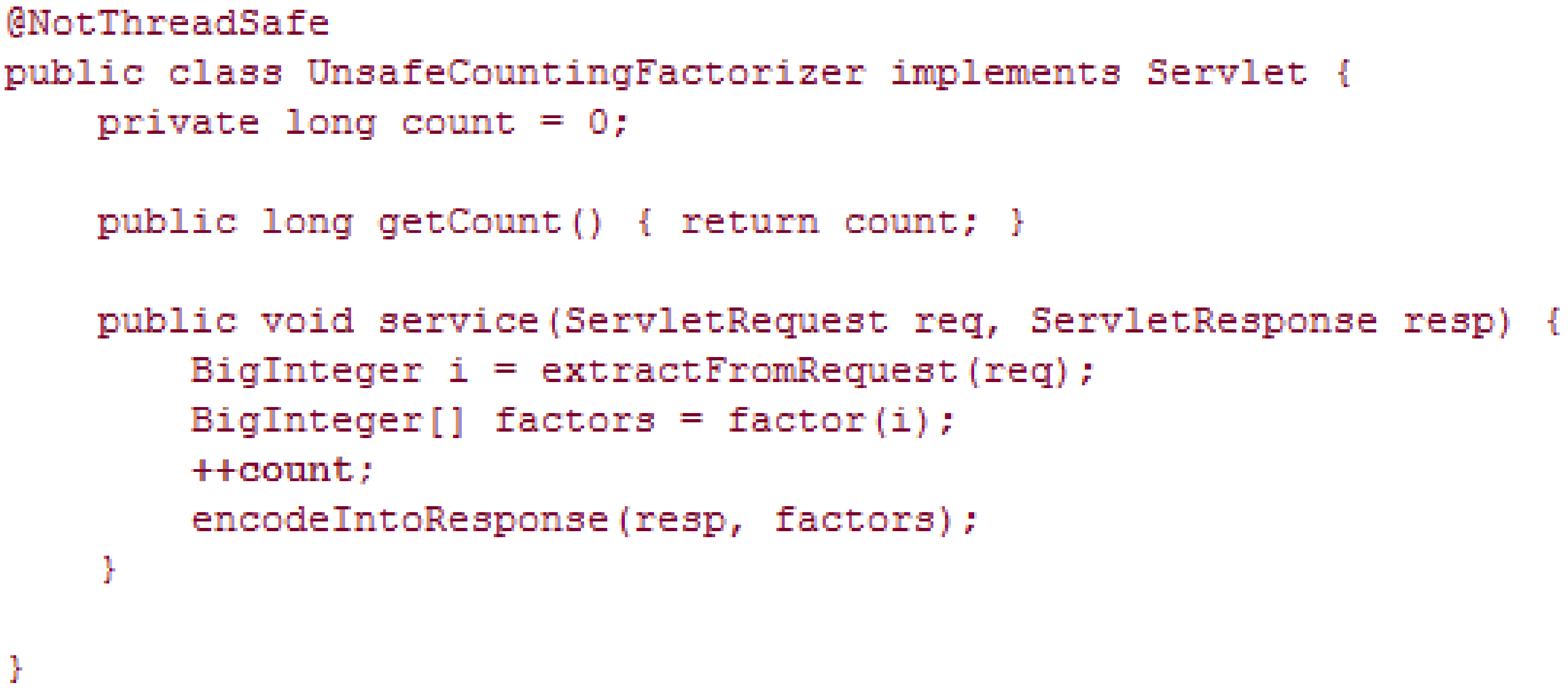
# Almost Immutability-Safe Publication

- **To publish an object safely, both the reference to the object and the object's state must be made visible to other threads at the same time.**
  - Initializing an object reference from a static initializer;
  - Storing a reference to it into a volatile field or AtomicReference;
  - Storing a reference to it into a final field of a properly constructed object; or
  - Storing a reference to it into a field that is properly guarded by a lock.

# Almost Immutability

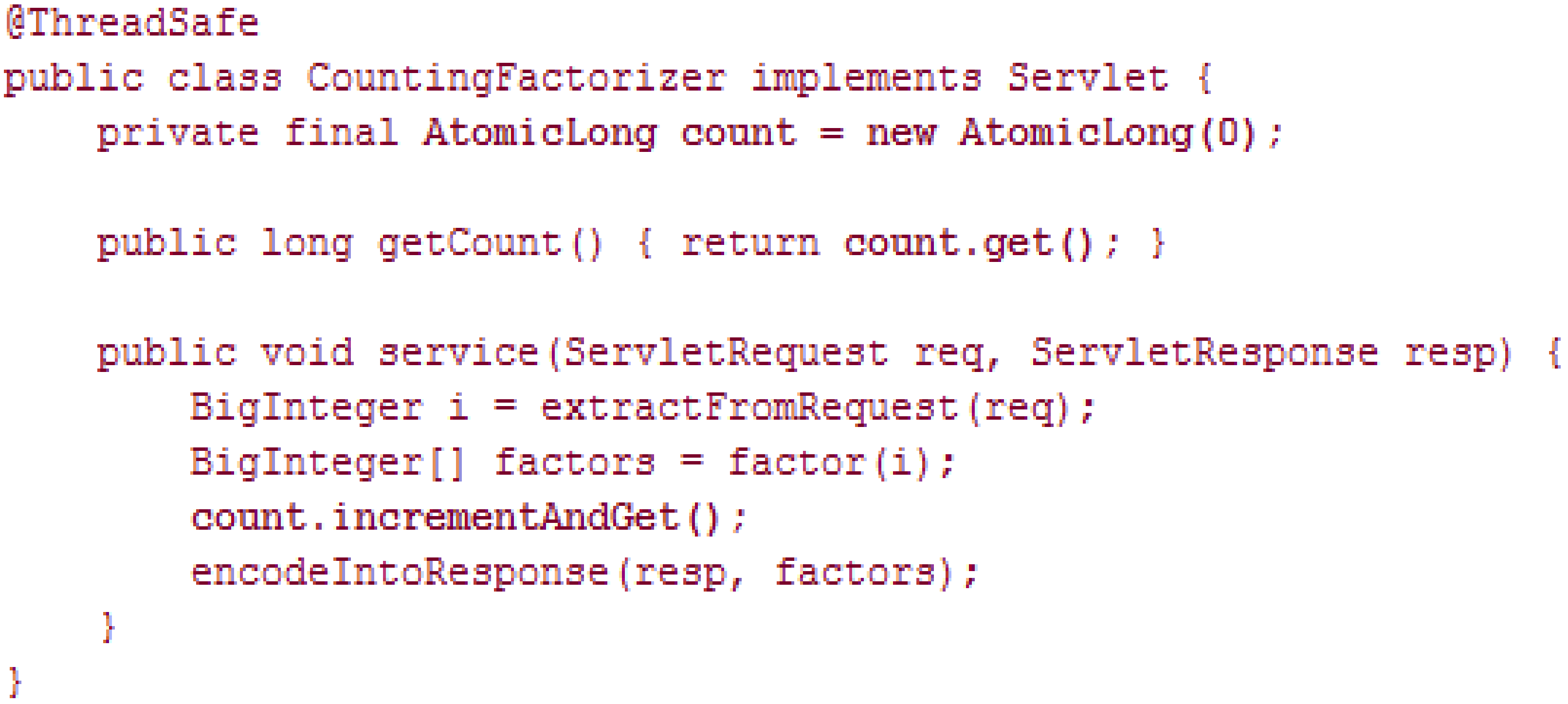
- **The publication requirements for an object depend on its mutability:**
  - Immutable objects can be published through any mechanism;
  - Effectively immutable objects must be safely published;
  - Mutable objects must be safely published, and must be either threadsafe or guarded by a lock.

# Atomicity



```
@NotThreadSafe
public class UnsafeCountingFactorizer implements Servlet {
    private long count = 0;

    public long getCount() { return count; }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        ++count;
        encodeIntoResponse(resp, factors);
    }

}
```

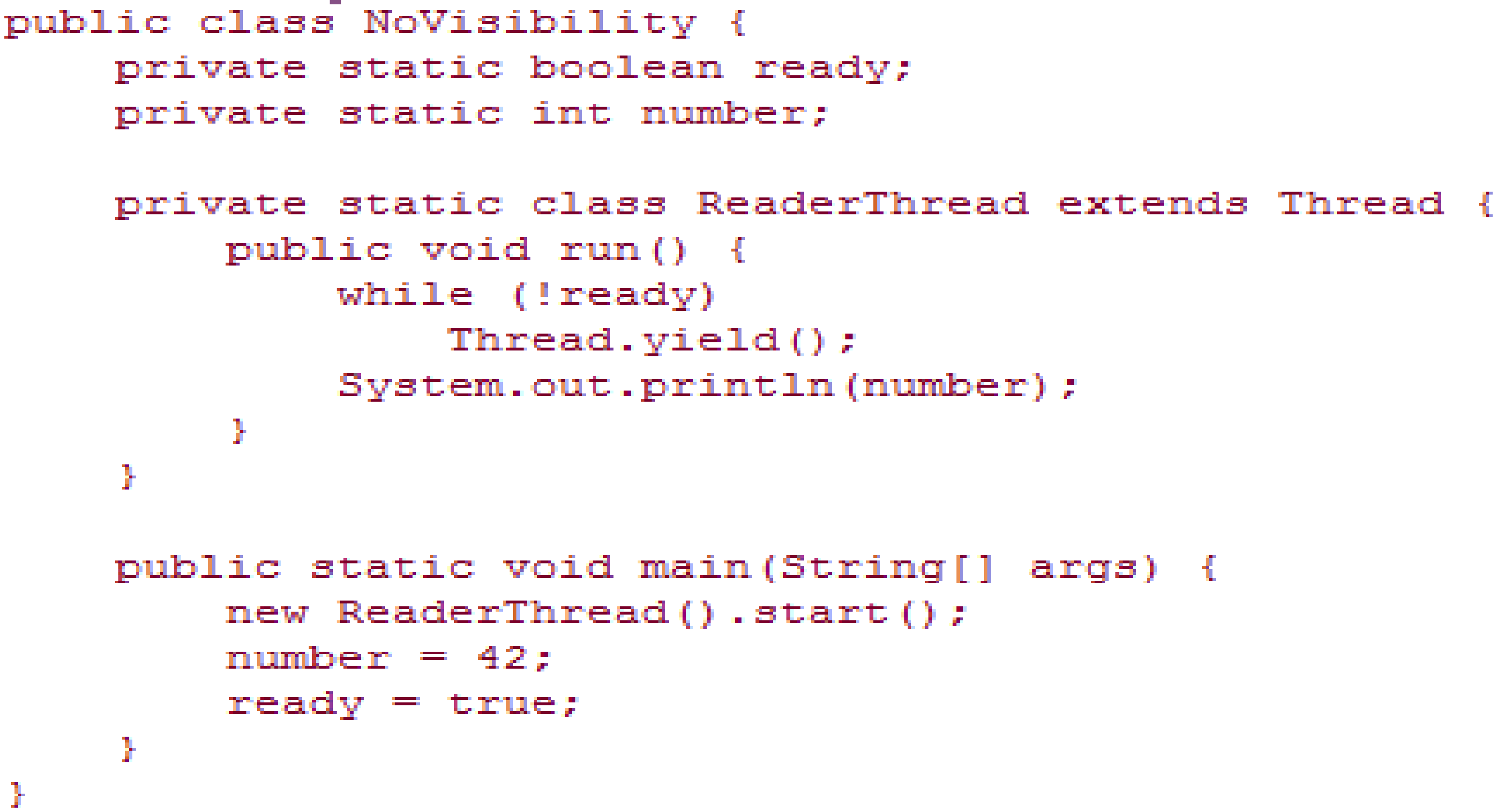
# Atomicity



```
@ThreadSafe
public class CountingFactorizer implements Servlet {
    private final AtomicLong count = new AtomicLong(0);

    public long getCount() { return count.get(); }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count.incrementAndGet();
        encodeIntoResponse(resp, factors);
    }
}
```

# Visibility

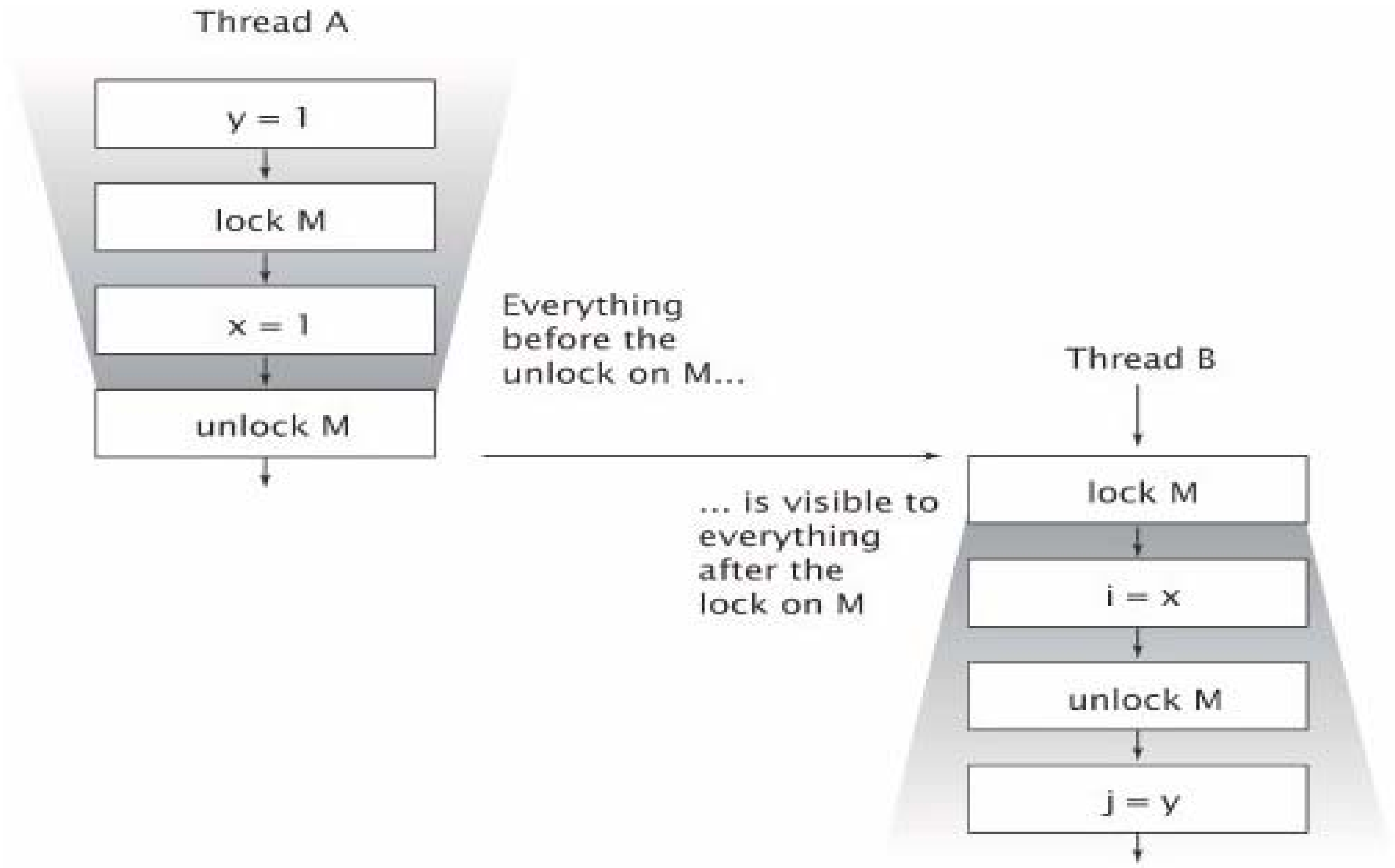
- NoVisibility could loop forever and it could print zero



```java
public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready)
                Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

# Visibility

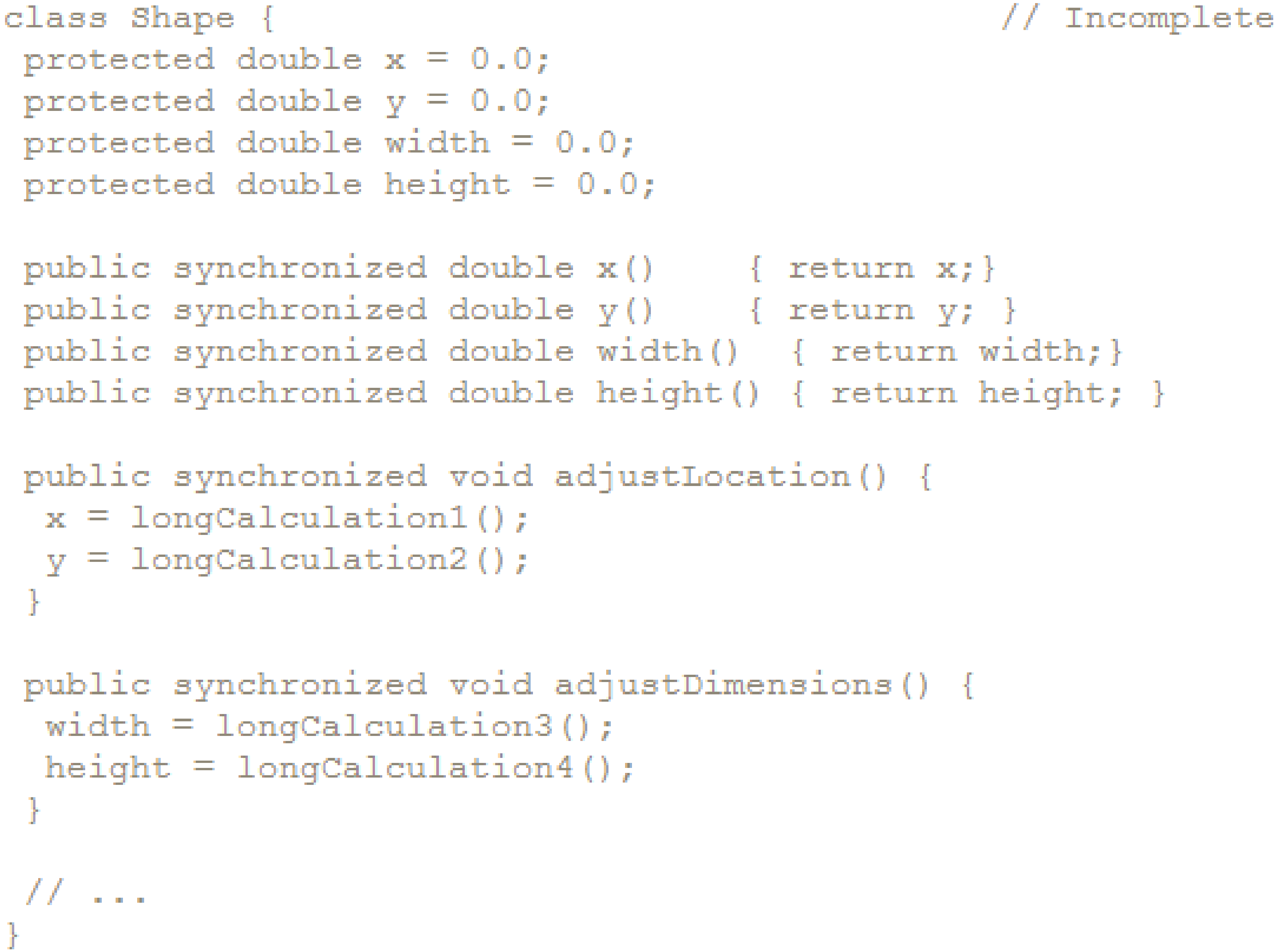
- Visibility guarantees of synchronization,volatile.


Thread A
y = 1
lock M
x = 1
unlock M
Everything before the unlock on M...
Thread B
lock M
... is visible to everything after the lock on M
i = x
unlock M
j = y

# Visibility

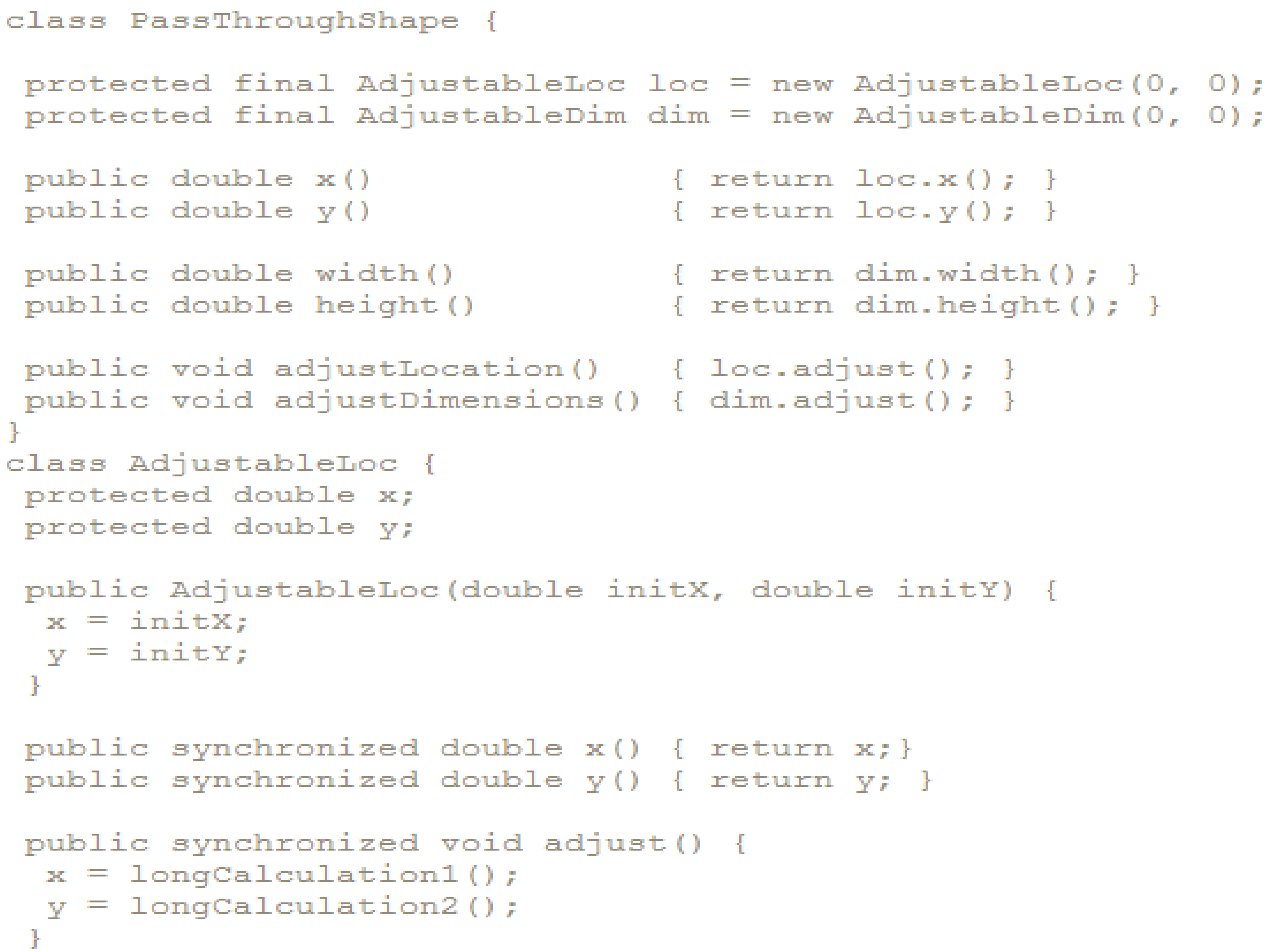
- Locking can guarantee both visibility and atomicity; volatile variables can only guarantee visibility.
- The semantics of volatile are not strong enough to make the increment operation (count++) atomic, unless you can guarantee that the variable is written only from a single thread

# Restructuring and Refactoring- Lock Splitting



```
class Shape {                                     // Incomplete
 protected double x = 0.0;
 protected double y = 0.0;
 protected double width = 0.0;
 protected double height = 0.0;

 public synchronized double x()     { return x;}
 public synchronized double y()     { return y; }
 public synchronized double width()  { return width;}
 public synchronized double height() { return height; }

 public synchronized void adjustLocation() {
  x = longCalculation1();
  y = longCalculation2();
 }

 public synchronized void adjustDimensions() {
  width = longCalculation3();
  height = longCalculation4();
 }

 // ...
}
```

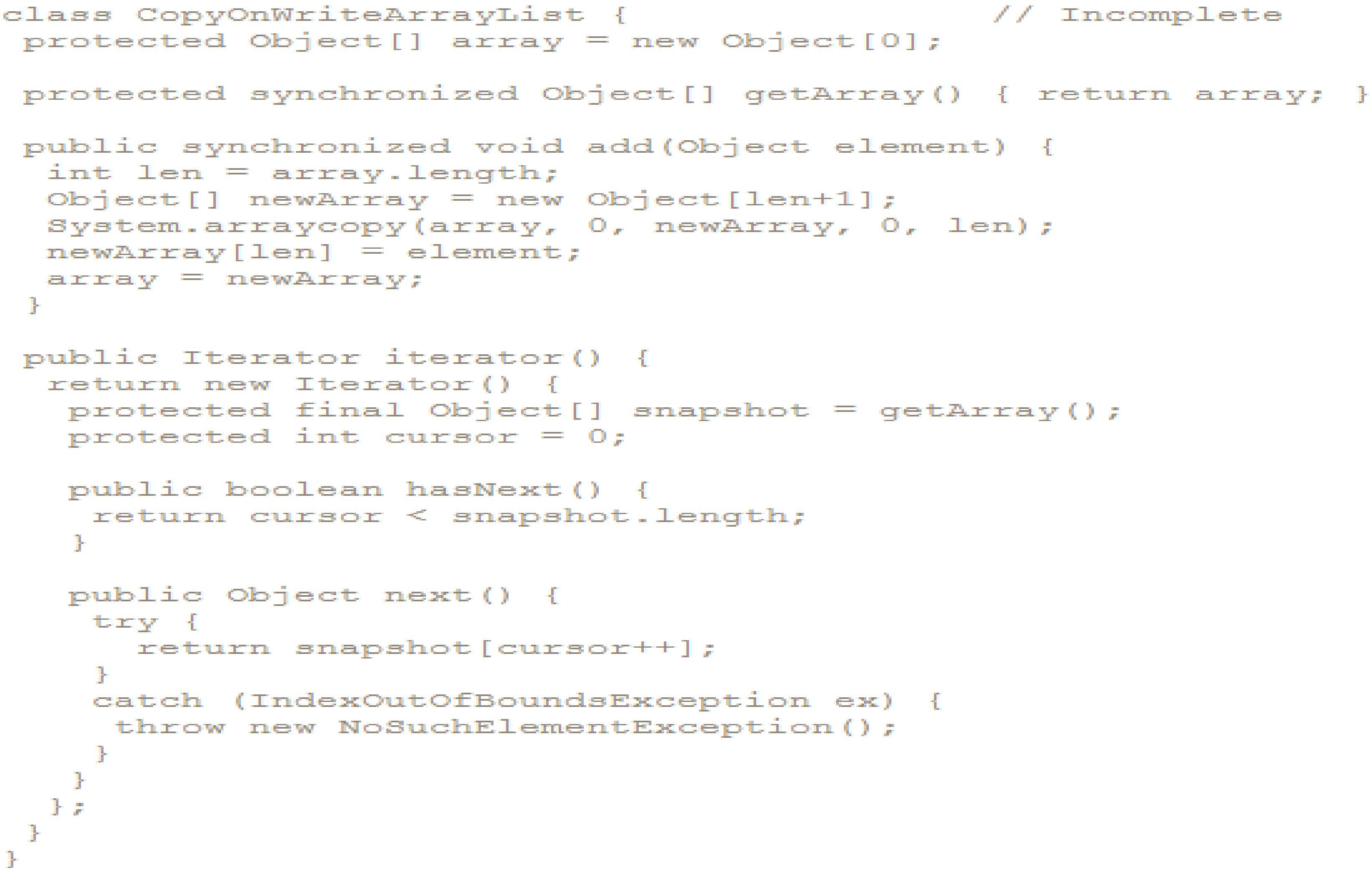
# Restructuring and Refactoring-Lock Splitting



```
class PassThroughShape {

 protected final AdjustableLoc loc = new AdjustableLoc(0, 0);
 protected final AdjustableDim dim = new AdjustableDim(0, 0);

 public double x()                  { return loc.x(); }
 public double y()                  { return loc.y(); }

 public double width()              { return dim.width(); }
 public double height()             { return dim.height(); }

 public void adjustLocation()       { loc.adjust(); }
 public void adjustDimensions()     { dim.adjust(); }
}
class AdjustableLoc {
 protected double x;
 protected double y;

 public AdjustableLoc(double initX, double initY) {
  x = initX;
  y = initY;
 }

 public synchronized double x() { return x;}
 public synchronized double y() { return y; }

 public synchronized void adjust() {
  x = longCalculation1();
  y = longCalculation2();
 }
```

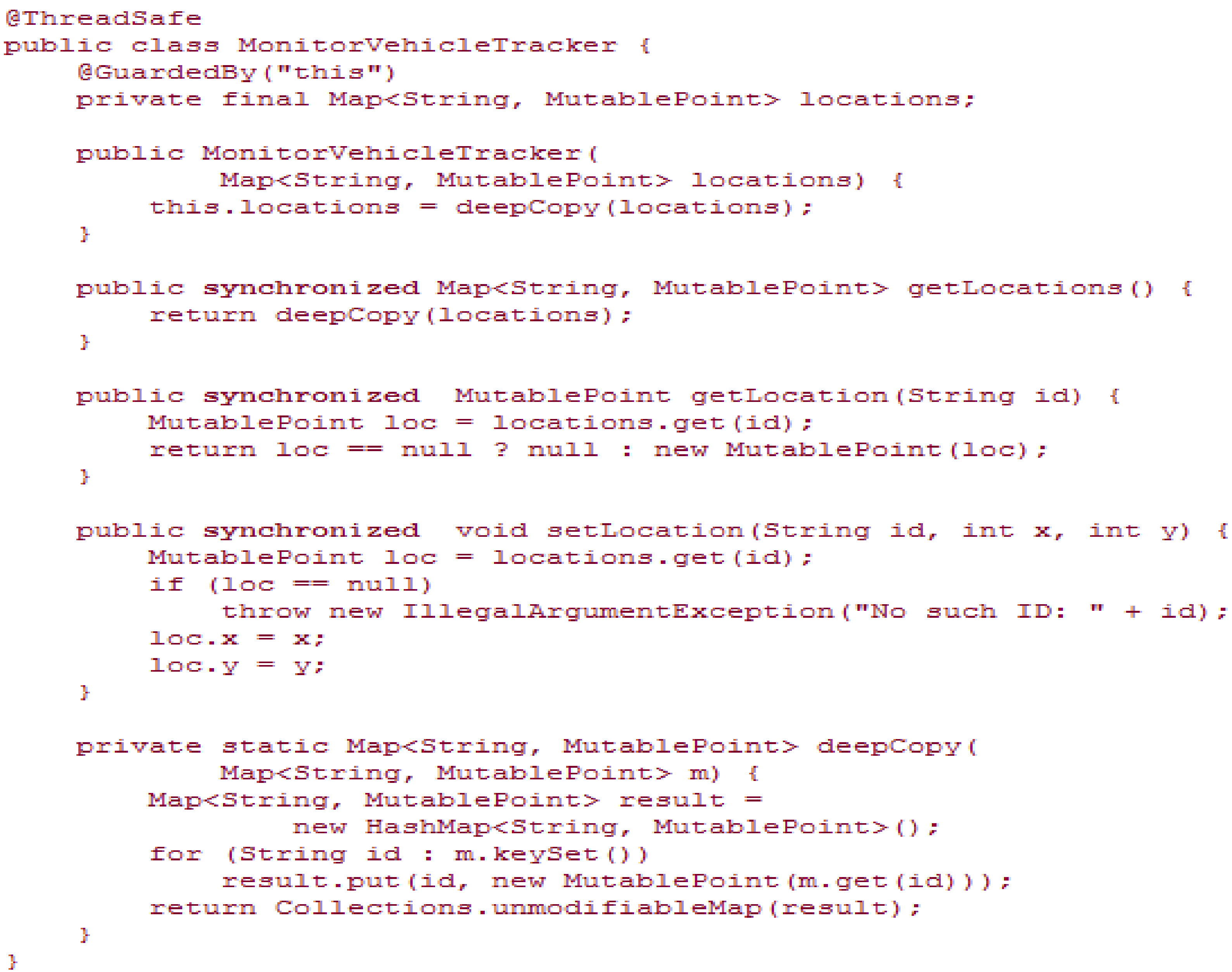
# Restructuring and Refactoring- COW

- **In new memory model volatile can replace sync.**



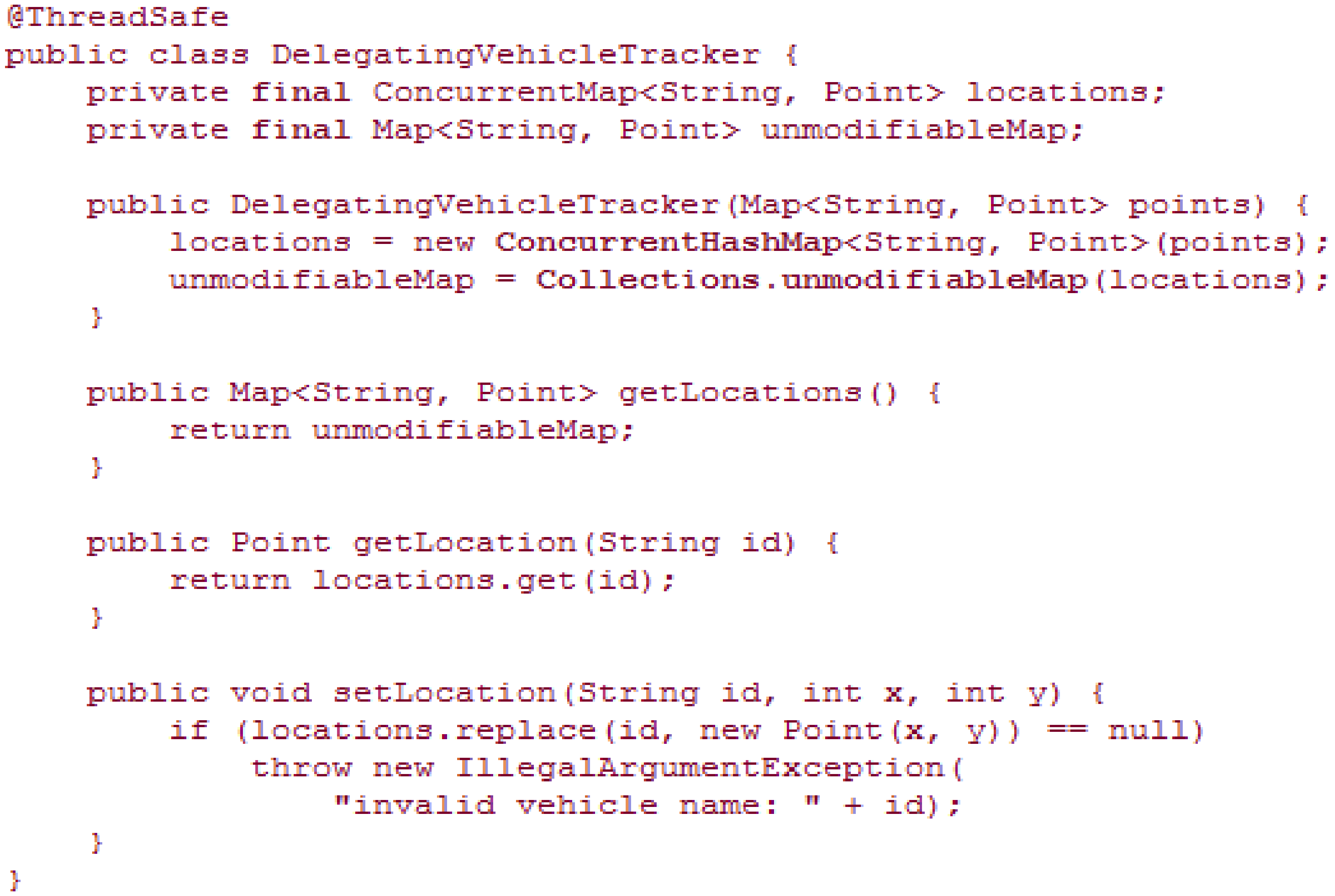
```
class CopyOnWriteArrayList {                    // Incomplete
 protected Object[] array = new Object[0];

 protected synchronized Object[] getArray() { return array; }

 public synchronized void add(Object element) {
  int len = array.length;
  Object[] newArray = new Object[len+1];
  System.arraycopy(array, 0, newArray, 0, len);
  newArray[len] = element;
  array = newArray;
 }

 public Iterator iterator() {
  return new Iterator() {
   protected final Object[] snapshot = getArray();
   protected int cursor = 0;

   public boolean hasNext() {
    return cursor < snapshot.length;
   }

   public Object next() {
    try {
      return snapshot[cursor++];
    }
    catch (IndexOutOfBoundsException ex) {
     throw new NoSuchElementException();
    }
   }
  };
 }
}
```

# Restructuring and Refactoring-Delegating



```
@ThreadSafe
public class MonitorVehicleTracker {
    @GuardedBy("this")
    private final Map<String, MutablePoint> locations;

    public MonitorVehicleTracker(
            Map<String, MutablePoint> locations) {
        this.locations = deepCopy(locations);
    }

    public synchronized Map<String, MutablePoint> getLocations() {
        return deepCopy(locations);
    }

    public synchronized  MutablePoint getLocation(String id) {
        MutablePoint loc = locations.get(id);
        return loc == null ? null : new MutablePoint(loc);
    }

    public synchronized  void setLocation(String id, int x, int y) {
        MutablePoint loc = locations.get(id);
        if (loc == null)
            throw new IllegalArgumentException("No such ID: " + id);
        loc.x = x;
        loc.y = y;
    }

    private static Map<String, MutablePoint> deepCopy(
            Map<String, MutablePoint> m) {
        Map<String, MutablePoint> result =
                new HashMap<String, MutablePoint>();
        for (String id : m.keySet())
            result.put(id, new MutablePoint(m.get(id)));
        return Collections.unmodifiableMap(result);
    }
}
```

# Restructuring and Refactoring-Delegating



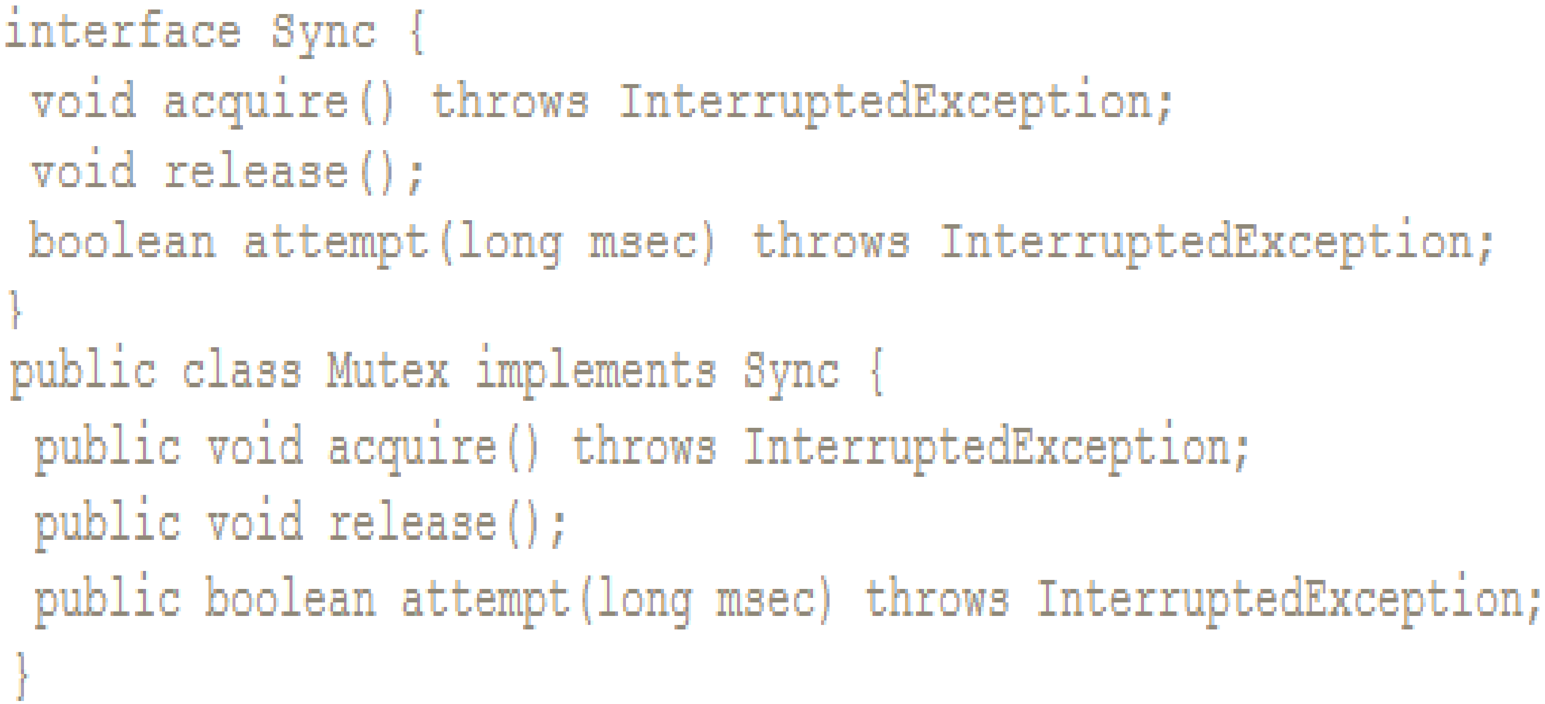
```
@ThreadSafe
public class DelegatingVehicleTracker {
    private final ConcurrentMap<String, Point> locations;
    private final Map<String, Point> unmodifiableMap;

    public DelegatingVehicleTracker(Map<String, Point> points) {
        locations = new ConcurrentHashMap<String, Point>(points);
        unmodifiableMap = Collections.unmodifiableMap(locations);
    }

    public Map<String, Point> getLocations() {
        return unmodifiableMap;
    }

    public Point getLocation(String id) {
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y) {
        if (locations.replace(id, new Point(x, y)) == null)
            throw new IllegalArgumentException(
                "invalid vehicle name: " + id);
    }
}
```

# Restructuring and Refactoring-Wrappers

- Returning synchronized or read-only Wrappers.

# Synchronizer Framework

- many concurrency control constructs conform to an acquire-release protocol that can be encompassed under the simple interface:



```
interface Sync {
 void acquire() throws InterruptedException;
 void release();
 boolean attempt(long msec) throws InterruptedException;
}
public class Mutex implements Sync {
 public void acquire() throws InterruptedException;
 public void release();
 public boolean attempt(long msec) throws InterruptedException;
 }
```

# Lock

- Use of monitor synchronization is just fine for most applications, but it has some shortcomings
    - Single wait-set per lock
    - No way to interrupt or time-out when waiting for a lock
    - Locking must be block-structured
        - Inconvenient to acquire a variable number of locks at once
        - Advanced techniques, such as hand-over-hand locking, are not possible
- Lock objects address these limitations
    - But harder to use: Need `finally` block to ensure release
    - So if you don’t need them, stick with `synchronized`

# Locks

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

- Additional flexibility
  - Interruptible, try-lock, not block-structured, multiple conditions
  - Advanced uses: e.g. Hand-over-hand or chained locking
- ReentrantLock: mutual-exclusion Lock implementation
  - Same basic semantics as synchronized

    Reentrant, must hold lock before using condition, …
  - Supports fair and non-fair behavior

    Fair lock granted to waiting threads ahead of new requests

# Locks

- Used extensively within `java.util.concurrent`

```
final Lock lock = new ReentrantLock();
 ...
 lock.lock();
 try {
   // perform operations protected by lock
 }
 catch(Exception ex) {
   // restore invariants & rethrow
 }
 finally {
   lock.unlock();
 }
```

- **Must manually ensure lock is released**

# Canned Synchronizers

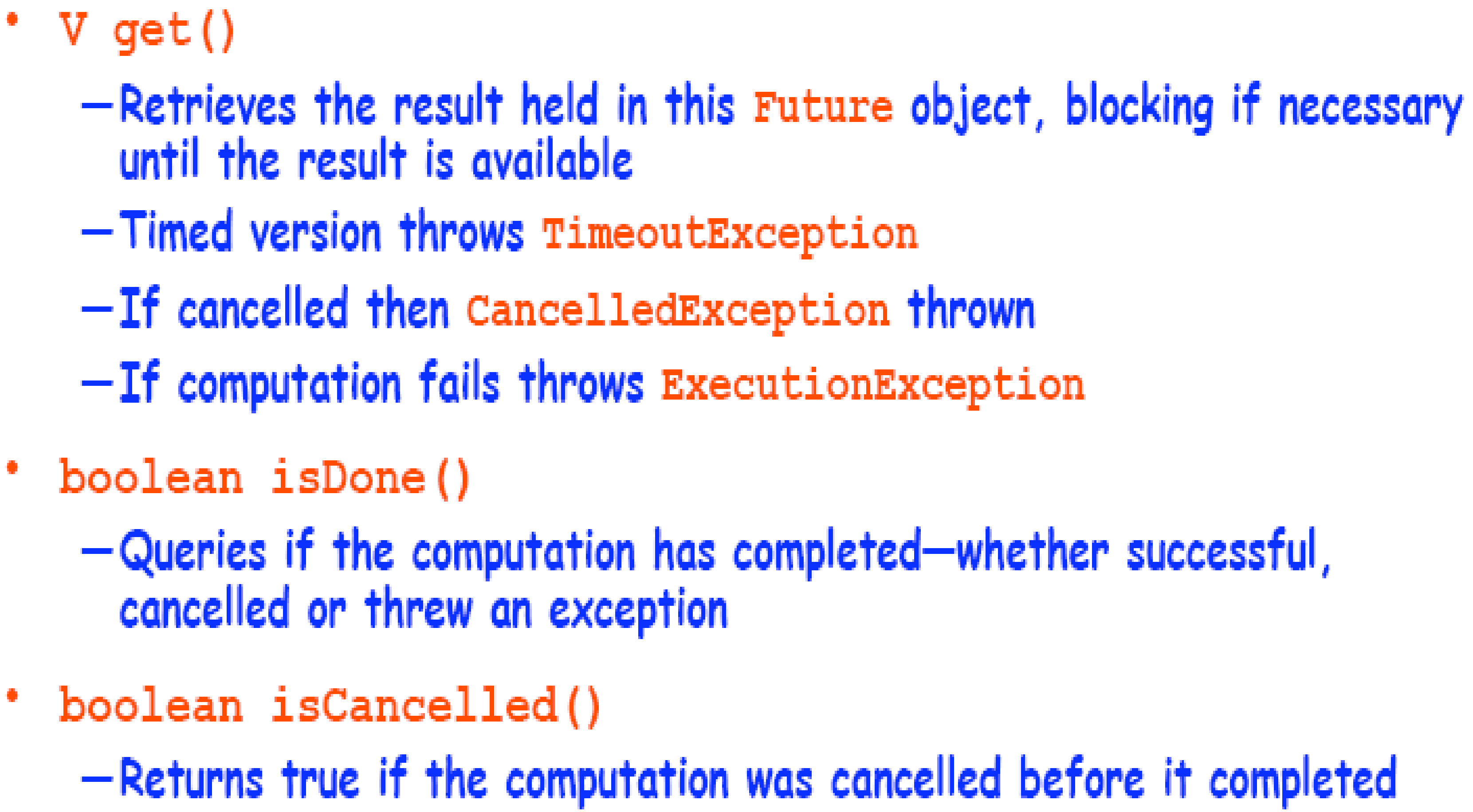
## Thread Safe Canned Synchronizers

- Future
- Semaphore
- Mutex
- Barrier
- CountdownLatches
- SynchronousQueue
- Exchanger

# Canned Synchronizers-Future

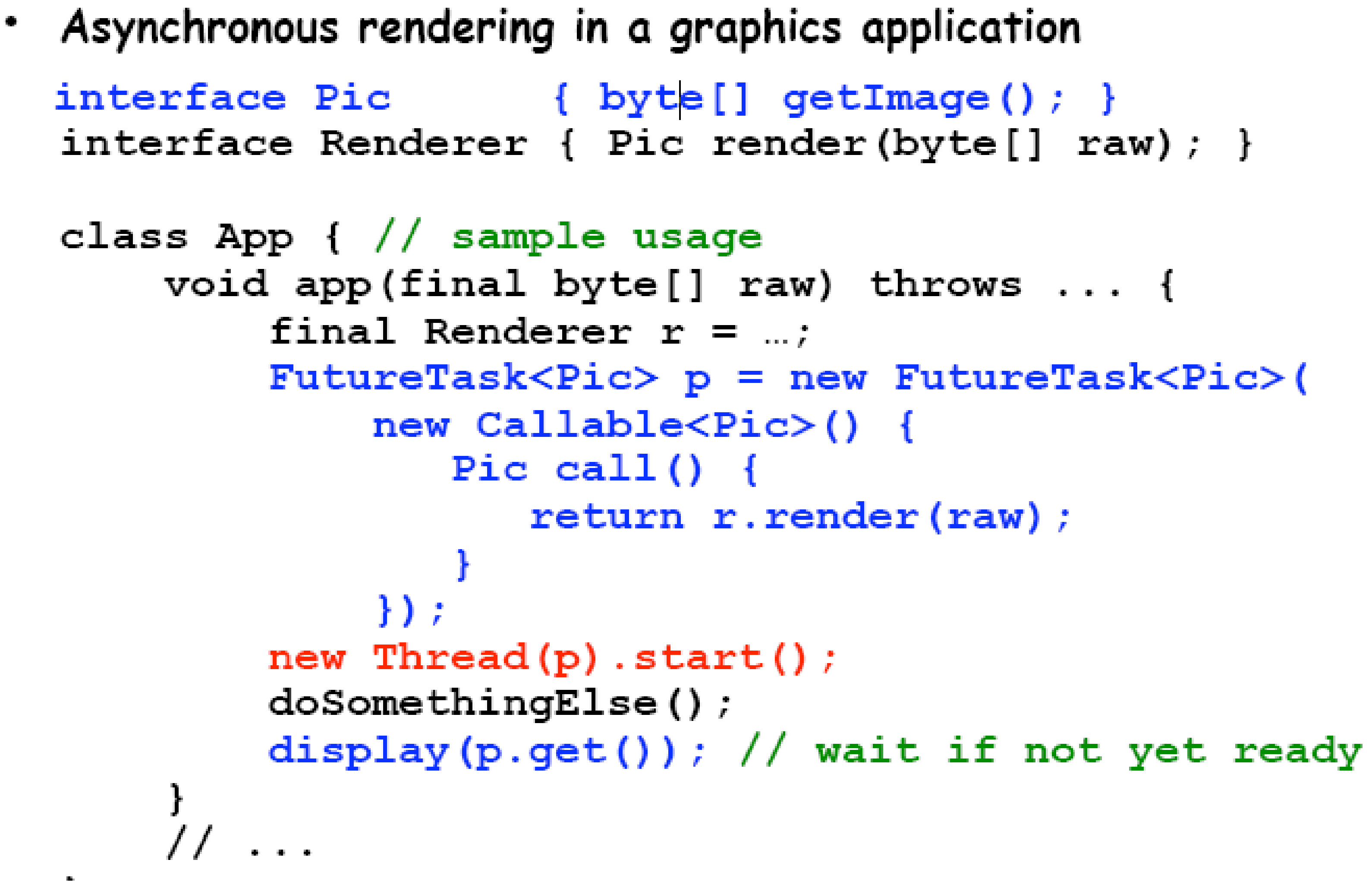
- Encapsulates waiting for the result of an asynchronous computation launched in another thread
  - The callback is encapsulated by the Future object
- Usage pattern
  - Client initiates asynchronous computation via oneway message
  - Client receives a "handle" to the result: a Future
  - Client performs additional tasks prior to using result
  - Client requests result from Future, blocking if necessary until result is available
  - Client uses result
- Assumes truly concurrent execution between client and task
  - Otherwise no point performing an asynchronous computation
- Assumes client doesn't need result immediately
  - Otherwise it may as well perform the task directly

# Canned Synchronizers-Future



- V get()
  - Retrieves the result held in this Future object, blocking if necessary until the result is available
  - Timed version throws TimeoutException
  - If cancelled then CancelledException thrown
  - If computation fails throws ExecutionException
- boolean isDone()
  - Queries if the computation has completed—whether successful, cancelled or threw an exception
- boolean isCancelled()
  - Returns true if the computation was cancelled before it completed

# Canned Synchronizers-Future



- Asynchronous rendering in a graphics application

```
interface Pic      { byte[] getImage(); }
interface Renderer { Pic render(byte[] raw); }

class App { // sample usage
    void app(final byte[] raw) throws ... {
        final Renderer r = …;
        FutureTask<Pic> p = new FutureTask<Pic>(
            new Callable<Pic>() {
               Pic call() {
                  return r.render(raw);
               }
            });
        new Thread(p).start();
        doSomethingElse();
        display(p.get()); // wait if not yet ready
    }
    // ...
.
```

# Canned Synchronizers-Semaphore

## Semaphore

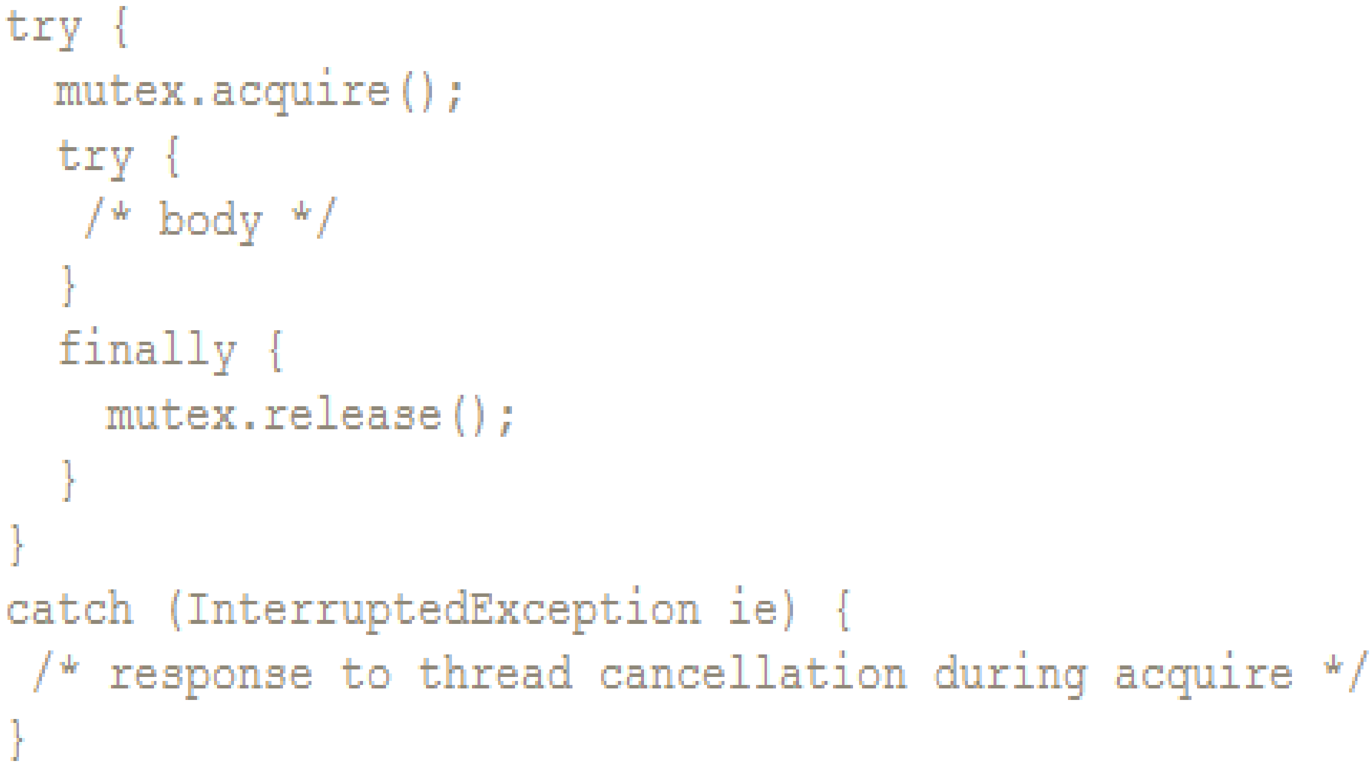
- Mutual exclusion lock guarantees that only one thread at a time can enter a critical section.
- If another thread wants to enter the critical section while it is occupied, then it blocks, suspending itself until another thread notifies it to try again.
- A Semaphore is a **generalization** of mutual exclusion locks.
- Each Semaphore has a *capacity, denoted by c. Instead of allowing* only one thread at a time into the critical section, a Semaphore allows at most *c*

# Canned Synchronizers-Semaphore

```
public class ResourcePool<T> {
  private final Semaphore sem =
      new Semaphore(MAX_RESOURCES, true);
  private final Queue<T> resources =
      new ConcurrentLinkedQueue<T>();

  public T getResource(long maxWaitMillis)
      throws InterruptedException, ResourceCreationException {

    // First, get permission to take or create a resource
    sem.tryAcquire(maxWaitMillis, TimeUnit.MILLISECONDS);

    // Then, actually take one if available...
    T res = resources.poll();
    if (res != null)
      return res;

    // ...or create one if none available
    try {
      return createResource();
    } catch (Exception e) {
      // Don't hog the permit if we failed to create a resource!
      sem.release();
      throw new ResourceCreationException(e);
    }
  }
  public void returnResource(T res) {
    resources.add(res);
    sem.release();
  }
}
```

# Canned Synchronizers-Mutex

- **Mutex are semaphore with the count of one and can be used like a lock but more verbose.**

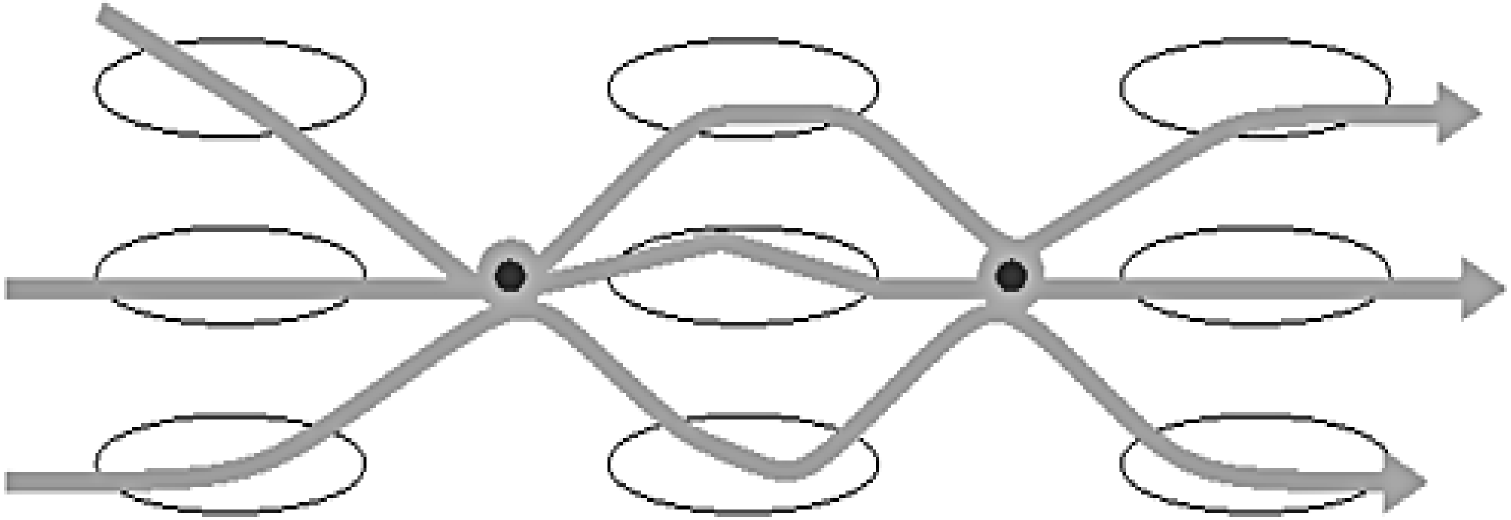


```
try {
  mutex.acquire();
  try {
   /* body */
  }
  finally {
    mutex.release();
  }
}
catch (InterruptedException ie) {
 /* response to thread cancellation during acquire */
}
```

# Canned Synchronizers-Barrier

## Barrier

- With barriers, many parallel iterative algorithms become easy to express.



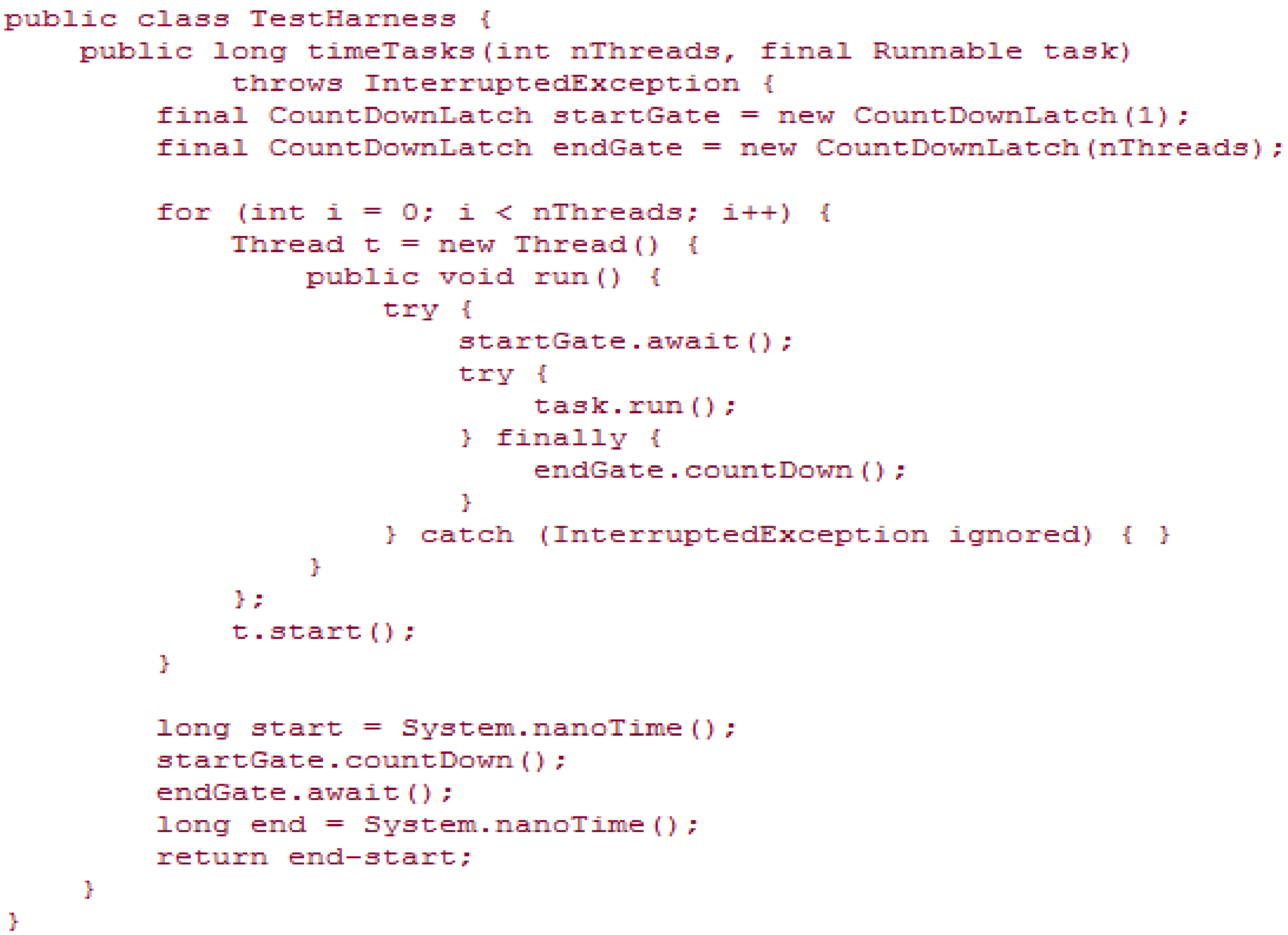
```
class Segment implements Runnable  {            // Code sketch
  final CyclicBarrier bar; // shared by all segments
  Segment(CyclicBarrier b, ...) { bar = b; ...; }

  void update() { ... }

  public void run() {
    // ...
    for (int i = 0; i < iterations; ++i) {
      update();
      bar.barrier();
    }
    // ...
  }
}

class Driver {
  // ...
  void compute(Problem problem) throws ... {
    int n = problem.size / granularity;
    CyclicBarrier barrier = new CyclicBarrier(n);
    Thread[] threads = new Thread[n];

    // create
    for (int i = 0; i < n; ++i)
      threads[i] = new Thread(new Segment(barrier, ...));

    // trigger
    for (int i = 0; i < n; ++i) threads[i].start();

    // await termination
    for (int i = 0; i < n; ++i) threads[i].join();
  }
}
```

# Canned Synchronizers-Latch

## Latch

- A latch acts as a gate: until the latch reaches the terminal state the gate is closed and no thread can pass, and in the terminal state the gate opens, allowing all threads to pass.
- Once the latch reaches the terminal state, it cannot change state again, so it remains open forever.
- Usage:
  - Ensuring that a computation does not proceed until resources it needs have been initialized
  - Waiting until all the parties involved in an activity, for instance the players in a multi-player game, are ready to proceed

# Canned Synchronizers-Latch



```
public class TestHarness {
    public long timeTasks(int nThreads, final Runnable task)
            throws InterruptedException {
        final CountDownLatch startGate = new CountDownLatch(1);
        final CountDownLatch endGate = new CountDownLatch(nThreads);

        for (int i = 0; i < nThreads; i++) {
            Thread t = new Thread() {
                public void run() {
                    try {
                        startGate.await();
                        try {
                            task.run();
                        } finally {
                            endGate.countDown();
                        }
                    } catch (InterruptedException ignored) { }
                }
            };
            t.start();
        }

        long start = System.nanoTime();
        startGate.countDown();
        endGate.await();
        long end = System.nanoTime();
        return end-start;
    }
}
```

# Canned Synchronizers- SynchronousQueue

## SynchronousQueue

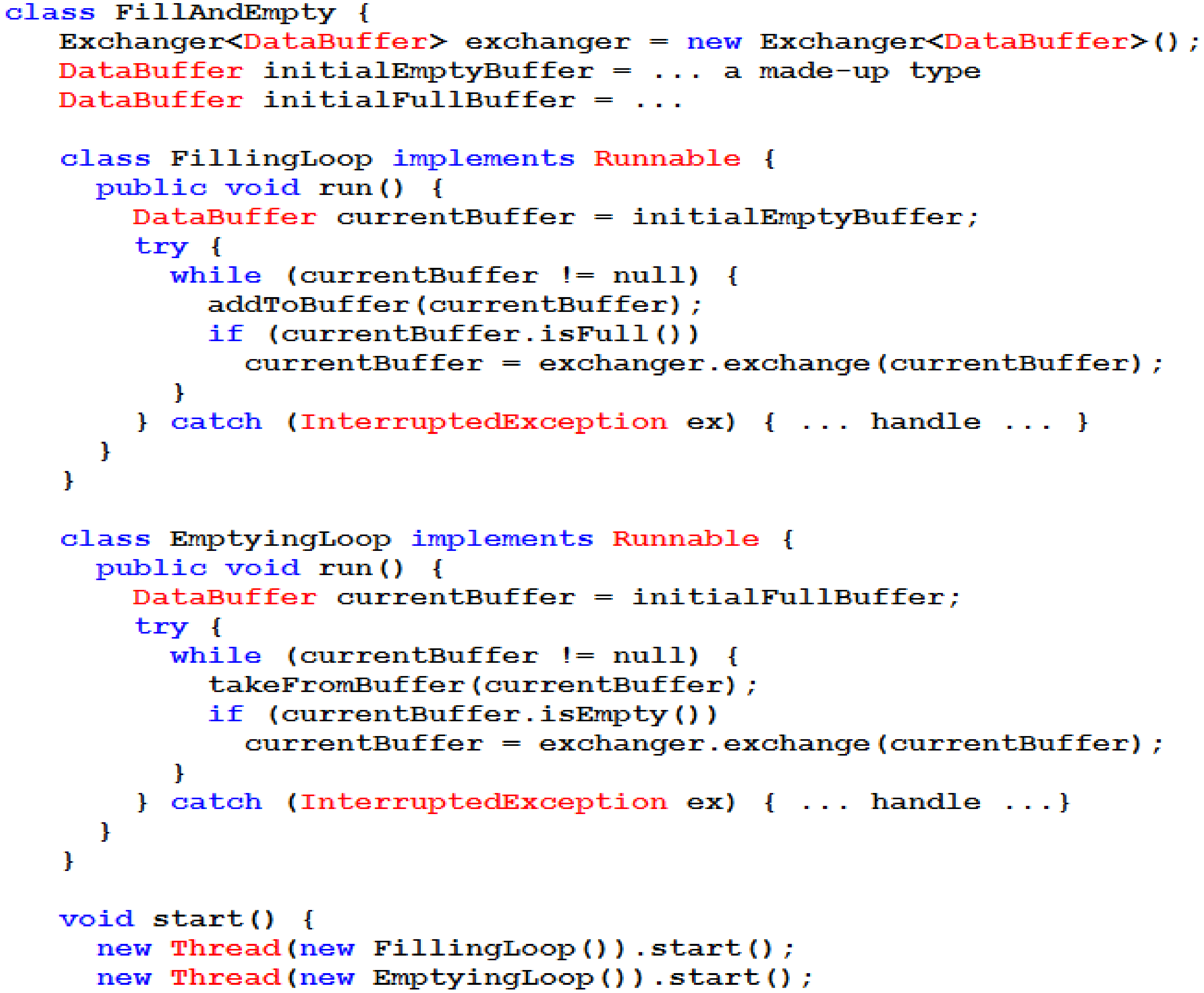
- It implements BlockingQueue
- SynchronousQueue, is not really a queue at all, in that it maintains no storage space for queued elements.
- When the handoff is accepted, it knows a consumer has taken responsibility for it, rather than simply letting it sit on a queue somewhere.
- Usage:
  - Synchronous queues are generally suitable only when there are enough consumers that there nearly always will be one ready to take the handoff.
  - As a one way rendezvous

# Canned Synchronizers-Exchanger

## Exchanger

- An exchanger acts as a synchronous channel except that instead of supporting two methods, put and take, it supports only one method, rendezvous (sometimes just called exchange) that combines their effects .
- This operation takes an argument representing an Object offered by one thread to another, and returns the Object offered by the other thread

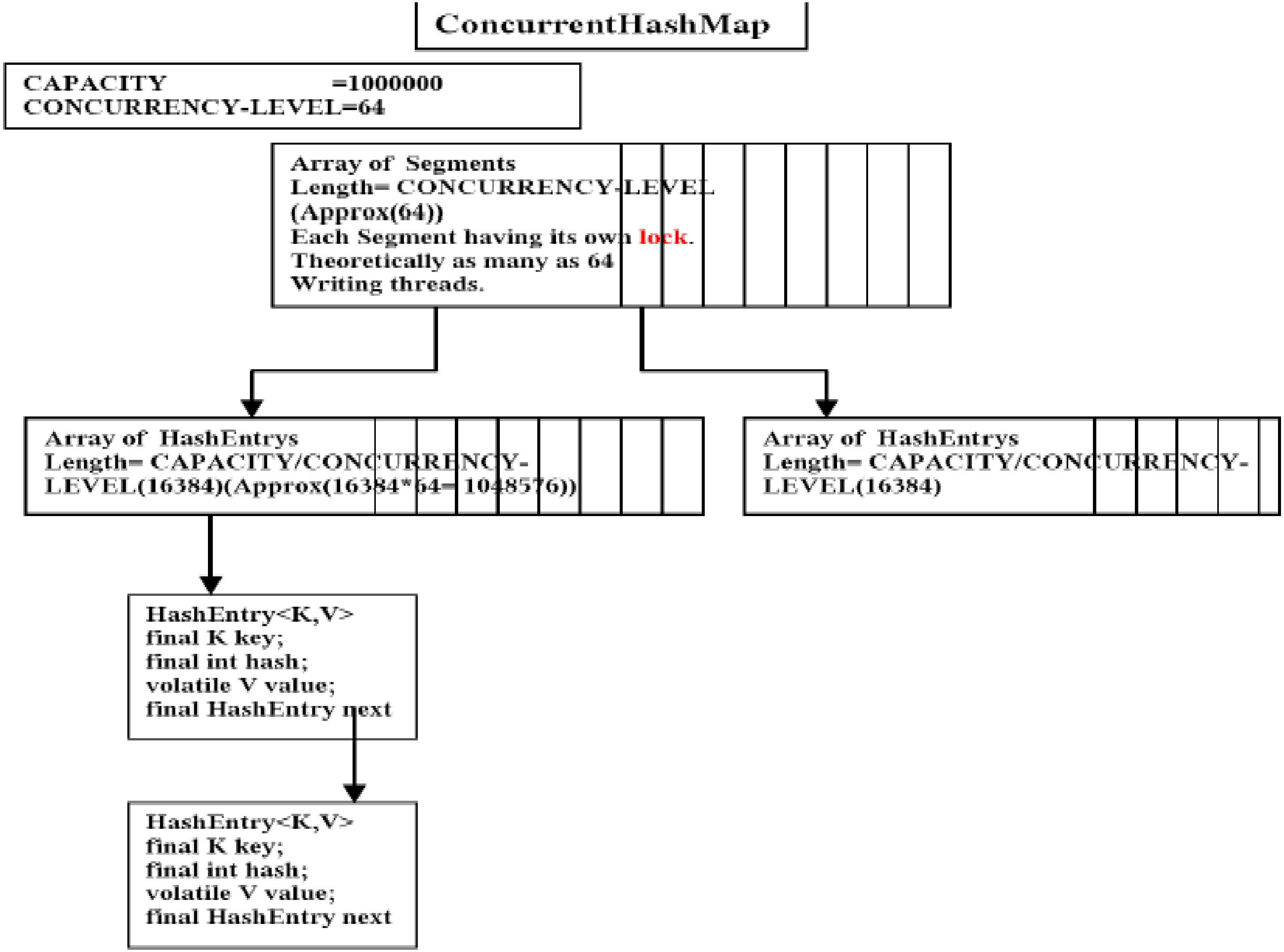
# Canned Synchronizers- Exchanger



```
class FillAndEmpty {
   Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>();
   DataBuffer initialEmptyBuffer = ... a made-up type
   DataBuffer initialFullBuffer = ...

   class FillingLoop implements Runnable {
     public void run() {
       DataBuffer currentBuffer = initialEmptyBuffer;
       try {
         while (currentBuffer != null) {
           addToBuffer(currentBuffer);
           if (currentBuffer.isFull())
             currentBuffer = exchanger.exchange(currentBuffer);
         }
       } catch (InterruptedException ex) { ... handle ... }
     }
   }

   class EmptyingLoop implements Runnable {
     public void run() {
       DataBuffer currentBuffer = initialFullBuffer;
       try {
         while (currentBuffer != null) {
           takeFromBuffer(currentBuffer);
           if (currentBuffer.isEmpty())
             currentBuffer = exchanger.exchange(currentBuffer);
         }
       } catch (InterruptedException ex) { ... handle ...}
     }
   }

   void start() {
     new Thread(new FillingLoop()).start();
     new Thread(new EmptyingLoop()).start();
```

# Concurrent Data Structures

# Concurrent Data Structures-ConcurrentHashMap



ConcurrentHashMap

CAPACITY =1000000
CONCURRENCY-LEVEL=64

Array of Segments
Length= CONCURRENCY-LEVEL
(Approx(64))
Each Segment having its own lock.
Theoretically as many as 64
Writing threads.

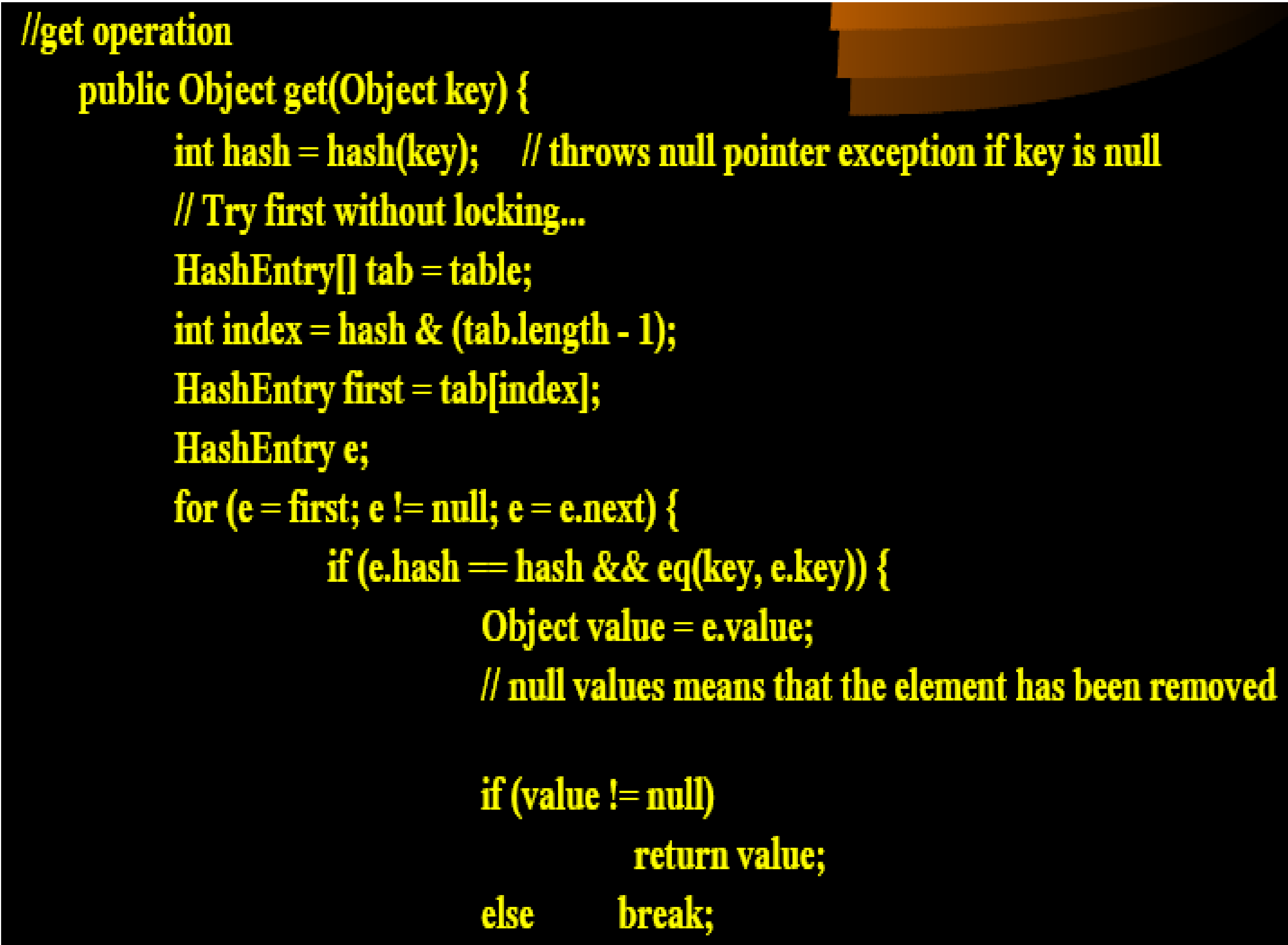
Array of HashEntrys
Length= CAPACITY/CONCURRENCY-LEVEL(16384)(Approx(16384*64= 1048576))

Array of HashEntrys
Length= CAPACITY/CONCURRENCY-LEVEL(16384)

HashEntry<K,V>
final K key;
final int hash;
volatile V value;
final HashEntry next

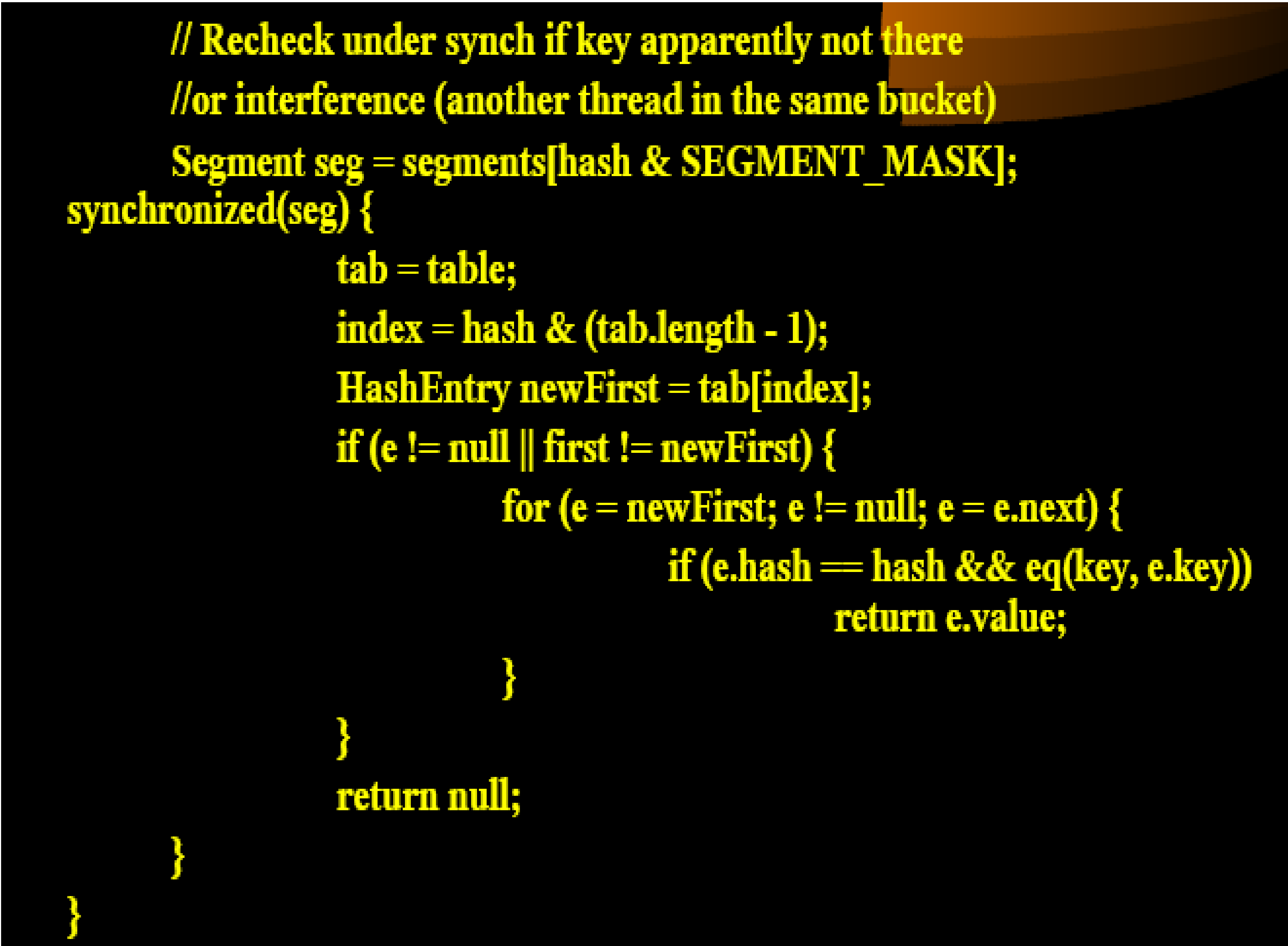
HashEntry<K,V>
final K key;
final int hash;
volatile V value;
final HashEntry next

# Concurrent Data Structures- ConcurrentHashMap



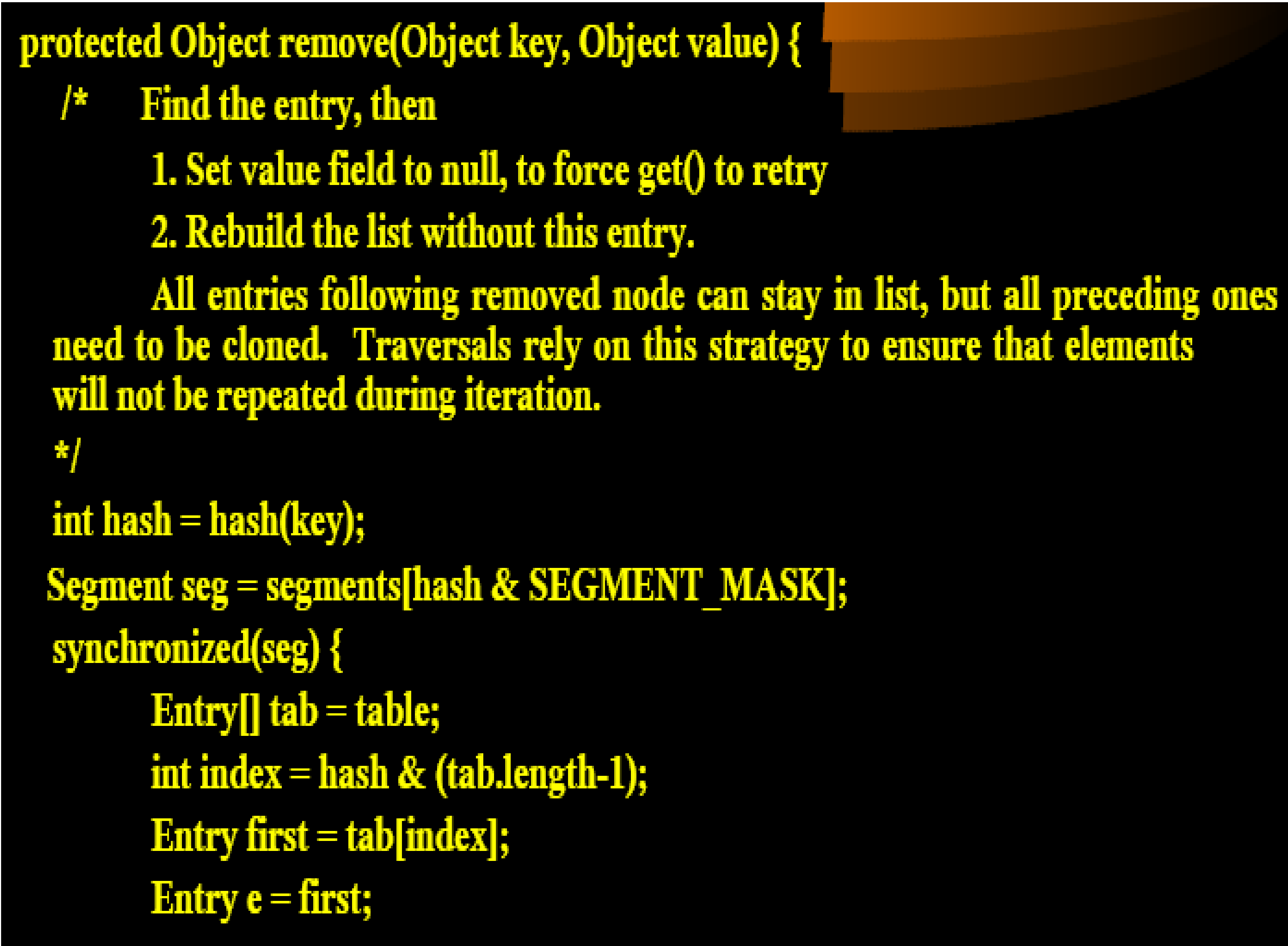
```
//get operation
    public Object get(Object key) {
        int hash = hash(key);    // throws null pointer exception if key is null
        // Try first without locking...
        HashEntry[] tab = table;
        int index = hash & (tab.length - 1);
        HashEntry first = tab[index];
        HashEntry e;
        for (e = first; e != null; e = e.next) {
            if (e.hash == hash && eq(key, e.key)) {
                Object value = e.value;
                // null values means that the element has been removed

                if (value != null)
                    return value;
                else    break;
```

# Concurrent Data Structures- ConcurrentHashMap



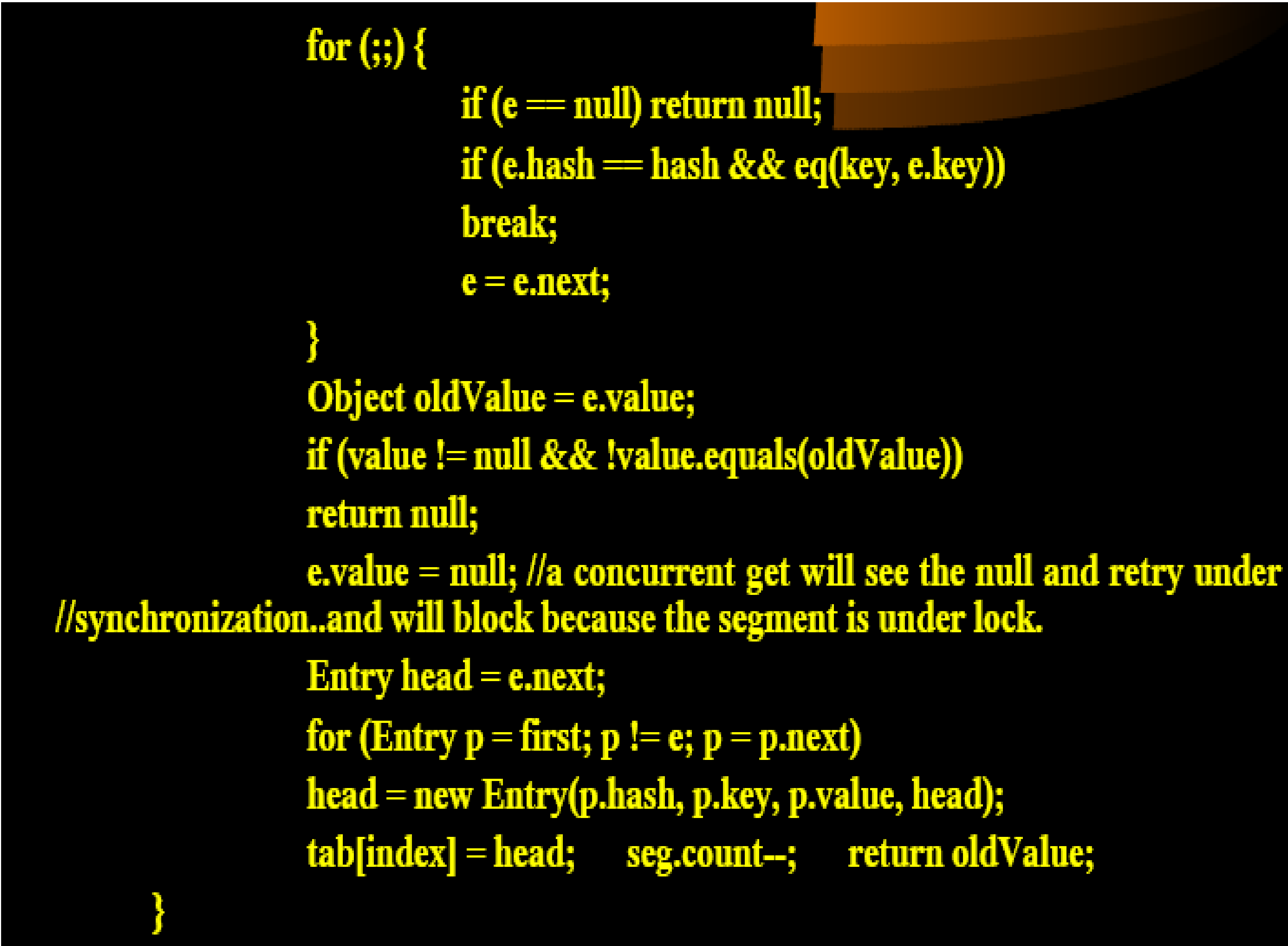
```
        // Recheck under synch if key apparently not there
        //or interference (another thread in the same bucket)
        Segment seg = segments[hash & SEGMENT_MASK];
    synchronized(seg) {
                tab = table;
                index = hash & (tab.length - 1);
                HashEntry newFirst = tab[index];
                if (e != null || first != newFirst) {
                        for (e = newFirst; e != null; e = e.next) {
                                if (e.hash == hash && eq(key, e.key))
                                        return e.value;
                        }
                }
                return null;
        }
    }
```

# Concurrent Data Structures- ConcurrentHashMap



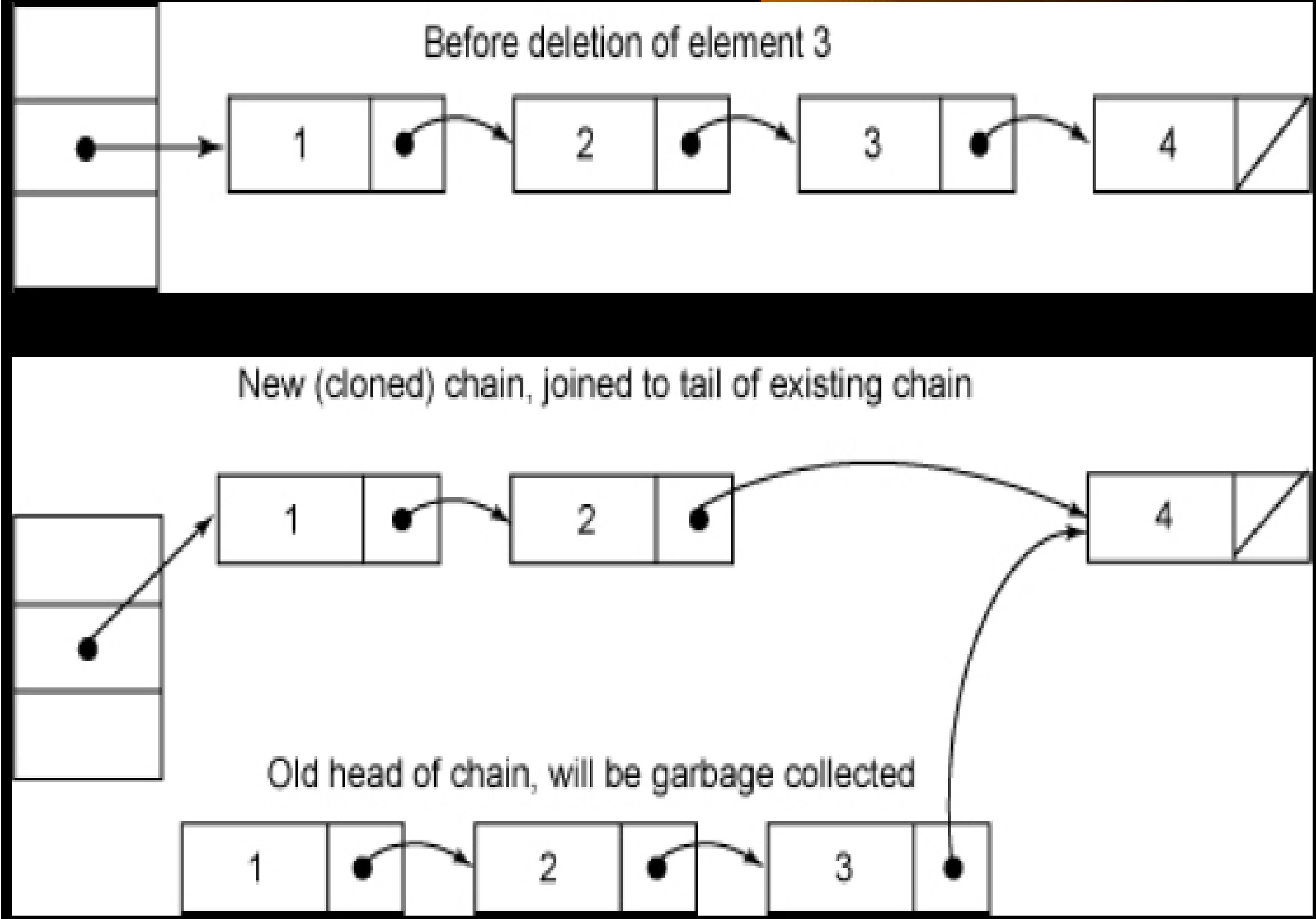
```
protected Object remove(Object key, Object value) {
    /*     Find the entry, then
             1. Set value field to null, to force get() to retry
             2. Rebuild the list without this entry.
             All entries following removed node can stay in list, but all preceding ones
    need to be cloned.  Traversals rely on this strategy to ensure that elements
    will not be repeated during iteration.
    */
    int hash = hash(key);
   Segment seg = segments[hash & SEGMENT_MASK];
    synchronized(seg) {
             Entry[] tab = table;
             int index = hash & (tab.length-1);
             Entry first = tab[index];
             Entry e = first;
```

# Concurrent Data Structures- ConcurrentHashMap

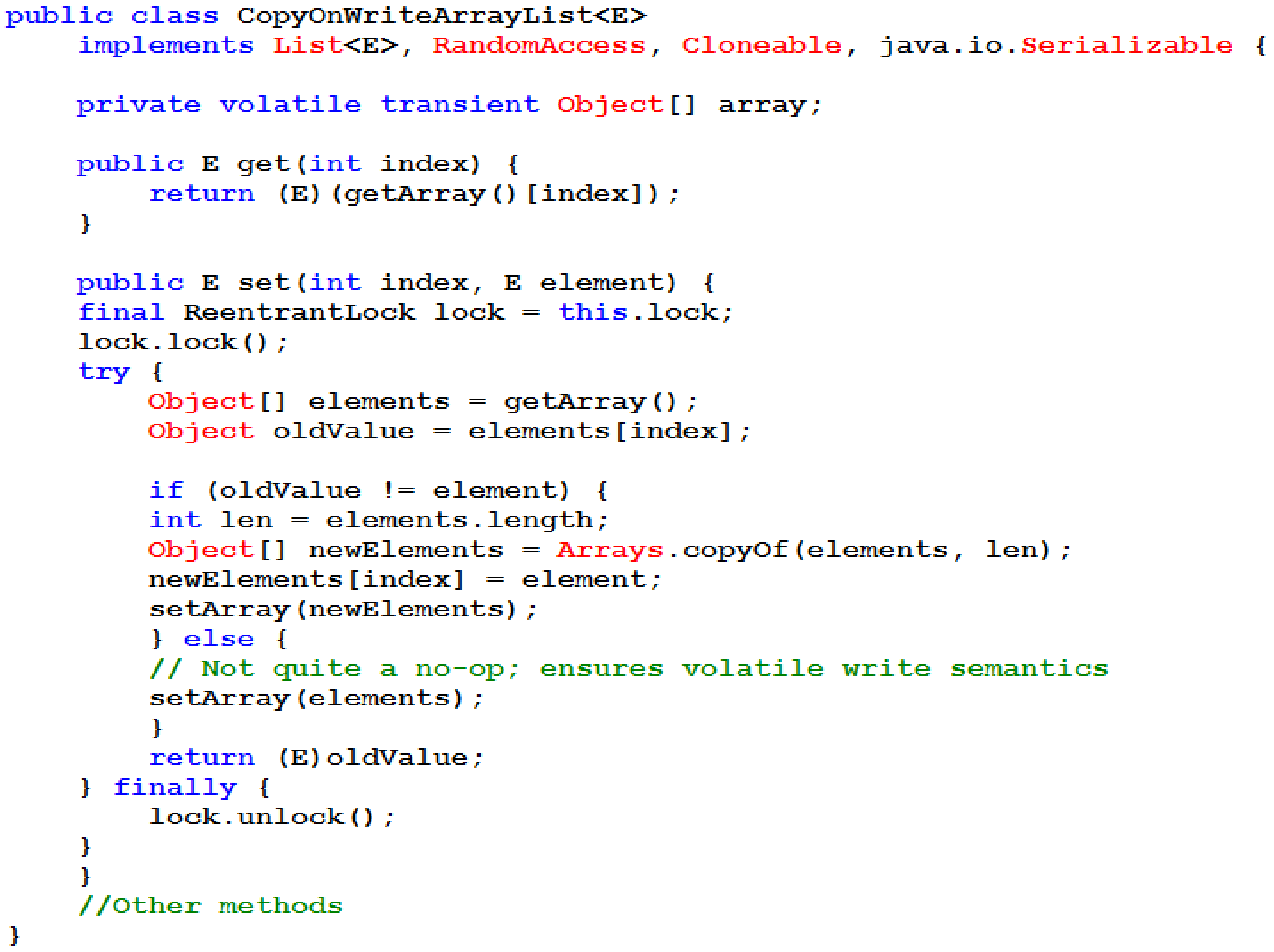


```
                    for (;;) {
                              if (e == null) return null;
                              if (e.hash == hash && eq(key, e.key))
                              break;
                              e = e.next;
                    }
                    Object oldValue = e.value;
                    if (value != null && !value.equals(oldValue))
                    return null;
                    e.value = null; //a concurrent get will see the null and retry under
   //synchronization..and will block because the segment is under lock.
                    Entry head = e.next;
                    for (Entry p = first; p != e; p = p.next)
                    head = new Entry(p.hash, p.key, p.value, head);
                    tab[index] = head;    seg.count--;    return oldValue;
          }
```

# Concurrent Data Structures-ConcurrentHashMap

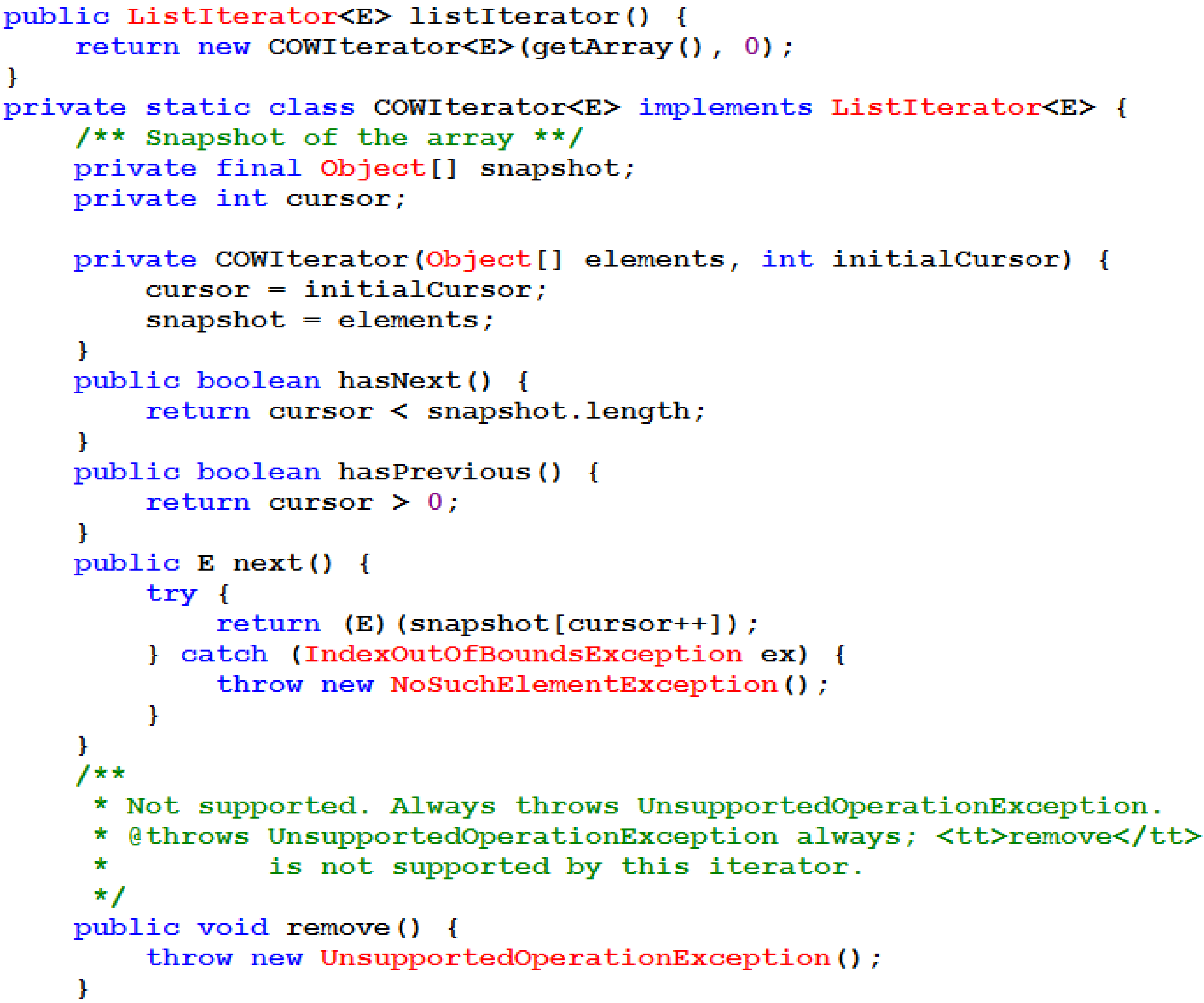

Before deletion of element 3
1
2
3
4
New (cloned) chain, joined to tail of existing chain
1
2
4
Old head of chain, will be garbage collected
1
2
3

# Concurrent Data Structures-CopyOnWriteArray(List/Set)



```
public class CopyOnWriteArrayList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

    private volatile transient Object[] array;

    public E get(int index) {
        return (E)(getArray()[index]);
    }

    public E set(int index, E element) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        Object oldValue = elements[index];

        if (oldValue != element) {
        int len = elements.length;
        Object[] newElements = Arrays.copyOf(elements, len);
        newElements[index] = element;
        setArray(newElements);
        } else {
        // Not quite a no-op; ensures volatile write semantics
        setArray(elements);
        }
        return (E)oldValue;
    } finally {
        lock.unlock();
    }
    }
    //Other methods
}
```

# Concurrent Data Structures- CopyOnWriteArray(List/Set)



```java
public ListIterator<E> listIterator() {
    return new COWIterator<E>(getArray(), 0);
}
private static class COWIterator<E> implements ListIterator<E> {
    /** Snapshot of the array **/
    private final Object[] snapshot;
    private int cursor;

    private COWIterator(Object[] elements, int initialCursor) {
        cursor = initialCursor;
        snapshot = elements;
    }
    public boolean hasNext() {
        return cursor < snapshot.length;
    }
    public boolean hasPrevious() {
        return cursor > 0;
    }
    public E next() {
        try {
            return (E)(snapshot[cursor++]);
        } catch (IndexOutOfBoundsException ex) {
            throw new NoSuchElementException();
        }
    }
    /**
     * Not supported. Always throws UnsupportedOperationException.
     * @throws UnsupportedOperationException always; <tt>remove</tt>
     *         is not supported by this iterator.
     */
    public void remove() {
        throw new UnsupportedOperationException();
    }
```

# Queue Interfaces

- Queue
- BlockingQueue
- Deque
- BlockingDeque

# Queue Interfaces-Queue

| | *Throws exception* | *Returns special value* |
|---|---|---|
| **Insert** | `add(e)` | `offer(e)` |
| **Remove** | `remove()` | `poll()` |
| **Examine** | `element()` | `peek()` |

# Queue Interfaces- BlockingQueue

| | *Exception* | *Special-value* | *Blocks* | *Times out* |
|---|---|---|---|---|
| **Insert** | `add(e)` | `offer(e)` | `put(e)` | `offer(e,time,unit)` |
| **Remove** | `remove()` | `poll()` | `take()` | `poll(time,unit)` |
| **Examine** | `element()` | `peek()` | *N/A* | *N/A* |

# Queue Interfaces-Deque

| | First Element (Head) | | Last Element (Tail) | |
|---|---|---|---|---|
| | *exception* | *special value* | *exception* | *special value* |
| **Insert** | `addFirst(e)` | `offerFirst(e)` | `addLast(e)` | `offerLast(e)` |
| **Remove** | `removeFirst()` | `pollFirst()` | `removeLast()` | `pollLast()` |
| **Examine** | `getFirst()` | `peekFirst()` | `getLast()` | `peekLast()` |

# Deque - Queue Equivalents

| Queue Method | Equivalent Deque Method |
|---|---|
| `offer(e)` | `offerLast(e)` |
| `add(e)` | `addLast(e)` |
| `poll()` | `pollFirst()` |
| `remove()` | `removeFirst()` |
| `peek()` | `peekFirst()` |
| `element()` | `getFirst()` |

# Using a Deque as a Stack

| Stack Method | Equivalent Deque Method |
|---|---|
| `push(e)` | `addFirst(e)` |
| `pop()` | `removeFirst()` |
| `peek()` | `peekFirst()` |

# Queue Interfaces-BlockingDeque

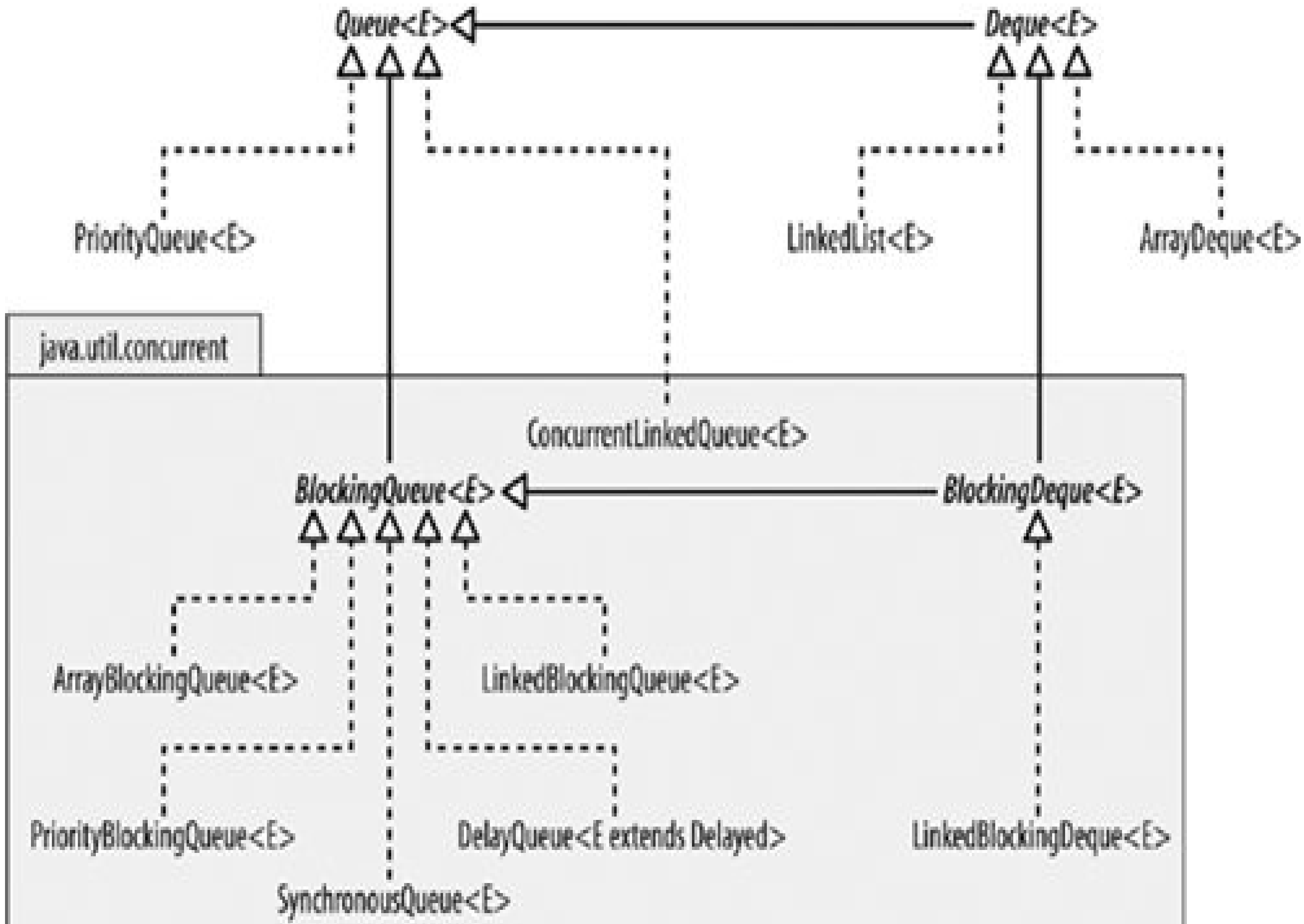
**First Element (Head)**

| | *Block* | *Time out* |
|---|---|---|
| **Insert** | `putFirst(e)` | `offerFirst(e,time,unit)` |
| **Remove** | `takeFirst()` | `pollFirst(time,unit)` |

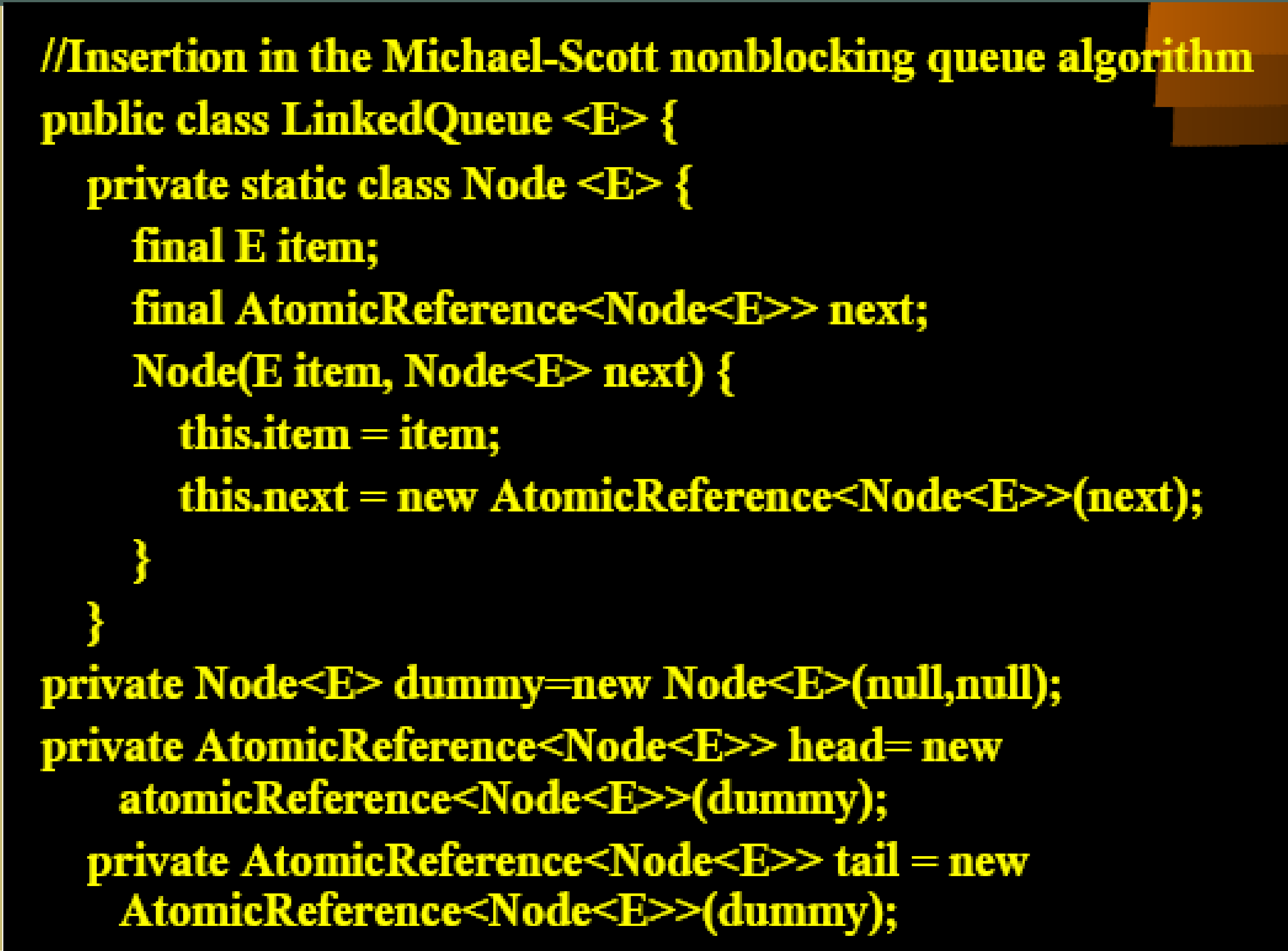
**Last Element (Tail)**

| | *Block* | *Time out* |
|---|---|---|
| **Insert** | `putLast(e)` | `offerLast(e,time,unit)` |
| **Remove** | `takeLast()` | `pollLast(time,unit)` |

# Queue Implementations

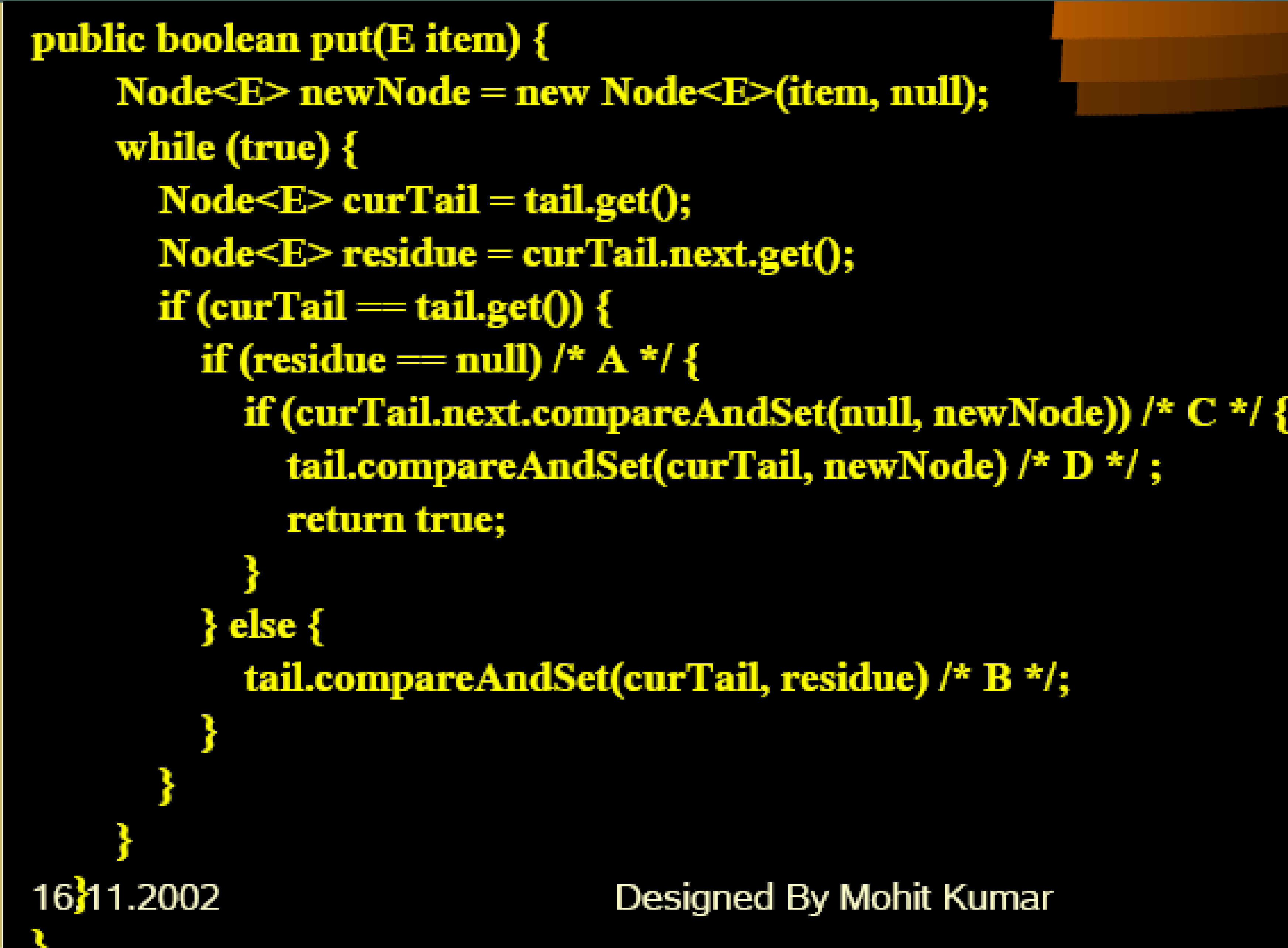


# ConcurrentLinkedQueue



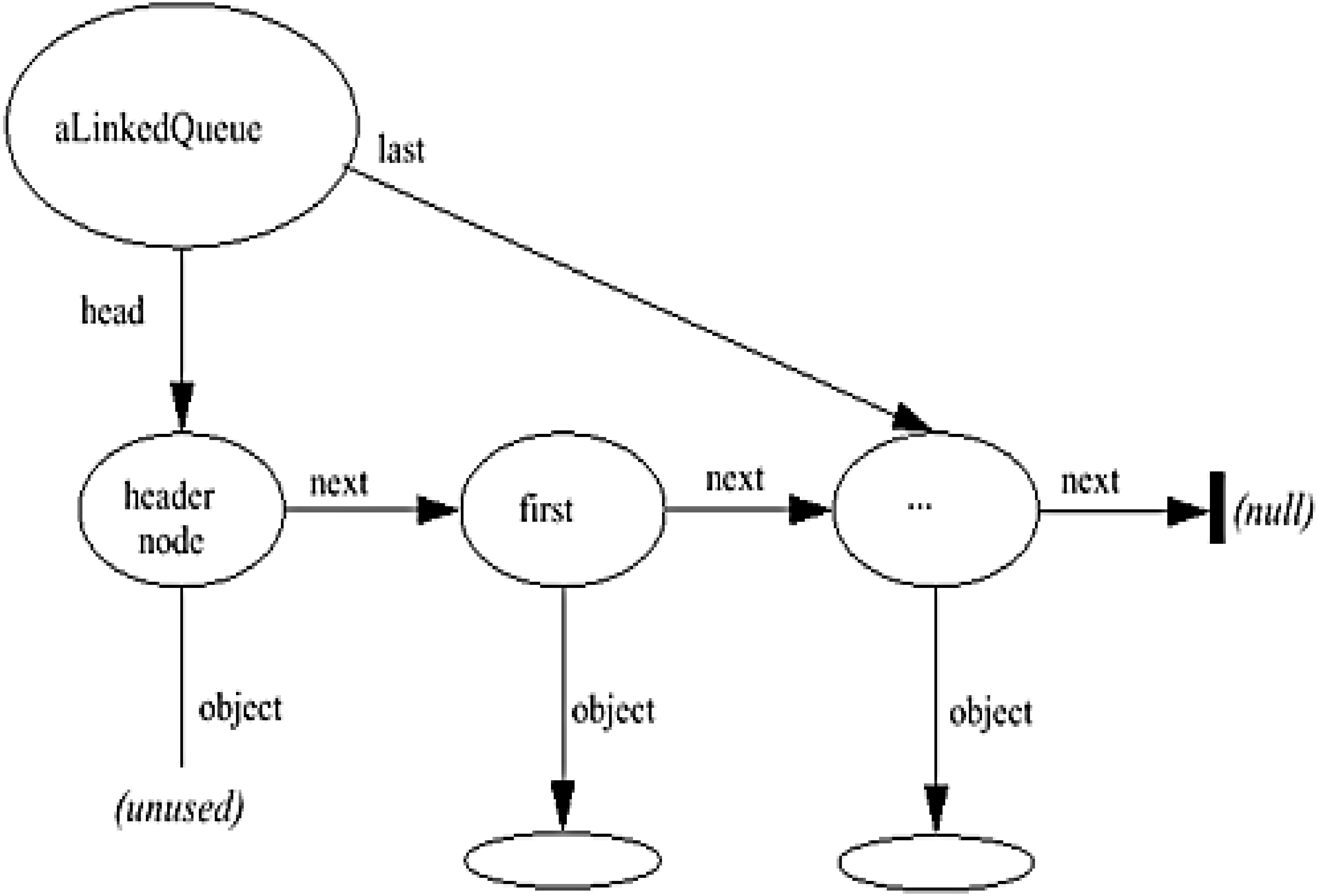
```
//Insertion in the Michael-Scott nonblocking queue algorithm
public class LinkedQueue <E> {
    private static class Node <E> {
        final E item;
        final AtomicReference<Node<E>> next;
        Node(E item, Node<E> next) {
            this.item = item;
            this.next = new AtomicReference<Node<E>>(next);
        }
    }
private Node<E> dummy=new Node<E>(null,null);
private AtomicReference<Node<E>> head= new
      atomicReference<Node<E>>(dummy);
    private AtomicReference<Node<E>> tail = new
      AtomicReference<Node<E>>(dummy);
```

# ConcurrentLinkedQueue



```
public boolean put(E item) {
    Node<E> newNode = new Node<E>(item, null);
    while (true) {
      Node<E> curTail = tail.get();
      Node<E> residue = curTail.next.get();
      if (curTail == tail.get()) {
        if (residue == null) /* A */ {
          if (curTail.next.compareAndSet(null, newNode)) /* C */ {
            tail.compareAndSet(curTail, newNode) /* D */ ;
            return true;
          }
        } else {
          tail.compareAndSet(curTail, residue) /* B */;
        }
      }
    }
  }
}
```

# LinkedBlockingQueue

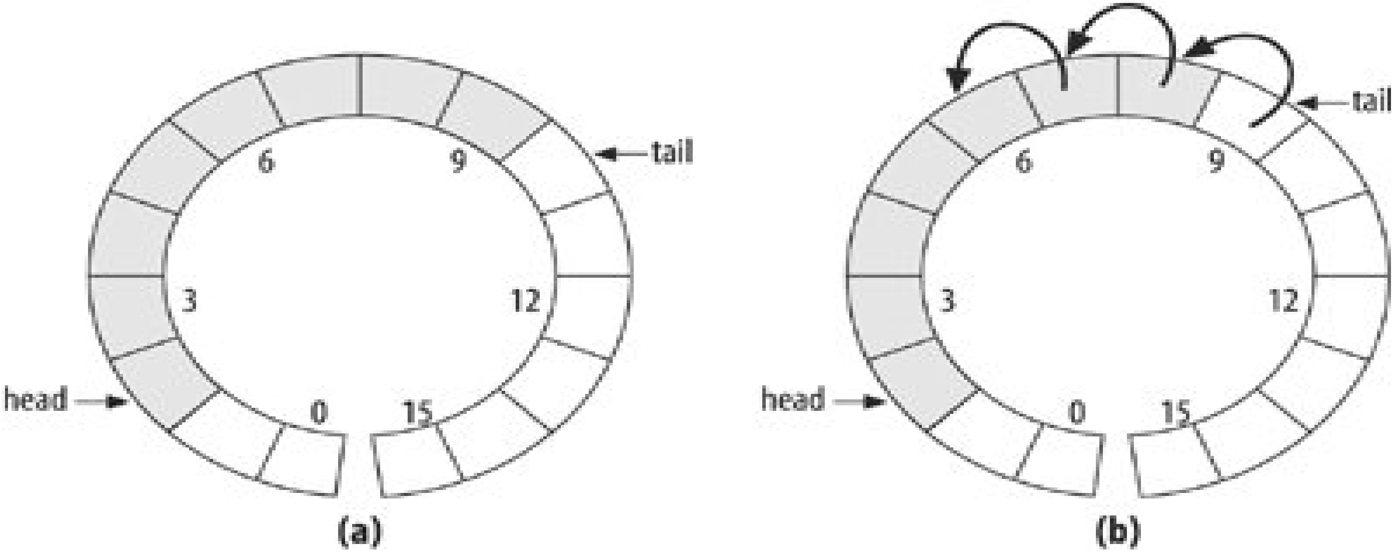


```
class LinkedQueue {
 protected Node head = new Node(null);
 protected Node last = head;

 protected final Object pollLock = new Object();
 protected final Object putLock = new Object();

 public void put(Object x) {
  Node node = new Node(x);
  synchronized (putLock) {   // insert at end of list
   synchronized (last) {
    last.next = node;    // extend list
    last = node;
   }
  }
 }

 public Object poll() {   // returns null if empty
  synchronized (pollLock) {
   synchronized (head) {
    Object x = null;
    Node first = head.next;  // get to first real node
    if (first != null) {
      x = first.object;
      first.object = null;  // forget old object
      head = first;       // first becomes new head
    }
    return x;
   }
  }
 }

 static class Node {          // local node class for queue
  Object object;
  Node next = null;

  Node(Object x) { object = x; }
```

# ArrayDeque and ArrayBlockingDeque

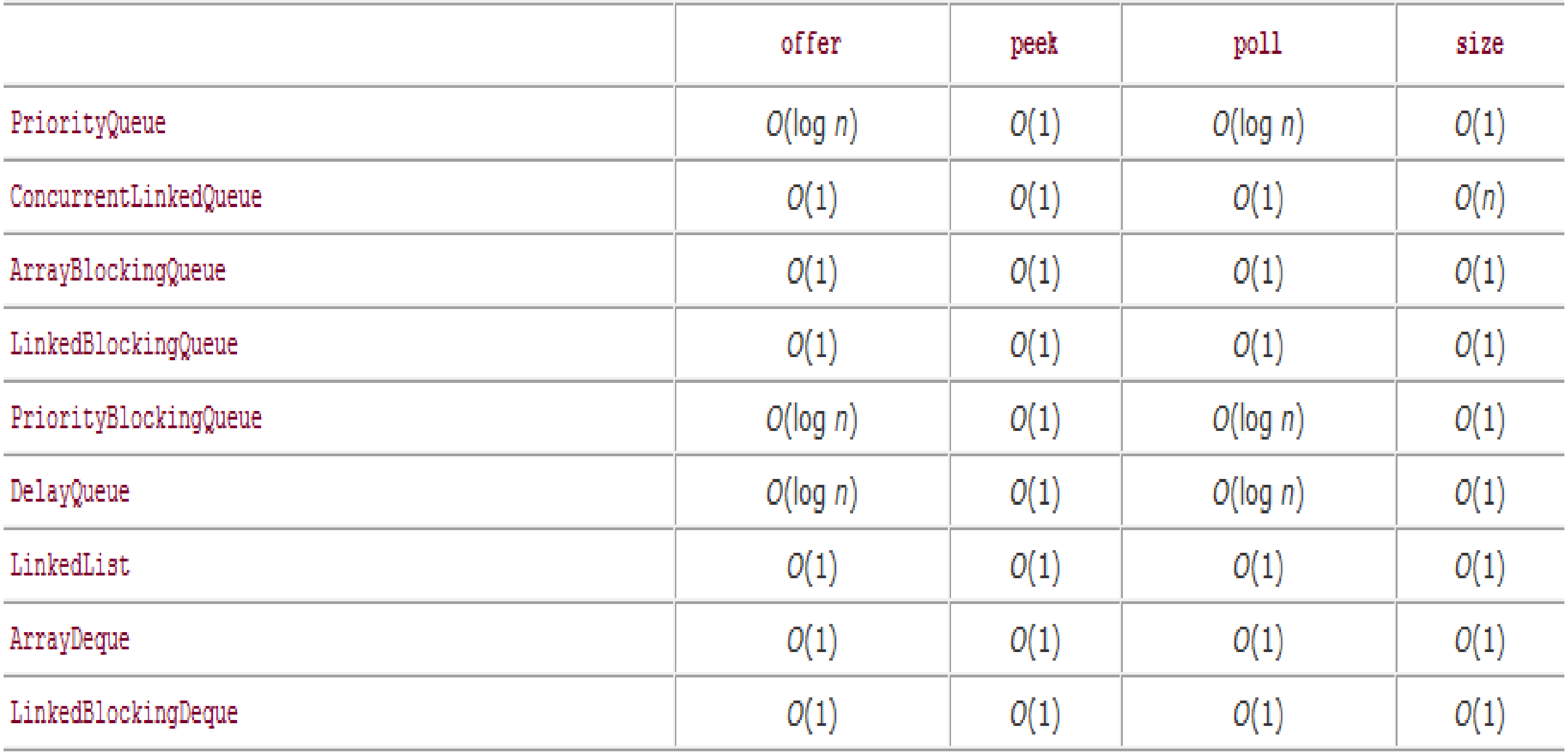
## Array Based Queues

- This implementation is based on a circular array.
- only the elements at the ends of the queue can be inserted and removed in constant time. If an element is to be removed from near the middle, which can be done for queues via the method Iterator.remove, then all the elements from one end must be moved along to maintain a compact representation.



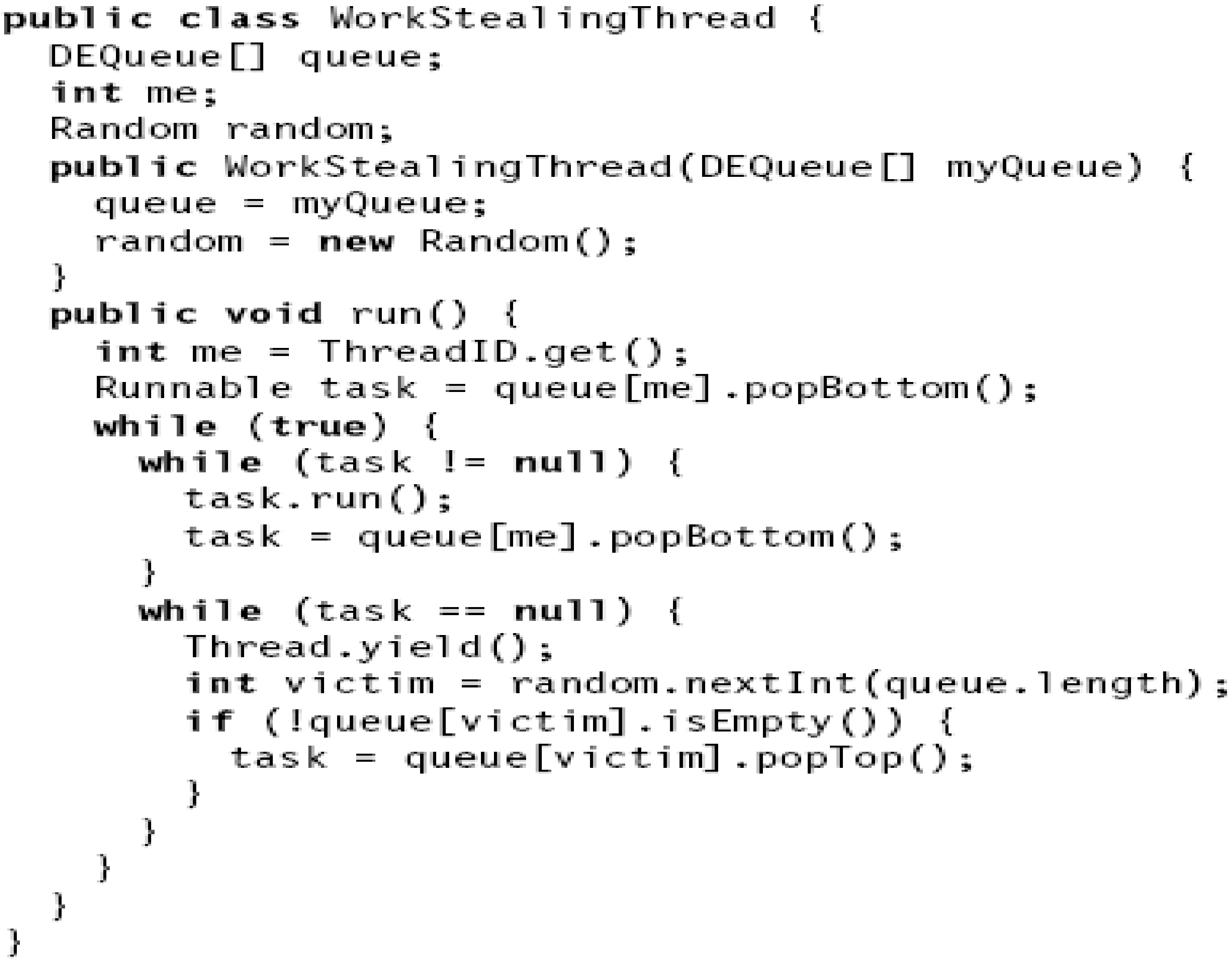
# Formulaic Performance

- These are only the formulae for sequential access; how they perform in concurrent use is a different question



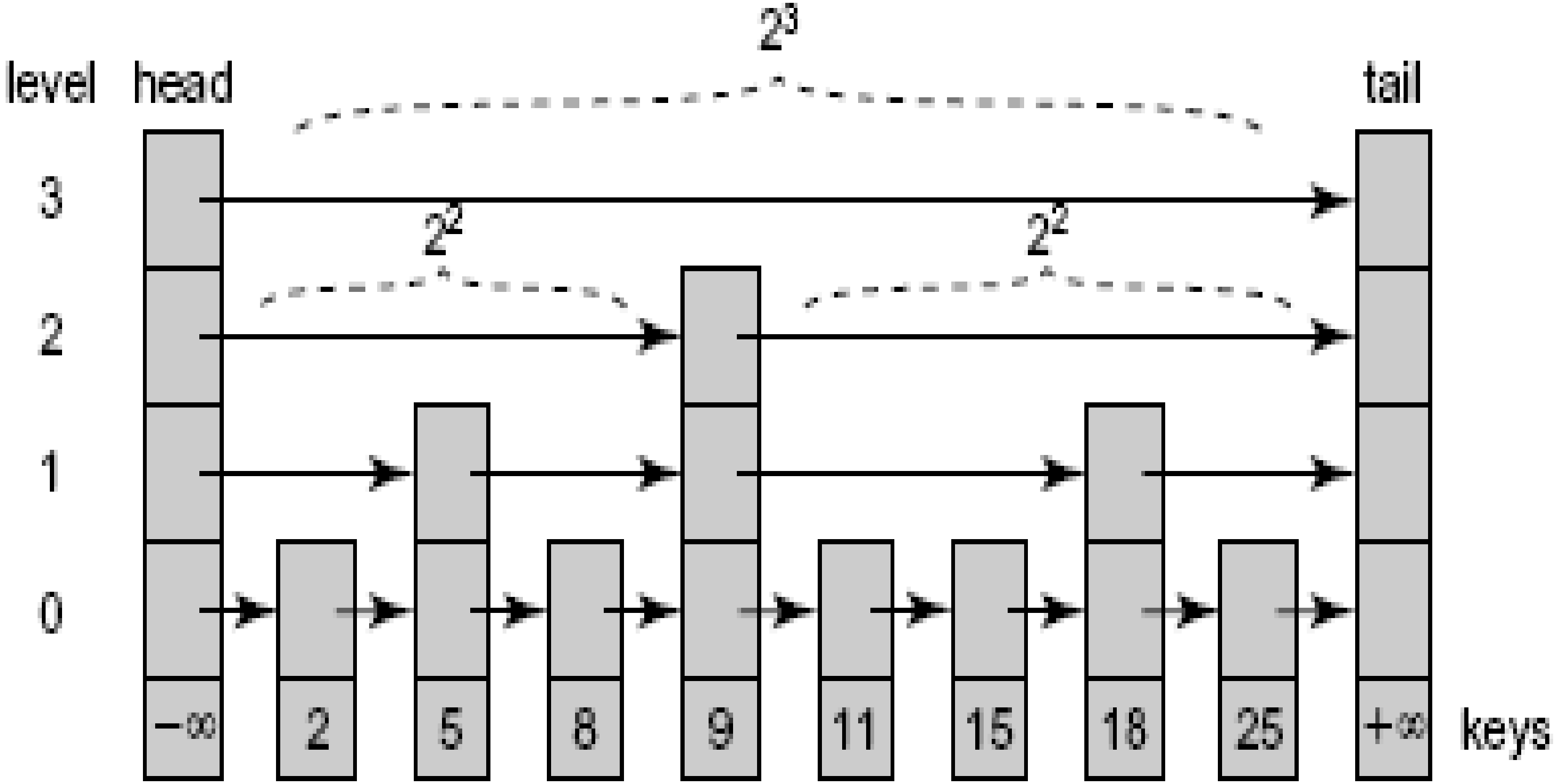
| | offer | peek | poll | size |
|---|---|---|---|---|
| PriorityQueue | $O(\log n)$ | $O(1)$ | $O(\log n)$ | $O(1)$ |
| ConcurrentLinkedQueue | $O(1)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| ArrayBlockingQueue | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| LinkedBlockingQueue | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| PriorityBlockingQueue | $O(\log n)$ | $O(1)$ | $O(\log n)$ | $O(1)$ |
| DelayQueue | $O(\log n)$ | $O(1)$ | $O(\log n)$ | $O(1)$ |
| LinkedList | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| ArrayDeque | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| LinkedBlockingDeque | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |

# Deques used for WorkStealing



```
public class WorkStealingThread {
  DEQueue[] queue;
  int me;
  Random random;
  public WorkStealingThread(DEQueue[] myQueue) {
    queue = myQueue;
    random = new Random();
  }
  public void run() {
    int me = ThreadID.get();
    Runnable task = queue[me].popBottom();
    while (true) {
      while (task != null) {
        task.run();
        task = queue[me].popBottom();
      }
      while (task == null) {
        Thread.yield();
        int victim = random.nextInt(queue.length);
        if (!queue[victim].isEmpty()) {
          task = queue[victim].popTop();
        }
      }
    }
  }
}
```

# ConcurrentSkipList(Set / Map)



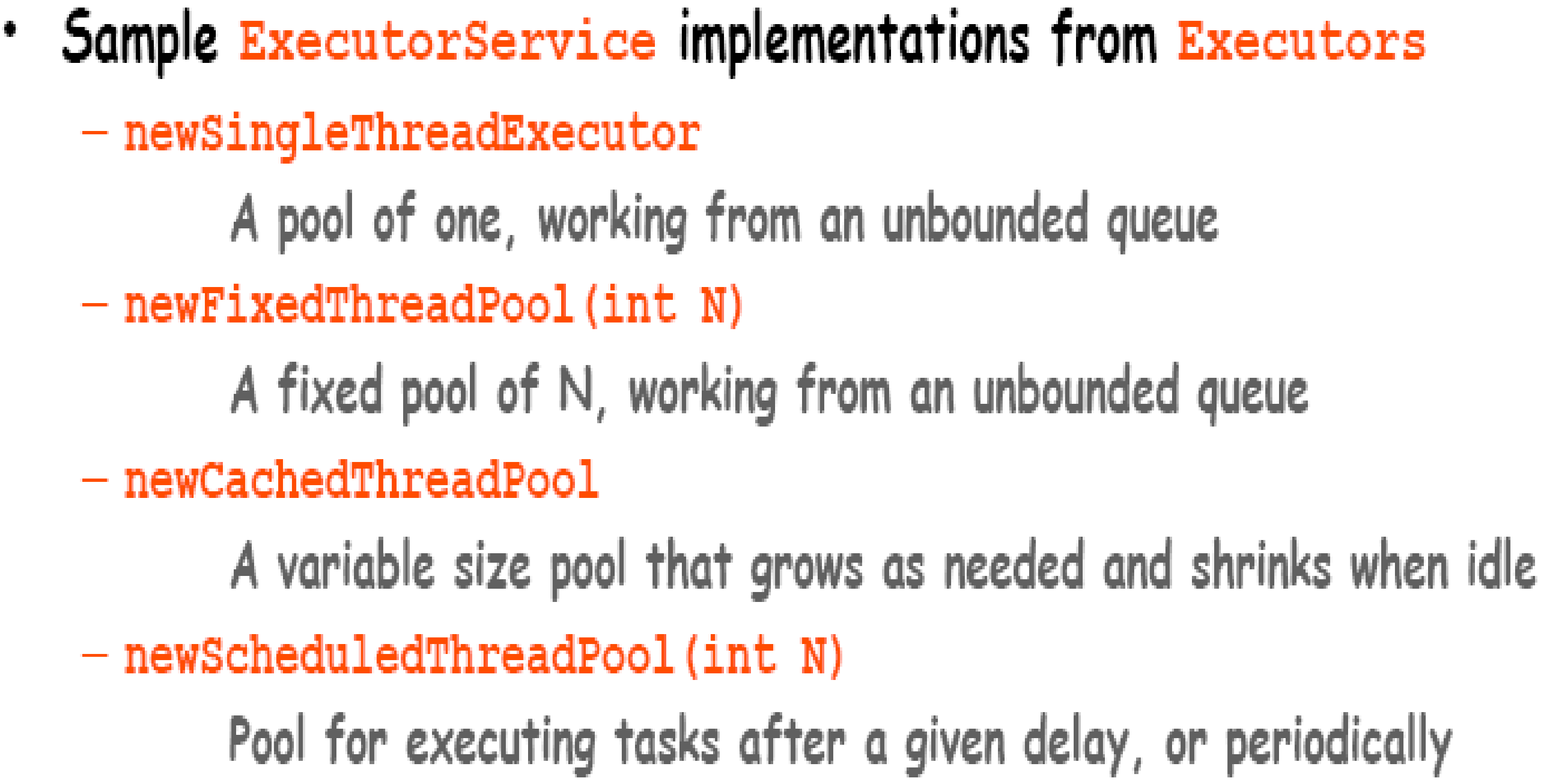
# Executors

- Framework for asynchronous task execution
- Standardize asynchronous invocation
  - Framework to execute `Runnable` and `Callable` tasks

    `Runnable: void run()`

    `Callable<V>: V call() throws Exception`
- Separate submission from execution policy
  - Use `anExecutor.execute(aRunnable)`
  - Not `new Thread(aRunnable).start()`
- Cancellation and shutdown support
- Usually created via `Executors` factory class
  - Configures flexible `ThreadPoolExecutor`
  - Customize shutdown methods, before/after hooks, saturation policies, queuing

# Executors



- Sample ExecutorService implementations from Executors
  - newSingleThreadExecutor

    A pool of one, working from an unbounded queue
  - newFixedThreadPool(int N)

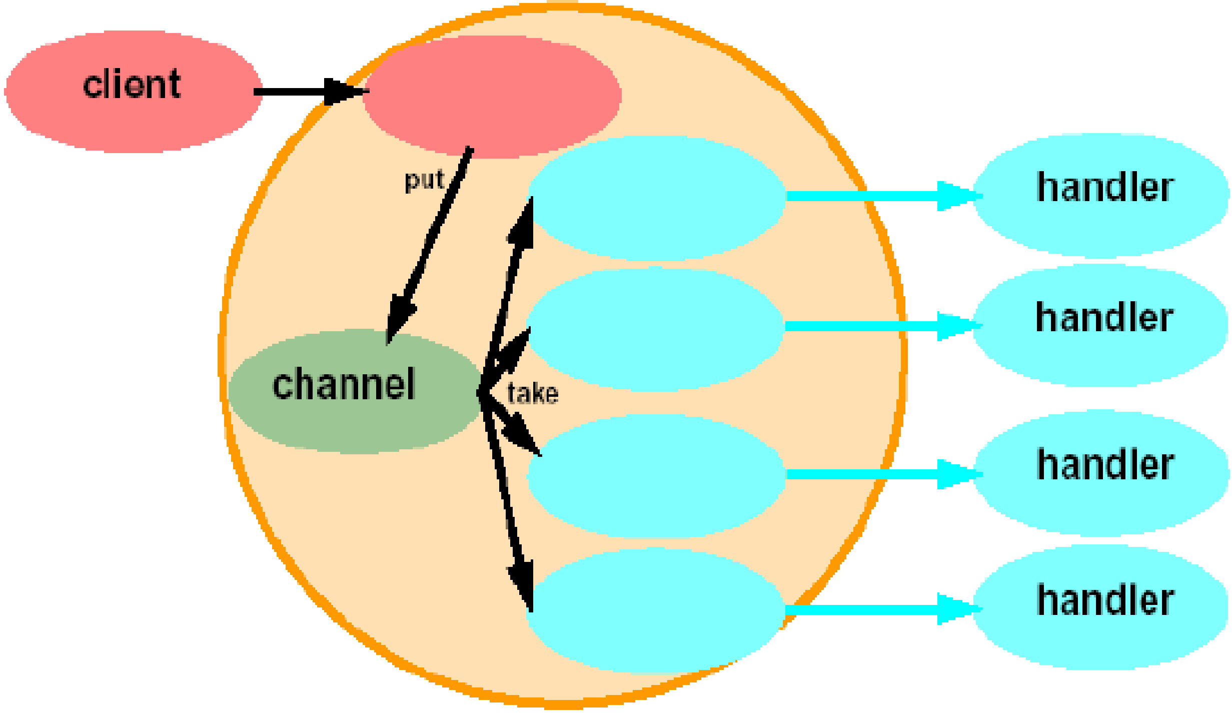
    A fixed pool of N, working from an unbounded queue
  - newCachedThreadPool

    A variable size pool that grows as needed and shrinks when idle
  - newScheduledThreadPool(int N)

    Pool for executing tasks after a given delay, or periodically

# Executors

## Thread Pools



- Use a collection of worker threads, not just one
  - Can limit maximum number and priorities of threads
  - Dynamic worker thread management
    - Sophisticated policy controls
  - Often faster than thread-per-message for I/O bound actions

# Executors

- Sophisticated `ExecutorService` implementation with numerous tuning parameters
  - Core and maximum pool size
    - Thread created on task submission until core size reached
    - Additional tasks queued until queue is full
    - Thread created if queue full until maximum size reached
    - Note: unbounded queue means the pool won't grow above core size
  - Keep-alive time
    - Threads above the core size terminate if idle for more than the keep-alive time
    - In JDK 6 core threads can also terminate if idle
  - Pre-starting of core threads, or else on demand

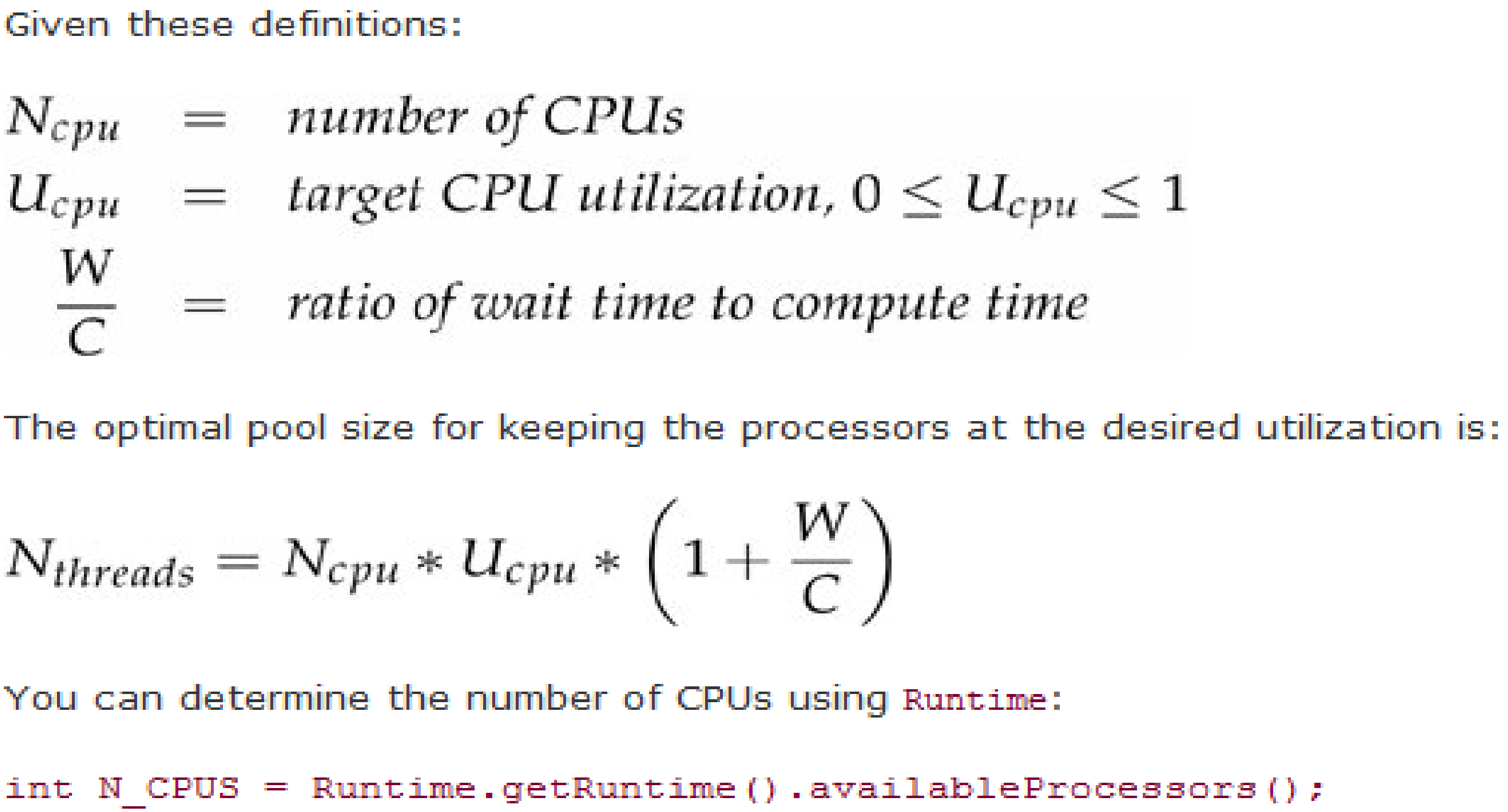
# Executors

- ThreadFactory used to create new threads
  - Default: Executors.defaultThreadFactory
- Queuing strategies: must be a BlockingQueue<Runnable>
  - Direct hand-off via SynchronousQueue: zero capacity; hands-off to waiting thread, else creates new one if allowed, else task rejected
  - Bounded queue: enforces resource constraints, when full permits pool to grow to maximum, then tasks rejected
  - Unbounded queue: potential for resource exhaustion but otherwise never rejects tasks
- Queue is used internally
  - Use remove or purge to clear out cancelled tasks
  - You should not directly place tasks in the queue

    Might work, but you need to rely on internal details
- Subclass customization hooks: beforeExecute and afterExecute

# Executors

## ❖ Sizing the pools:



Given these definitions:

$$\begin{aligned} N_{cpu} &= \textit{number of CPUs} \\ U_{cpu} &= \textit{target CPU utilization}, 0 \le U_{cpu} \le 1 \\ \frac{W}{C} &= \textit{ratio of wait time to compute time} \end{aligned}$$

The optimal pool size for keeping the processors at the desired utilization is:

$$N_{threads} = N_{cpu} * U_{cpu} * \left(1 + \frac{W}{C}\right)$$

You can determine the number of CPUs using `Runtime`:

```
int N_CPUS = Runtime.getRuntime().availableProcessors();
```

# Fork and Join

- **Hardware trends drive programming idioms**
  - The concurrency primitives that were sensible in 1995 reflected the hardware reality of the time:
  - Most commercially available systems provided no parallelism at all, and even the most expensive systems provided only limited parallelism.
  - Threads were used primarily for expressing *asynchrony*, not *concurrency*, and as a result, these mechanisms were generally adequate to the demands of the time.

# Fork and Join

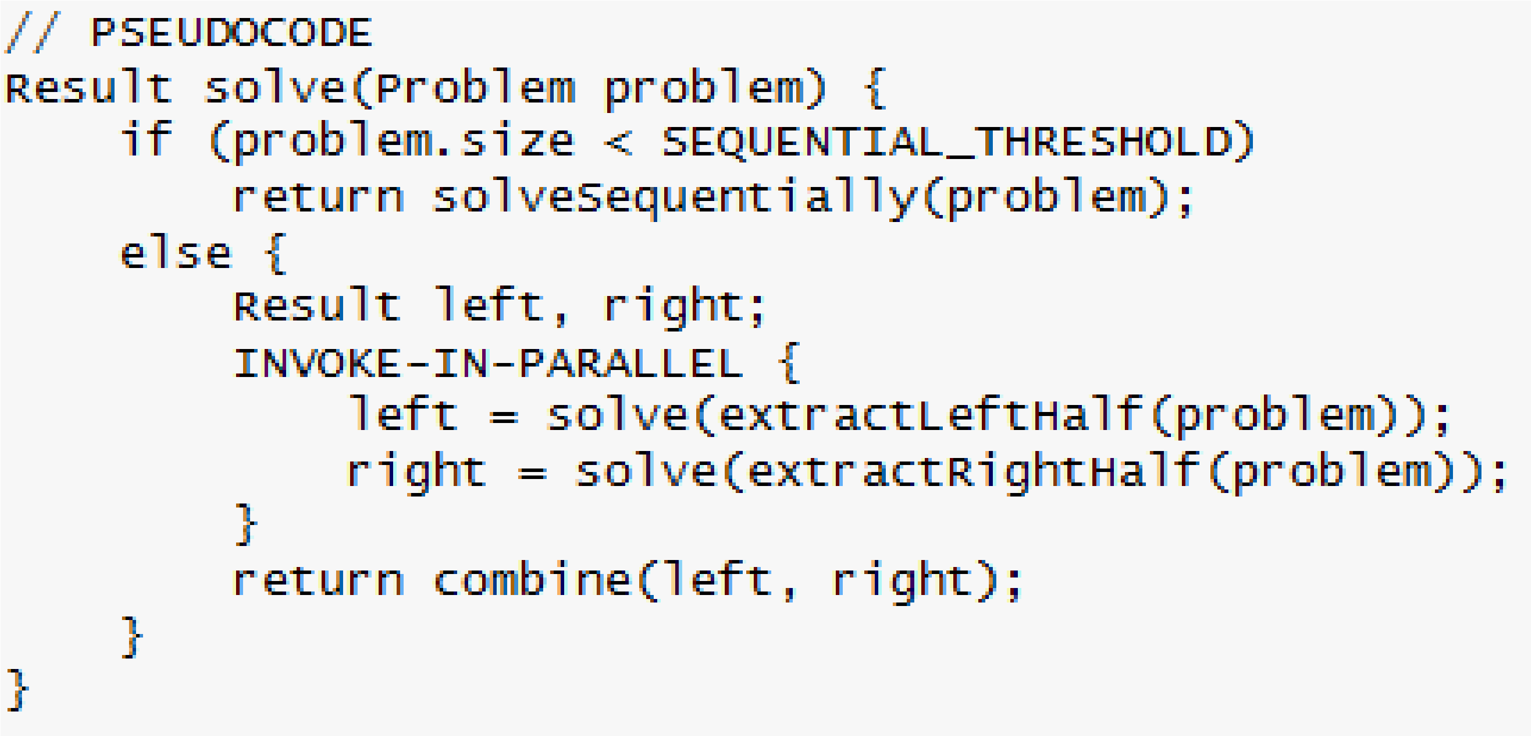
## Hardware trends drive programming idioms

- Going forward, the hardware trend is clear; **Moore Law** will not be delivering higher clock rates, but instead delivering more cores per chip.
- It is easy to imagine how you can keep a dozen processors busy using a coarse-grained task boundary such as a **user request**, but this technique will not scale to **thousands of processors**
- We will need to find finer-grained parallelism or risk keeping **processors idle even though there is plenty of work to do**

# Fork and Join

## Divide and Conquer

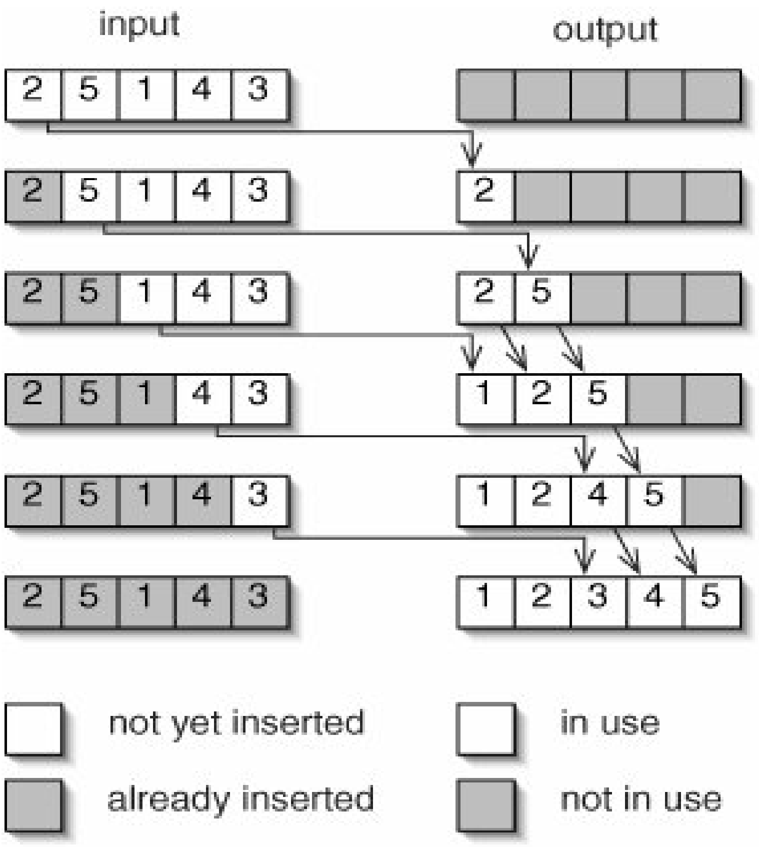
- Most sorting algorithms are recursive.

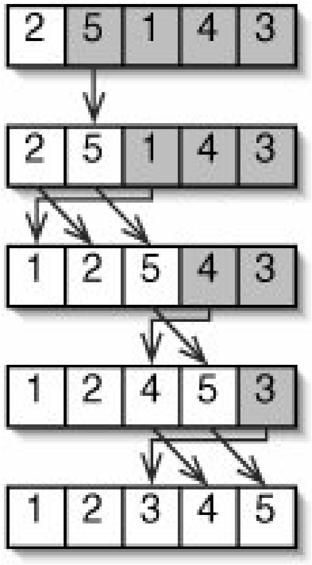


```
// PSEUDOCODE
Result solve(Problem problem) {
    if (problem.size < SEQUENTIAL_THRESHOLD)
        return solveSequentially(problem);
    else {
        Result left, right;
        INVOKE-IN-PARALLEL {
            left = solve(extractLeftHalf(problem));
            right = solve(extractRightHalf(problem));
        }
        return combine(left, right);
    }
}
```

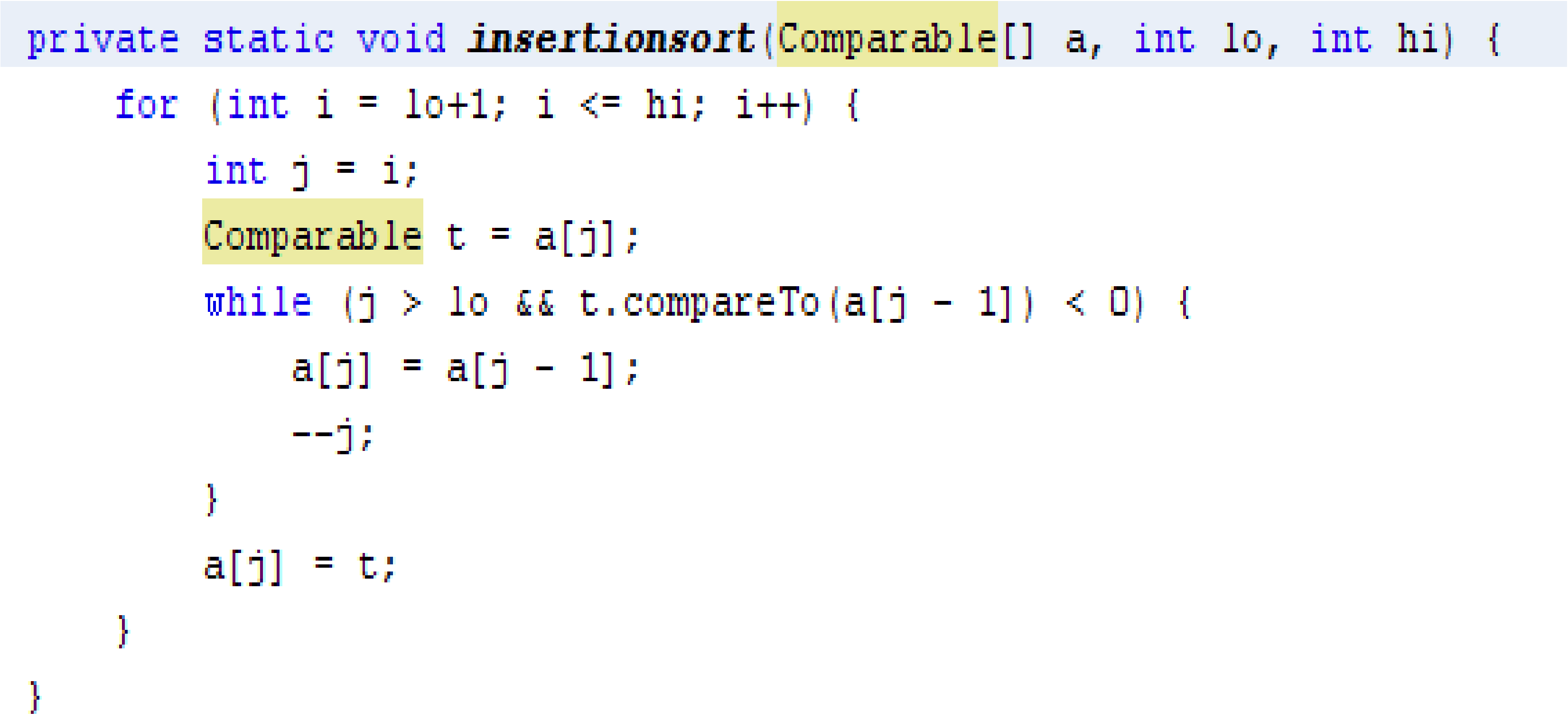
# Fork and Join

- Consider Insertion Sort





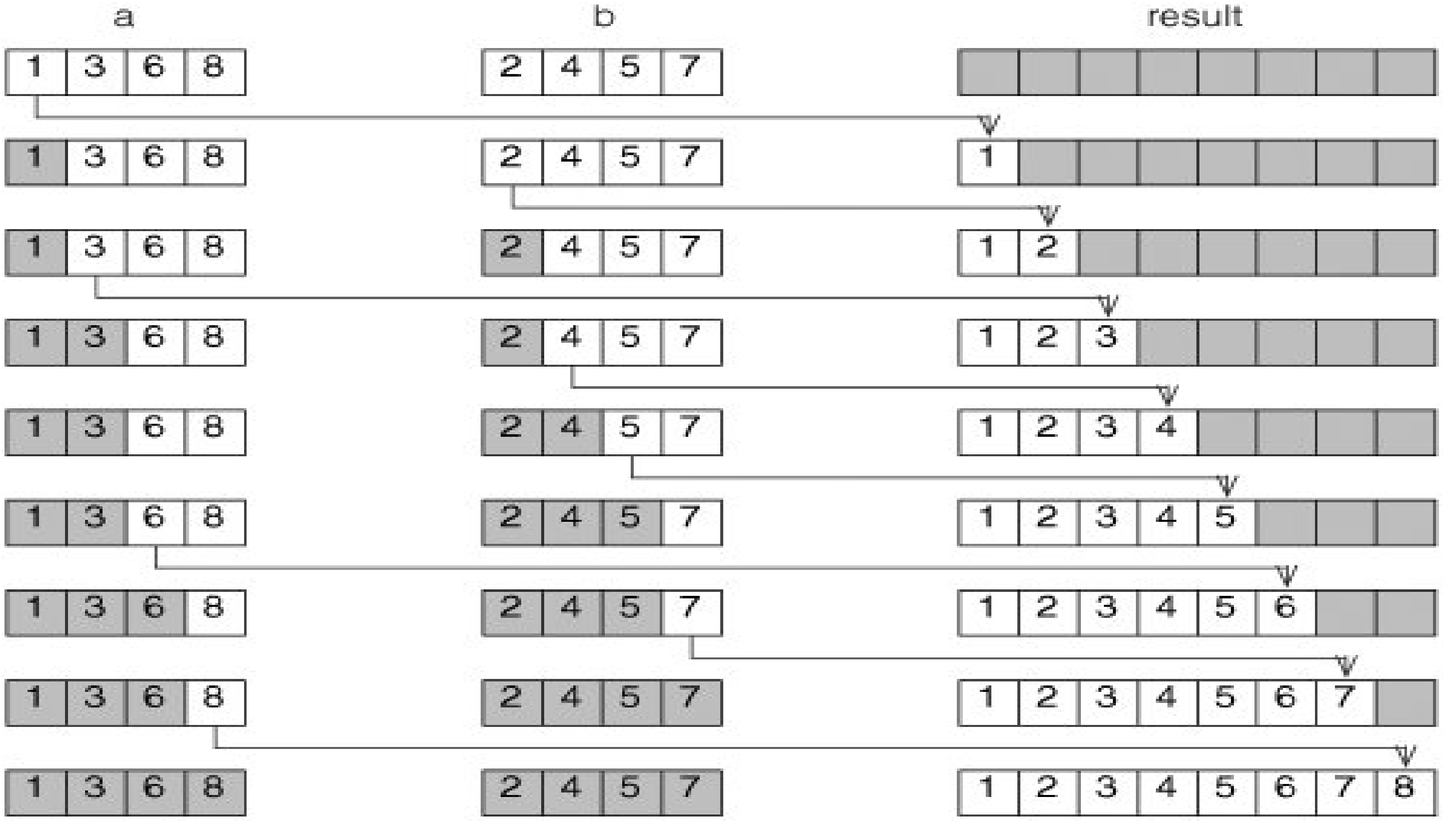
# Fork and Join



```
private static void insertionsort(Comparable[] a, int lo, int hi) {
    for (int i = lo+1; i <= hi; i++) {
        int j = i;
        Comparable t = a[j];
        while (j > lo && t.compareTo(a[j - 1]) < 0) {
            a[j] = a[j - 1];
            --j;
        }
        a[j] = t;
    }
}
```

# Fork and Join

## ❖ Recursive Merge Sort



a b result

1 3 6 8 2 4 5 7

1 3 6 8 2 4 5 7 1

1 3 6 8 2 4 5 7 1 2

1 3 6 8 2 4 5 7 1 2 3

1 3 6 8 2 4 5 7 1 2 3 4

1 3 6 8 2 4 5 7 1 2 3 4 5

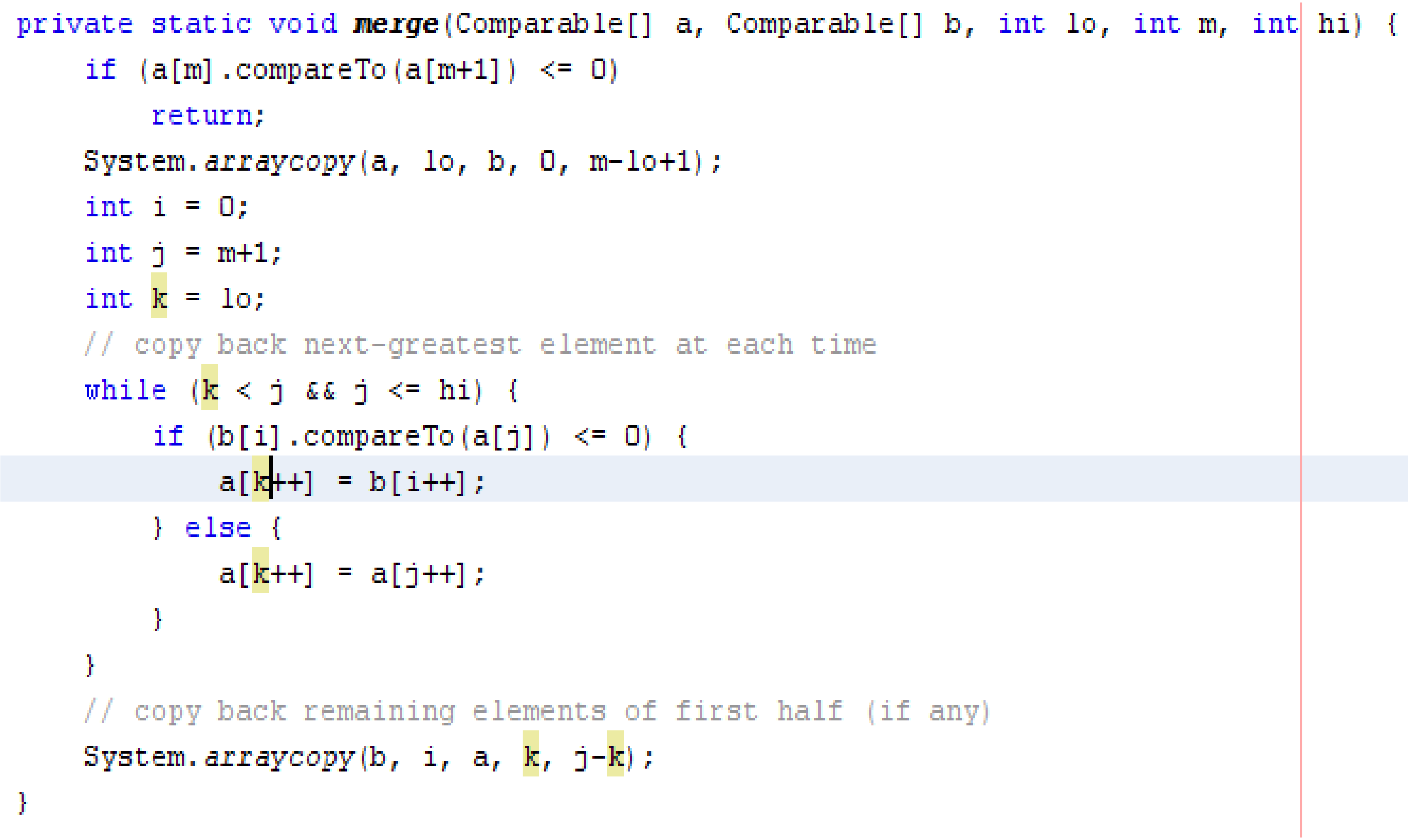
1 3 6 8 2 4 5 7 1 2 3 4 5 6

1 3 6 8 2 4 5 7 1 2 3 4 5 6 7

1 3 6 8 2 4 5 7 1 2 3 4 5 6 7 8

# Fork and Join

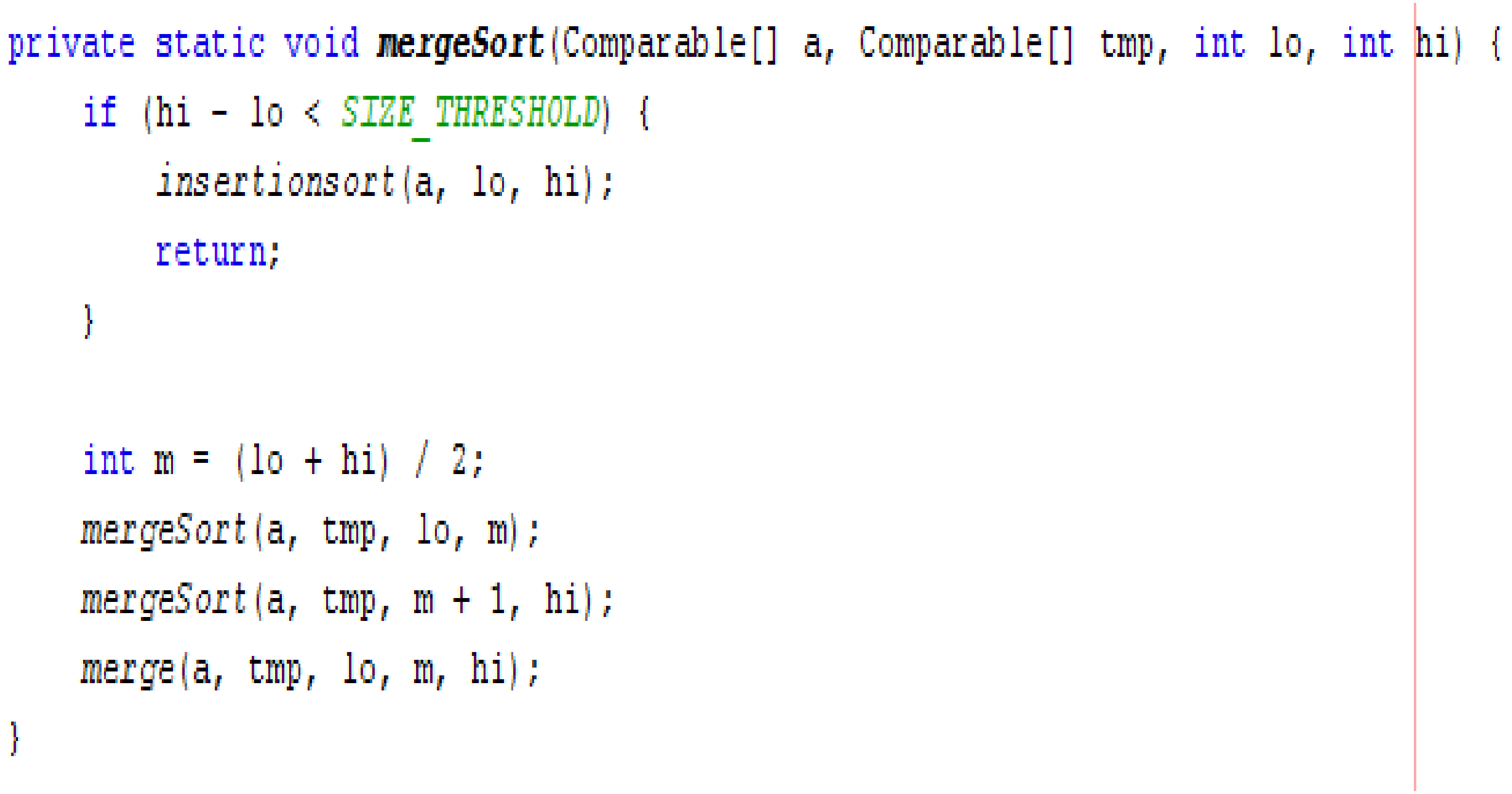
## ❖ Recursive Merge Sort



```java
private static void merge(Comparable[] a, Comparable[] b, int lo, int m, int hi) {
    if (a[m].compareTo(a[m+1]) <= 0)
        return;
    System.arraycopy(a, lo, b, 0, m-lo+1);
    int i = 0;
    int j = m+1;
    int k = lo;
    // copy back next-greatest element at each time
    while (k < j && j <= hi) {
        if (b[i].compareTo(a[j]) <= 0) {
            a[k++] = b[i++];
        } else {
            a[k++] = a[j++];
        }
    }
    // copy back remaining elements of first half (if any)
    System.arraycopy(b, i, a, k, j-k);
}
```

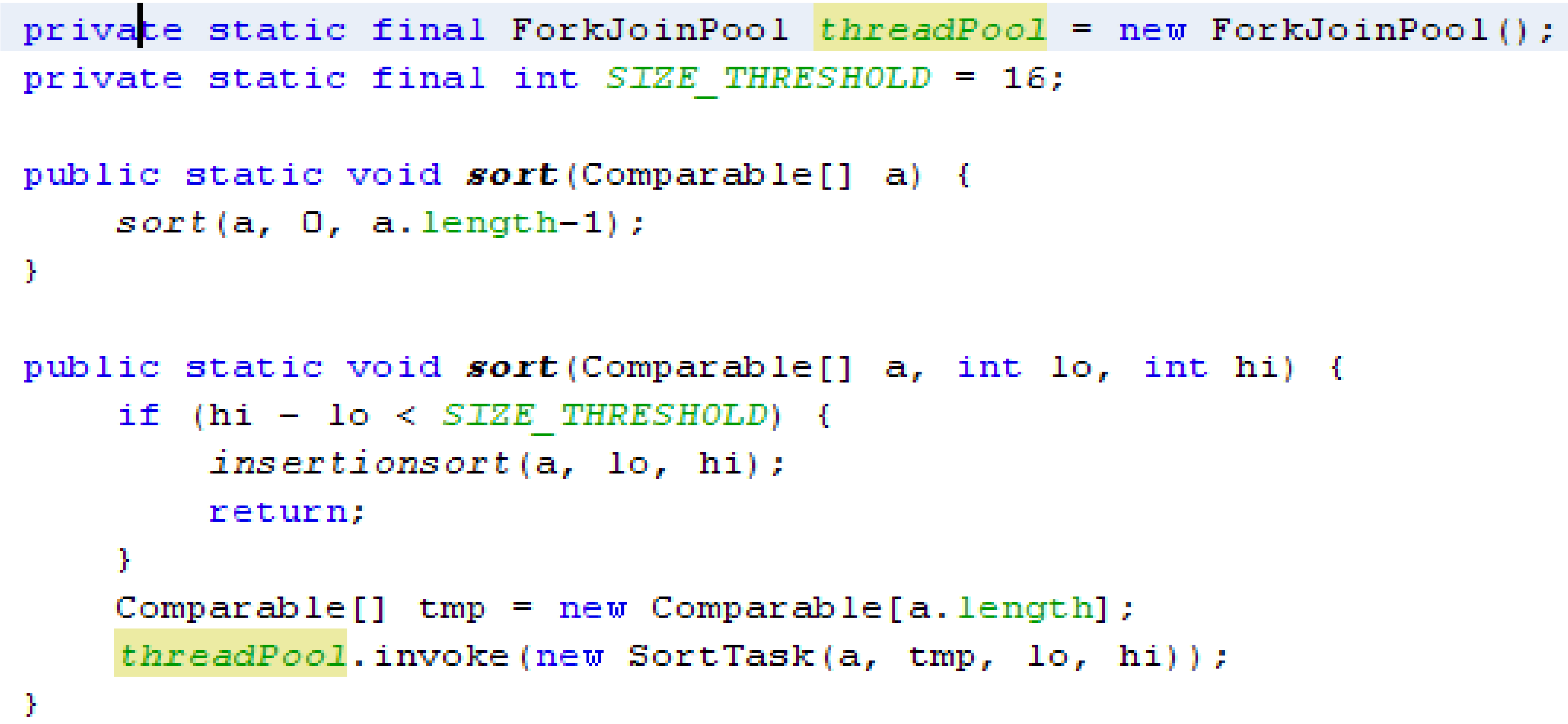
# Fork and Join

## Recursive Merge Sort



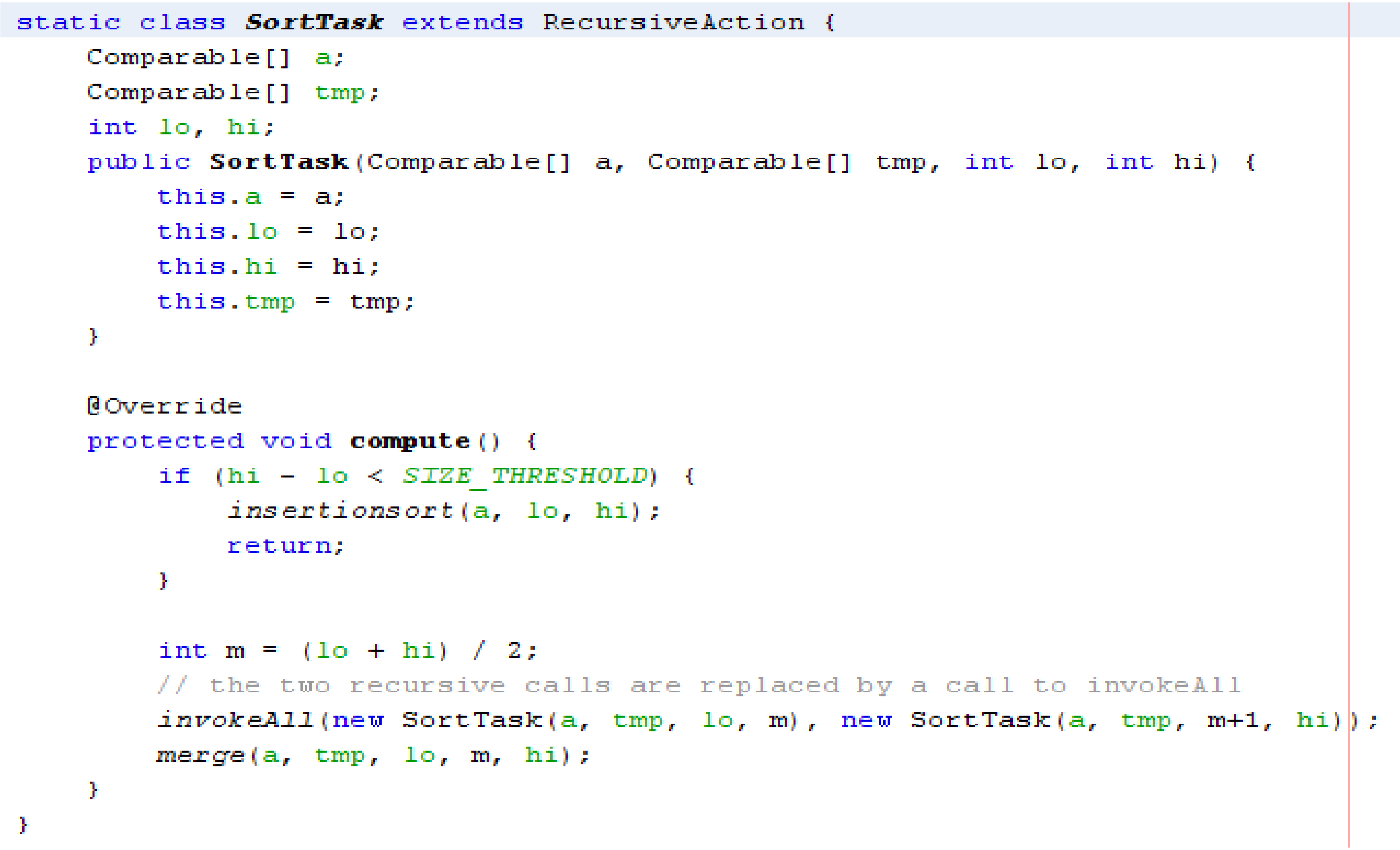
```
private static void mergeSort(Comparable[] a, Comparable[] tmp, int lo, int hi) {
    if (hi - lo < SIZE_THRESHOLD) {
        insertionsort(a, lo, hi);
        return;
    }

    int m = (lo + hi) / 2;
    mergeSort(a, tmp, lo, m);
    mergeSort(a, tmp, m + 1, hi);
    merge(a, tmp, lo, m, hi);
}
```

# Fork and Join-Parallelizing Recursion



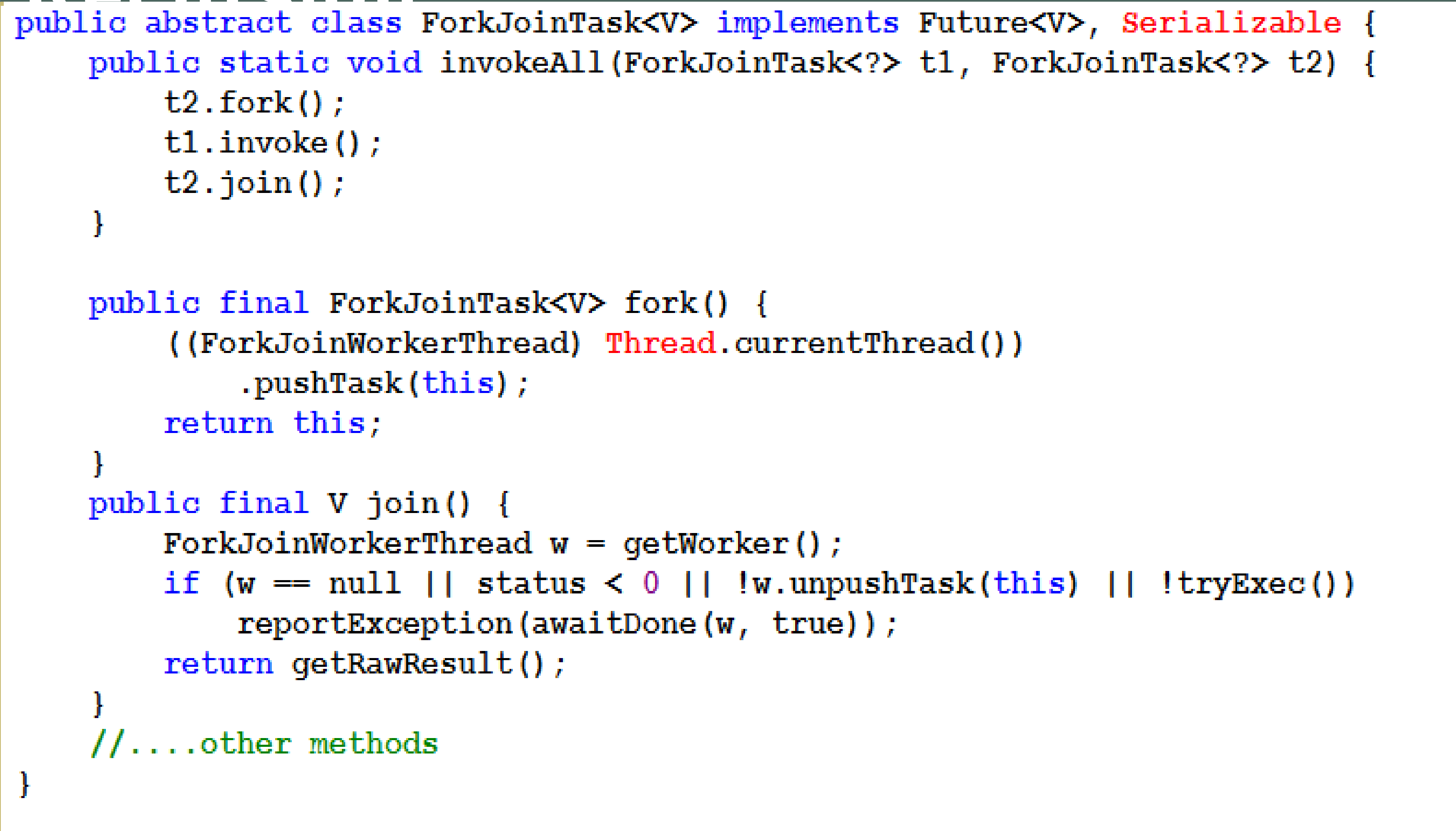
```
private static final ForkJoinPool threadPool = new ForkJoinPool();
private static final int SIZE_THRESHOLD = 16;

public static void sort(Comparable[] a) {
    sort(a, 0, a.length-1);
}

public static void sort(Comparable[] a, int lo, int hi) {
    if (hi - lo < SIZE_THRESHOLD) {
        insertionsort(a, lo, hi);
        return;
    }
    Comparable[] tmp = new Comparable[a.length];
    threadPool.invoke(new SortTask(a, tmp, lo, hi));
}
```

# Fork and Join-Parallelizing Recursion



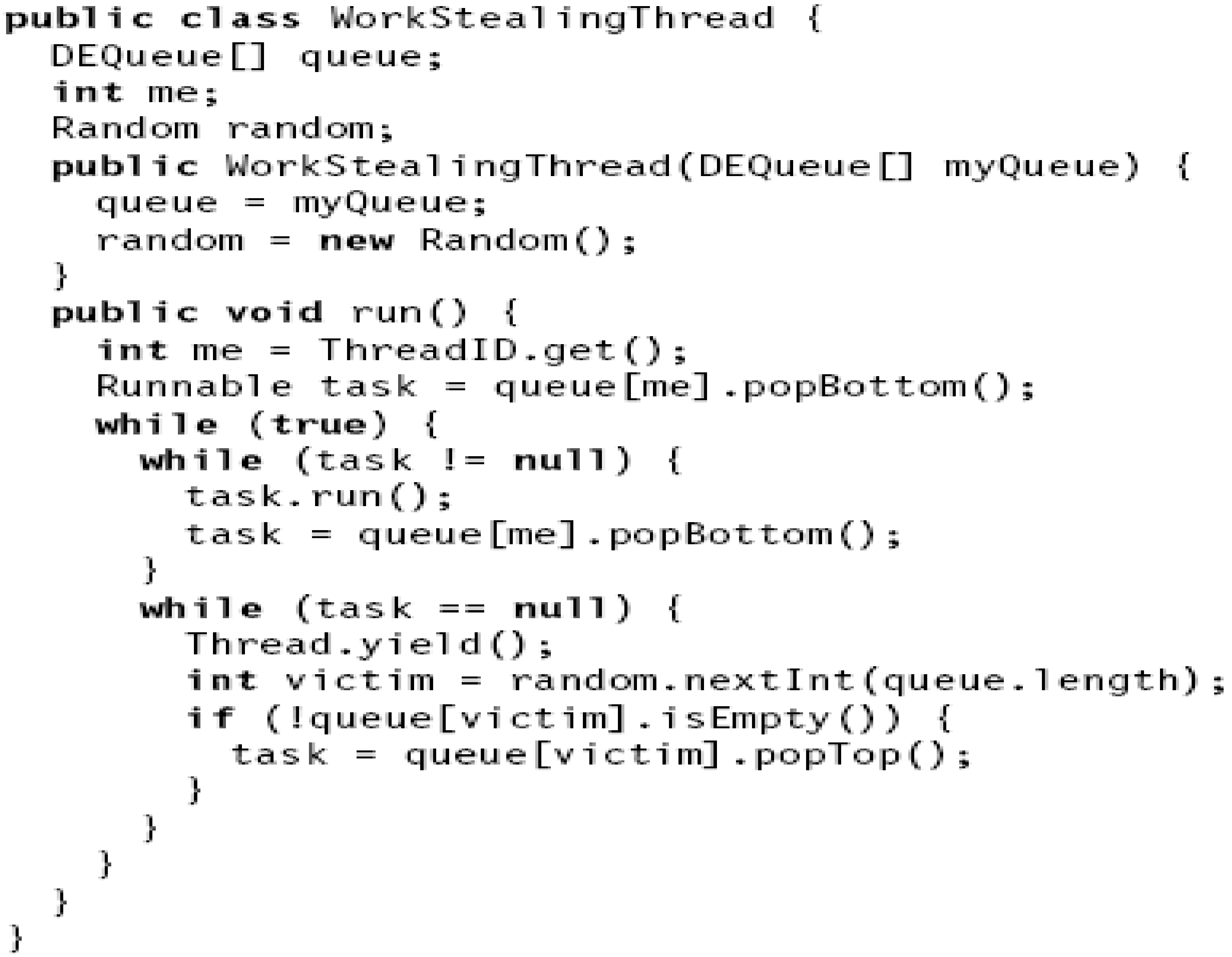
```java
static class SortTask extends RecursiveAction {
    Comparable[] a;
    Comparable[] tmp;
    int lo, hi;
    public SortTask(Comparable[] a, Comparable[] tmp, int lo, int hi) {
        this.a = a;
        this.lo = lo;
        this.hi = hi;
        this.tmp = tmp;
    }

    @Override
    protected void compute() {
        if (hi - lo < SIZE_THRESHOLD) {
            insertionsort(a, lo, hi);
            return;
        }

        int m = (lo + hi) / 2;
        // the two recursive calls are replaced by a call to invokeAll
        invokeAll(new SortTask(a, tmp, lo, m), new SortTask(a, tmp, m+1, hi));
        merge(a, tmp, lo, m, hi);
    }
}
```

# Fork and Join-Parallelizing Recursion



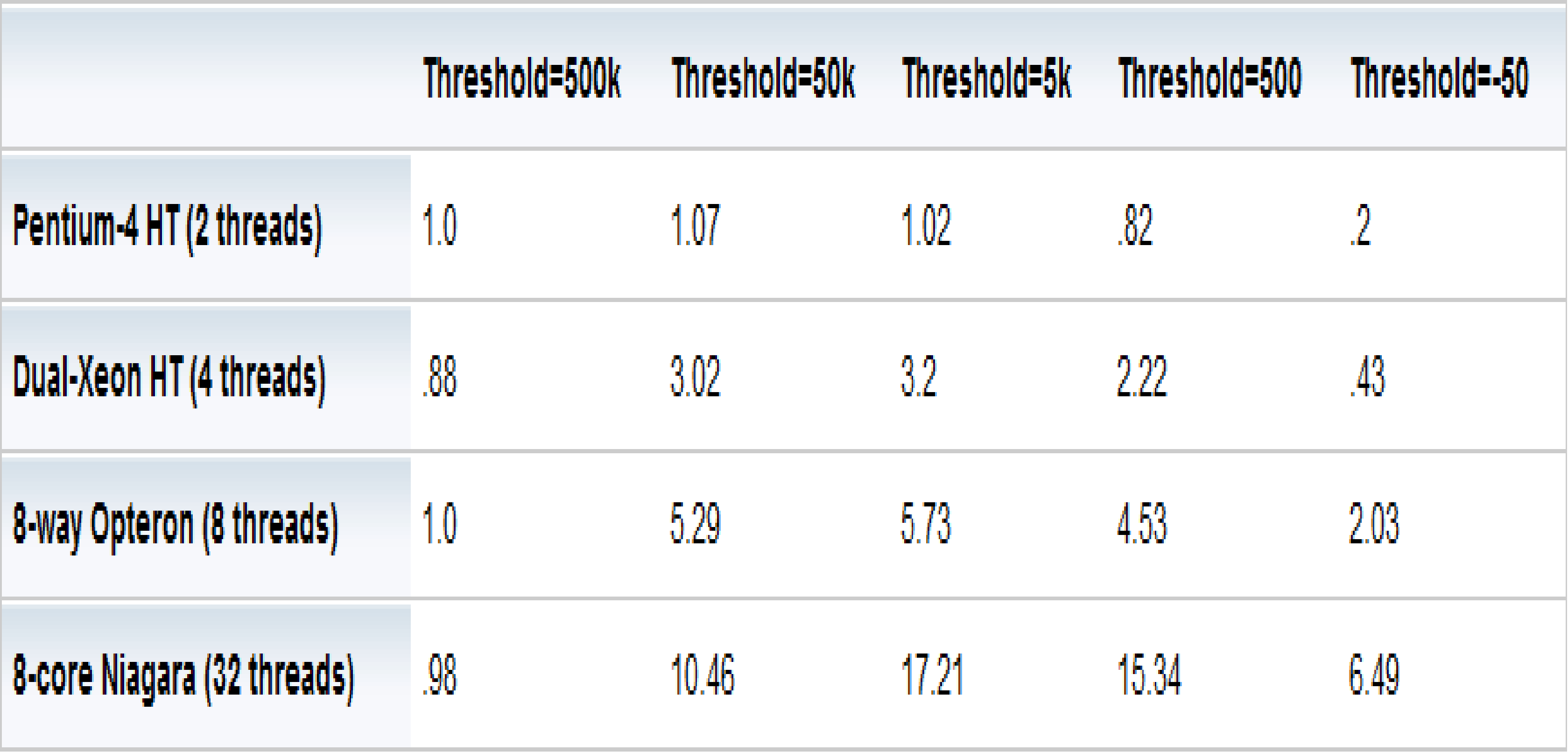
```java
public abstract class ForkJoinTask<V> implements Future<V>, Serializable {
    public static void invokeAll(ForkJoinTask<?> t1, ForkJoinTask<?> t2) {
        t2.fork();
        t1.invoke();
        t2.join();
    }

    public final ForkJoinTask<V> fork() {
        ((ForkJoinWorkerThread) Thread.currentThread())
            .pushTask(this);
        return this;
    }
    public final V join() {
        ForkJoinWorkerThread w = getWorker();
        if (w == null || status < 0 || !w.unpushTask(this) || !tryExec())
            reportException(awaitDone(w, true));
        return getRawResult();
    }
    //....other methods
}
```

# Deques used for WorkStealing



```
public class WorkStealingThread {
  DEQueue[] queue;
  int me;
  Random random;
  public WorkStealingThread(DEQueue[] myQueue) {
    queue = myQueue;
    random = new Random();
  }
  public void run() {
    int me = ThreadID.get();
    Runnable task = queue[me].popBottom();
    while (true) {
      while (task != null) {
        task.run();
        task = queue[me].popBottom();
      }
      while (task == null) {
        Thread.yield();
        int victim = random.nextInt(queue.length);
        if (!queue[victim].isEmpty()) {
          task = queue[victim].popTop();
        }
      }
    }
  }
}
```

# Deques used for WorkStealing



| | Threshold=500k | Threshold=50k | Threshold=5k | Threshold=500 | Threshold=-50 |
|---|---|---|---|---|---|
| Pentium-4 HT (2 threads) | 1.0 | 1.07 | 1.02 | .82 | .2 |
| Dual-Xeon HT (4 threads) | .88 | 3.02 | 3.2 | 2.22 | .43 |
| 8-way Opteron (8 threads) | 1.0 | 5.29 | 5.73 | 4.53 | 2.03 |
| 8-core Niagara (32 threads) | .98 | 10.46 | 17.21 | 15.34 | 6.49 |

# Thank You !

www.themegallery.com