JAVA CONCURRENCY FRAMEWORK

CSCI 5448

Spring 2011

Jay Daugherty

SUMMARY

- The Java Concurrency Framework provides a set of safe and robust services that allow Java programmers to easily create code that will be able to take advantage of concurrent programming
- This presentation will introduce the framework and show examples of how the framework can be used in multithreaded programs

ABOUT THE JAVA CONCURRENCY FRAMEWORK

- Framework was in-part developed by Doug Lea and was available for three years before integration into J2SE 5.0
- Added to Java in J2SE 5.0 as Java Specification Request 166
- Replaced the existing and limited Java support for concurrency which often required developers to create their own solutions to solve concurrency problems
- Framework also caused JVMs to be updated to properly support the new functionality
- Three main packages:
  - java.util.concurrent
  - java.util.concurrent.atomic
  - java.util.concurrent.locks

PURPOSE

- Meant to have the same effect on Java as java.util.Collections framework
- Provides Java with set of utilities that are:
  - Standardized
  - Easy to use
  - Easy to understand
  - High quality
  - High performance
  - Useful in a large set of applications with a range of expertise from beginner to expert

PRIMARY CLASSES AND SERVICES

- The main interfaces and classes in the framework are:
  - Executors
  - Thread Factory
  - Futures
  - Queues
  - Conditions
  - Synchronizers
  - Concurrent Collections
  - Atomic Variables
  - Locks

EXECUTORS

- An executor is simply an object that executes runnable tasks
- Decouples task submission from the details of how a task will executed
- Does not require task to be run asynchronously
- The framework provides two sub-interfaces and three implementations of the Executor interface:
  - ExecutorService – extends base interface to shut-down termination and support Futures
  - ScheduledExecutorService – extends ExecutorService to include delays in execution
  - AbstractExecutorService – default implementation of ExecutorService
  - ScheduledThreadPoolExecutor – extension of ThreadPoolExecutor that icludes services to delay thread execution
  - ThreadPoolExecutor – implementation with a set of threads to run submitted tasks; minimizes thread-creation overhead since this Executor uses its own set of threads instead of creating new threads to execute tasks

EXECUTOR EXAMPLE (I)

```
4  public class ExecutorExample implements Executor {
5      private static ArrayBlockingQueue<Thread> threads = new ArrayBlockingQueue<Thread>(2);
6      private static int id = 0;
7
8      public void execute(Runnable r, String name, int priority) {
9          ThreadFactoryExample tFactory = new ThreadFactoryExample();
10         Thread t = tFactory.newThread(r,name,priority);
11         if (threads.offer(t)) {
12             System.out.println("Thread "+t.getName()+" added to execution queue");
13         }
14         try {
15             t = threads.take();
16         } catch (InterruptedException e) {
17             e.printStackTrace();
18         }
19         System.out.println("Thread "+t.getName()+" started with priority "+t.getPriority());
20         t.start();
21     }
22
23     public void execute(Runnable c) {
24         this.execute(c,"Task "+id++,Thread.NORM_PRIORITY);
25     }
26
27 }
```

EXECUTOR EXAMPLE (II)

- The previous code shows the ExecutorExample class that implements the Executor interface
- The execute() method is overloaded to allow for a customized version of Executor allowing the name and priority of a thread to be controlled or set to a default value if they are not used in the execute() call
- The customized version contains a ThreadFactory to control the creation of new threads used in the executor
- It also contains an ArrayBlockingQueue used to store the threads so that if multiple runnable tasks are submitted to the ExecutorExample, they will by safely handled and run in the order submitted to the queue

THREAD FACTORY

- A ThreadFactory enable a specific type of thread to be created in a standardized way without intervention from the client
- This example shows how a thread factory can be used to standardize the creation of a custom thread
- The name and priority of a new thread are set at creation of the thread instead multi-stage process

```
3  public class ThreadFactoryExample implements ThreadFactory {
4      private static int id = 0;
5
6      public Thread newThread(Runnable r, String name, int priority) {
7          Thread t = new Thread(r);
8          if (priority>Thread.MAX_PRIORITY) {
9              priority=Thread.MAX_PRIORITY;
10         } else if (priority<Thread.MIN_PRIORITY) {
11             priority=Thread.MIN_PRIORITY;
12         }
13         t.setPriority(priority);
14         t.setName(name);
15         return t;
16     }
17
18     public Thread newThread(Runnable r) {
19         return this.newThread(r, "Task "+id++,Thread.NORM_PRIORITY);
20     }
21
22 }
```

FUTURES

- A future is an object that holds the result of an asynchronous computation
- Methods are provided to check completion status via isDone()
- The computation can be cancelled via the cancel() method if the computation has already completed
- The result is retrieved via get() which will block until the computation is complete
- Futures can be returned by Executors or used directly in code

FUTURE EXAMPLE

- 100 Callable Calculation objects are submitted to an executor with a linked list of Futures to store the result
- The results of the Callables are stored in the Future objects until get() is called
- If the callable has not returned, the get() will wait for the result to return

```
 9 public class FutureExample {
10     public void test ( ) {
11         ExecutorService executor = Executors.newFixedThreadPool(100);
12         List<Future<Integer>> values = new LinkedList<Future<Integer>>();
13         for (int i=0;i<100;i++) {
14             Callable<Integer> value = new Calculation();
15             Future<Integer> task = executor.submit(value);
16             values.add(task);
17         }
18         Float mean = (float)0;
19         for (Future<Integer> result : values) {
20             try {
21                 mean += (float)result.get()/100;
22             } catch (InterruptedException e) {
23                 e.printStackTrace();
24             } catch (ExecutionException e) {
25                 e.printStackTrace();
26             }
27         }
28         System.out.println("The mean of 100 random integers is "+mean);
29     }
30 }
```

RUNNABLE AND CALLABLE

- Meant to be used instead of sub classing a Thread
- The Runnable interface provides a way for an object to execute code when it is active
- Runnable classes do not return a result
- Callable classes return a result
- Callable can also throw an exception if it cannot calculated a result
- The following example shows a simple Callable class that returns a random integer.
- Runnable examples are in shown later examples

```
4  public class Calculation implements Callable<Integer> {
5      public Integer call() {
6          return new Random().nextInt(1000);
7      }
8  }
9
```

QUEUES

- Queues are a synchronized structures to hold tasks before being executed in the Java Concurrent Framework
- Standard queue commands like offer(), remove(), poll() and others are available
- Various forms in java.util.concurrency:
  - AbstractQueue – default implementation that provides basic queue services
  - ArrayBlockQueue – FIFO that is bounded by a fixed capacity set at construction and based on an array
  - BlockingQueue – extends Queue with services to wait for the queue to be not empty or full; good for producer/consumer
  - ConcurrenctLinkedQueue – an unbounded queue based on linked nodes
  - DelayQueue – an extension of a BlockingQueue where the head of the queue can execute if its delay has expired
  - LinkedBlockingQueue – an extension of the BlockingQueue based on linking nodes instead of an array; may be bounded or unbounded
  - PriorityBlockingQueue – extends PriorityQueue to make it thread-safe
  - SynchronousQueue – A queue with no storage, a put() must wait for a get()

SYNCHRONOUS QUEUE EXAMPLE (I)

```
3  public class SynchronousQueueExample {
4
5      public void test ( ) {
6          SynchronousQueue<Integer> sq
7              = new SynchronousQueue<Integer>();
8          Taker t = new Taker(sq);
9          Putter o = new Putter(sq);
10         for (int i = 0; i < 5; i++) {
11             new Thread(t).start();
12             new Thread(o).start();
13         }
14     }
15
16 }
```

```
1 placed in queue.
3 removed from queue.
4 placed in queue.
4 removed from queue.
2 removed from queue.
3 placed in queue.
2 placed in queue.
1 removed from queue.
5 removed from queue.
5 placed in queue.
```

```
5 placed in queue.
1 placed in queue.
5 removed from queue.
2 removed from queue.
4 placed in queue.
1 removed from queue.
3 removed from queue.
4 removed from queue.
2 placed in queue.
3 placed in queue.
```

```
3  public class Taker implements Runnable {
4      SynchronousQueue<Integer> sq;
5
6      Taker (SynchronousQueue<Integer> pbq) {
7          super();
8          this.sq=pbq;
9      }
10
11     public void run() {
12         try {
13             System.out.println(sq.take().toString()
14                     + " removed from queue.");
15         } catch (InterruptedException e) {
16             e.printStackTrace();
17         }
18     }
19 }
```

```
4  public class Putter implements Runnable {
5      SynchronousQueue<Integer> sq;
6      private static AtomicInteger value
7          = new AtomicInteger(1);
8
9      Putter (SynchronousQueue<Integer> sq) {
10         super();
11         this.sq=sq;
12     }
13
14     public void run() {
15         try {
16             Integer i = value.getAndIncrement();
17             sq.put(i);
18             System.out.println(i.toString()
19                     + " placed in queue.");
20         } catch (InterruptedException e) {
21             e.printStackTrace();
22         }
23     }
24 }
```

SYNCHRONOUS QUEUE EXAMPLE (I)

- The synchronous queue controls the access to the queue so that unless a Putter has put() a value into the queue a Taker cannot take() a value
- The synchronous queue also prevents a Putter from putting a new value onto the queue until there is a Taker to receive the value
- In other words, the queue does not have any storage capacity and acts like a traffic controller to route a synchronous transaction between two threads
- In the sample output, there are 5 Putter threads each putting a value into the queue and 5 Taker threads removing a value from the queue and each threads execution occurs at the same time so the reporting statements show up out of order for what actually happens in the queue

CONDITIONS

- Provide a framework to allow a thread to suspend safely and allow another thread to enable a condition where execution can continue
- Replaces Java Object methods wait, notify and notifyAll that implemented a monitor
- Condition is bonded to a lock
- Condition can be
  - Interruptable – condition causes thread to wait until signaled or interrupted before resuming
  - Non interruptable – condition causes thread to wait until signaled before resuming
  - Timed – condition causes thread to wait a set amount of time (or until signaled/interrupted) before trying to resume

CONDITION EXAMPLE (I)

```
5  public class ConditionLockExample {
6      private Lock lock = new ReentrantLock();
7      private Condition zero = lock.newCondition();
8      private Condition max = lock.newCondition();
9      private int value = 0;
10     private final int MAX = 5;
11     private final int MIN = 0;
12
13     public void increment ( ) {
14         lock.lock();
15         try {
16             while (value >= MAX) {
17                 zero.await();
18             }
19             value++;
20             max.signal();
21             System.out.println("Value incremented to " + value);
22         } catch (InterruptedException e) {
23             e.printStackTrace();
24         } finally {
25             lock.unlock();
26         }
27     }
28
29     public void decrement ( ) {
30         lock.lock();
31         try {
32             while (value <= MIN) {
33                 max.await();
34             }
35             value--;
36             zero.signal();
37             System.out.println("Value decremented to " + value);
38         } catch (InterruptedException e) {
39             e.printStackTrace();
40         } finally {
41             lock.unlock();
42         }
43     }
44 }
```

```
Value incremented to 1
Value incremented to 2
Value decremented to 1
Value incremented to 2
Value decremented to 1
Value decremented to 0
```

```
Value incremented to 1
Value decremented to 0
Value incremented to 1
Value incremented to 2
Value decremented to 1
Value decremented to 0
```

```
2  public class CLEIncrementor implements Runnable {
3      ConditionLockExample conditionLock;
4      public CLEIncrementor (ConditionLockExample conditionLock) {
5          super();
6          this.conditionLock = conditionLock;
7      }
8
9      public void run() {
10         conditionLock.increment();
11     }
12 }
```

```
2  public class CLIEDecrementor implements Runnable {
3      ConditionLockExample conditionLock;
4      public CLIEDecrementor (ConditionLockExample conditionLock) {
5          super();
6          this.conditionLock = conditionLock;
7      }
8
9      public void run() {
10         conditionLock.decrement();
11     }
12
13 }
```

CONDITION EXAMPLE (II)

- The previous condition example implements a monitor of sorts
- The condition prevents a value from being decremented below 0 and incremented above 5
- If the threads with the conditional locks try to violate the contract of the lock, they are told to wait
- When the conditions of the lock change to allow further execution the threads are resumed and can then execute as normal

SYNCHRONIZERS

- Semaphore – provides a way to limit access to a shared resource and can control the access to n resource
  - Used with acquire and release methods in Java
  - Java also supports fairness (or not) so that the order of an acquire request is honored by the semaphore (FIFO)
- Mutex – similar to a binary semaphore
  - Implemented as Locks in Java
- Barrier – good for controlling the execution flow of a group of threads that need to synchronize at various points before continuing executing
  - await is the main method in a barrier which causes the threads to wait until all of the threads in a barrier have called await before being released
  - The constructor is called with the number of threads the barrier is managing

BARRIER EXAMPLE (I)

```
3  public class BarrierExample {
4      public void test( ) {
5          int id = 1;
6          final CyclicBarrier cb = new CyclicBarrier(6,
7              new Runnable() {
8                  public void run() {
9                      System.out.println("Barrier released.");
10                 }
11             } );
12
13         for (int i = 0; i < 6; i++) {
14             new Thread(new BarrierTask(cb, id++)).start();
15         }
16     }
17 }
```

```
4  public class BarrierTask implements Runnable {
5      final CyclicBarrier barrier;
6      final int id;
7
8      BarrierTask(CyclicBarrier barrier, int id) {
9          super();
10         this.barrier=barrier;
11         this.id=id;
12     }
13
14     public void run() {
15         try {
16             System.out.println("Task "+id+" waiting at barrier.");
17             barrier.await();
18             System.out.println("Task "+id+" released by barrier.");
19         } catch (InterruptedException e) {
20             e.printStackTrace();
21         } catch (BrokenBarrierException e) {
22             e.printStackTrace();
23         }
24     }
25
26 }
```

BARRIER EXAMPLE (II)

- Six runnable tasks delayed at a barrier and then released
- The order that the tasks reach the barrier do not matter
- Ensures that execution will by synchronized at the barrier checkpoint before further execution is performed

```
Task 1 waiting at barrier.
Task 6 waiting at barrier.
Task 5 waiting at barrier.
Task 4 waiting at barrier.
Task 3 waiting at barrier.
Task 2 waiting at barrier.
Barrier released.
Task 2 released by barrier.
Task 1 released by barrier.
Task 6 released by barrier.
Task 4 released by barrier.
Task 3 released by barrier.
Task 5 released by barrier.
```

```
Task 1 waiting at barrier.
Task 2 waiting at barrier.
Task 3 waiting at barrier.
Task 4 waiting at barrier.
Task 5 waiting at barrier.
Task 6 waiting at barrier.
Barrier released.
Task 6 released by barrier.
Task 1 released by barrier.
Task 3 released by barrier.
Task 5 released by barrier.
Task 2 released by barrier.
Task 4 released by barrier.
```

ATOMIC VARIABLES

- Atomic variables ensure that access to the variable happens as a single instruction, preventing more than one thread from accessing the value at the same time
- java.util.concurrent.atomic implements a number of variables to enable atomic execution without using an outside lock while still being thread-safe
  - boolean, int, arrays, etc.
- Extends the existing volatile Java behavior
- Basic set of atomic methods
  - get()
  - set()
  - compareAndSet(<type> expect, <type> update) – compares the current value to expect and if equal, sets the value to update

ATOMIC VARIABLE EXAMPLE (I)

- Sample code has two integers
  - Java Concurrent Framework AtomicInteger object
  - Regular Java Integer object
- The code sample creates five threads for each integer that increment the value of the integer and print the value
- The code illustrates what looks like an atomic action in Java, ++regularInteger, does not behave atomically since there are several steps to the instruction:
  - Value is retrieved
  - Value is incremented
  - Value is saved

ATOMIC VARIABLE EXAMPLE (II)

```
1  import java.util.concurrent.atomic.AtomicInteger;
2
3  public class AtomicVarExample {
4      private AtomicInteger atomicInteger = new AtomicInteger(0);
5      private Integer regularInteger = new Integer(0);
6
7      private class AtomicVarThread implements Runnable {
8          public void run ( ) {
9              System.out.println("Atomic Integer Value: " + atomicInteger.incrementAndGet());
10         }
11
12     }
13
14     private class RegularVarThread implements Runnable {
15         public void run ( ) {
16             System.out.println("Regular Integer Value: " + (++regularInteger).toString());
17         }
18     }
19
20     public void testAtomicVar (  ) {
21         AtomicVarThread avt = new AtomicVarThread();
22         RegularVarThread navt =new RegularVarThread();
23
24         for (int i = 0; i < 5; i++) {
25             new Thread(avt).start();
26             new Thread(navt).start();
27         }
28     }
29 }
```

ATOMIC VARIABLE EXAMPLE (III)

- Here are three sample outputs
- In all of the output, the AtomicInteger correctly increments a value from 1 to 5
- In the first example, the regular Integer value is not properly updated by the threads and the value is incremented to 2 instead of 5
- In the second example, the regular Integer is updated a little better, but the final thread has a stale copy of the value that it acts upon and looks like it actually decrements the value
- In all of the regular Integer examples a subset of the five threads grab the initial value at the same time and do not work in concert to atomically increment the value

```
Atomic Integer Value: 1
Atomic Integer Value: 2
Atomic Integer Value: 3
Atomic Integer Value: 4
Atomic Integer Value: 5
Regular Integer Value: 1
Regular Integer Value: 2
Regular Integer Value: 1
Regular Integer Value: 1
Regular Integer Value: 1
```

```
Atomic Integer Value: 1
Atomic Integer Value: 2
Atomic Integer Value: 3
Regular Integer Value: 1
Regular Integer Value: 1
Regular Integer Value: 2
Atomic Integer Value: 4
Atomic Integer Value: 5
Regular Integer Value: 3
Regular Integer Value: 2
```

```
Atomic Integer Value: 1
Atomic Integer Value: 2
Atomic Integer Value: 3
Regular Integer Value: 1
Regular Integer Value: 1
Regular Integer Value: 2
Atomic Integer Value: 4
Atomic Integer Value: 5
Regular Integer Value: 3
Regular Integer Value: 4
```

CONCURRENTMAP (I)

- ConcurrentMap and ConcurrentNavigableMap provide thread-safe interfaces to Map and NavigableMap
- ConcurrentHashMap implements ConcurrentMap to provide a thread-safe hash map
- ConcurrentSkipListMap implements ConcurrentNavigableMap to provide a thread-safe skip list
- The following example shows a ConcurrentHashMap used between three threads to setup a simple department store with items going in an out of stock

CONCURRENTHASHMAP EXAMPLE (II)

```
3  public class HashMapExample {
4      public void test ( ) {
5          ConcurrentHashMap<String,String> conncurrentMap = new ConcurrentHashMap<String,String>();
6          ExecutorExample ee = new ExecutorExample();
7          ee.execute(new HashMapStock(conncurrentMap), "Stock", Thread.MAX_PRIORITY);
8          ee.execute(new HashMapEmpty(conncurrentMap), "Empty", Thread.NORM_PRIORITY);
9          ee.execute(new HashMapReplace(conncurrentMap), "Replace", Thread.MIN_PRIORITY);
10     }
11 }
```

Executor

Replacement Thread

```
6  public class HashMapReplace implements Runnable {
7      ConcurrentHashMap<String,String> cMap = new ConcurrentHashMap<String,String>();
8
9      HashMapReplace (ConcurrentHashMap<String,String> cMap) {
10         super();
11         this.cMap = cMap;
12     }
13
14     public void run() {
15         Iterator<String> i = cMap.keySet().iterator();
16         while (i.hasNext()) {
17             String s = i.next();
18             if (cMap.replace(s, "sold out", "in stock")) {
19                 System.out.println(s+" back in stock");
20             }
21             try {
22                 Thread.sleep(new Random().nextInt(500));
23             } catch (InterruptedException e) {
24                 e.printStackTrace();
25             }
26         }
27     }
28
29 }
```

Sample Output

```
Thread Stock added to execution queue
Thread Stock started with priority 10
Socks in stock
Shirts in stock
Pants in stock
Shoes in stock
Thread Empty added to execution queue
Thread Empty started with priority 5
Thread Replace added to execution queue
Thread Replace started with priority 1
Socks sold out
Socks back in stock
Shirts sold out
Shoes sold out
Pants sold out
Shirts back in stock
Shoes back in stock
Pants back in stock
```

CONCURRENTHASHMAP EXAMPLE (II)

Original Stock Thread

```
3  public class HashMapStock implements Runnable {
4      ConcurrentHashMap<String,String> cMap = new ConcurrentHashMap<String,String>();
5
6      HashMapStock (ConcurrentHashMap<String,String> cMap) {
7          super();
8          this.cMap = cMap;
9      }
10
11     public void run ( ) {
12         cMap.putIfAbsent("Socks", "in stock");
13         System.out.println("Socks in stock");
14         cMap.putIfAbsent("Shirts", "in stock");
15         System.out.println("Shirts in stock");
16         cMap.putIfAbsent("Pants", "in stock");
17         System.out.println("Pants in stock");
18         cMap.putIfAbsent("Shoes", "in stock");
19         System.out.println("Shoes in stock");
20     }
21
22 }
```

Sample Output

```
Thread Stock added to execution queue
Thread Stock started with priority 10
Socks in stock
Shirts in stock
Pants in stock
Shoes in stock
Thread Empty added to execution queue
Thread Empty started with priority 5
Thread Replace added to execution queue
Thread Replace started with priority 1
Socks sold out
Socks back in stock
Shirts sold out
Shoes sold out
Pants sold out
```

Emptying Thread

```
5  public class HashMapEmpty implements Runnable {
6      ConcurrentHashMap<String,String> cMap = new ConcurrentHashMap<String,String>();
7
8      HashMapEmpty (ConcurrentHashMap<String,String> cMap) {
9          super();
10         this.cMap = cMap;
11     }
12
13     public void run ( ) {
14         Iterator<String> i = cMap.keySet().iterator();
15         while (i.hasNext()) {
16             String s = i.next();
17             if (cMap.replace(s, "in stock", "sold out")) {
18                 System.out.println(s+" sold out");
19             }
20             try {
21                 Thread.sleep(new Random().nextInt(500));
22             } catch (InterruptedException e) {
23                 e.printStackTrace();
24             }
25         }
26     }
27
28 }
```

LOCKS

- A lock controls access to a shared resource
- Typically access control is limited to one thread at a time
- A more flexible option over Java's built in synchronization and monitors
- With more flexibility, comes more complexity and care to create thread-safe code
- Different types of locks
  - Reentrant
  - Read/Write – allows multiple threads to read a resource, but only one to write the resource. A read cannot happen at the same time as a write though.
- Also known as a mutex

LOCK EXAMPLE (I)

```
3  public class RWLockExample {
4      private int value = 0;
5      private ReentrantReadWriteLock lock
6              = new ReentrantReadWriteLock();
7
8      public int read ( ) {
9          try {
10             lock.readLock().lock();
11             System.out.println("Read Lock: "
12                     + lock.getReadLockCount());
13             return value;
14         } finally {
15             System.out.println("Read lock released.");
16             lock.readLock().unlock();
17         }
18     }
19
20     public int writeNewValue (int newValue) {
21         try {
22             lock.writeLock().lock();
23             System.out.println("Write Lock: "
24                     + lock.getWriteHoldCount());
25             value = newValue;
26             return value;
27         } finally {
28             System.out.println("Write lock released.");
29             lock.writeLock().unlock();
30         }
31     }
32 }
```

```
3  public class RWLockWriter implements Runnable {
4      private RWLockExample rwle;
5
6      RWLockWriter (RWLockExample rwle) {
7          super();
8          this.rwle = rwle;
9      }
10
11     public void run() {
12         System.out.println("Value wrote is "
13                 + rwle.writeNewValue(
14                 new Random().nextInt(100)));
15     }
16 }
```

```
2  public class RWLockReader implements Runnable {
3      private RWLockExample rwle;
4
5      RWLockReader (RWLockExample rwle) {
6          super();
7          this.rwle = rwle;
8      }
9
10     public void run() {
11         System.out.println("Value read is "
12                 + rwle.read());
13     }
14
15 }
```

LOCK EXAMPLE (II)

- 3 readers and 3 writers vie for access to an single integer value controlled by the Read/Write Lock RWLockExample
- Multiple readers can have access at the same time, but not when a write is being performed
- Only one writer can have access as long a no other thread is in the lock
- Writers use the atomic writeNewValue method to change the value controlled by the lock
- Sample output shows cases with multiple reads and atomic writes in different orders
- The number of readers or writers at any time are displayed

```
Read Lock: 2
Read Lock: 2
Read lock released.
Read lock released.
Value read is 0
Value read is 0
Write Lock: 1
Write lock released.
Value wrote is 13
Write Lock: 1
Write lock released.
Value wrote is 6
Read Lock: 1
Read lock released.
Value read is 6
Write Lock: 1
Write lock released.
Value wrote is 37
```

```
Read Lock: 1
Read lock released.
Value read is 0
Write Lock: 1
Write lock released.
Value wrote is 65
Read Lock: 1
Read Lock: 2
Read lock released.
Value read is 65
Read lock released.
Value read is 65
Write Lock: 1
Write lock released.
Value wrote is 79
Write Lock: 1
Write lock released.
Value wrote is 28
```

REFERENCES

- **Concurrent Programming with J2SE 5.0, Qusay H. Mahmoud (http://java.sun.com/developer/technicalArticles/J2SE/concurrency/)**
- **Let's Resync: What's New for Concurrency on the Java Platform, Standard (http://www.oracle.com/technetwork/java/j1sessn-jsp-156525.html)**
- **Java Concurrent Animated (http://javaconcurrenta.sourceforge.net/)**
- **JavaTM 2 Platform Standard Ed. 5.0 (http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/package-summary.html)**
- **Becoming a Better Programmer: A Conversation With Java Champion Heinz Kabutz, Janice J. Heiss, (http://www.oracle.com/technetwork/articles/javase/kabutz-qa-136652.html)**
- **Java Concurrency Series, Baptiste Wicht, (http://www.baptiste-wicht.com)**
- **Doug Lea's Concurrency JSR-166 Interest Site (http://gee.cs.oswego.edu/dl/concurrency-interest/index.html)**
- **Java Concurrency Guidelines, Long, Mohindra, Seacord and Svoboda, May 2010, Carnegie Mellon**
- **Java™ Concurrency Utilities in Practice (http://www.oopsla.org/oopsla2007/index.php?page=sub/&id=69)**