Advanced Concurrent Programming

Presentation Topics

In this presentation we will cover:

- Introduction to Concurrent Libraries
- Working with Synchronizers
- Using the Execution Framework

Objectives

When we are done, you should be able to:

- Identify two motivations for the concurrent libraries
- List key components of the execution framework
- Describe one synchronizer

Introduction to Concurrent Libraries

Who in the world is Doug Lea?

Introduction to Concurrent Libraries

- Java provides built-in basic structures for concurrent programming
- Beyond "basic" concurrent solutions, built-in facilities are limited; foundational but not complete
- As a result, community created own concurrency oriented libraries to address complex situations

Java Thread Model Limitations

Based on block-structured locking

- Locks associated with entire objects
  - Can't notify specific thread based on condition
  - Have to notify unknown waiting thread
- No way to:
  - "take back" or timeout attempt to acquire lock
  - Modify lock semantics
- No "built-in":
  - Pooling mechanism
  - Auto-blocking lists
  - Limited atomic operation support

Overview of Concurrent Libraries

- Introduced as part of Java SE 5.0
- Driven by JSR 166
  - Adaptation of Doug Lea's ***`util.concurrent`*** package
  - Defined in three packages:
    - ***`java.util.concurrent`***
    - ***`java.util.concurrent.atomic`***
    - ***`java.util.concurrent.locks`***

Motivations for Concurrent Libraries

- Address limitations of Java's thread model
- Standardize and simplify common concurrency mechanisms
  - Lower complexity in development concurrent programs
  - Increase maintainability of concurrent code
  - Lessen common "concurrency" issues
- Provide robust, efficient, and high-performance utilities

java.util.concurrent Package

- Main concurrency package
- Contains classes to aid in concurrency development
- Three main facilities:
  - Concurrent collections
  - Execution framework
  - Synchronizers

Concurrent Collections

Concurrent Collections

- Extend Collections framework into concurrency world
  - More scalable than standard `Collections` classes
  - Compliant with `Collection` framework
- Provide thread safety
  - More “lightweight” than synchronized
  - Typically synchronize on manipulation
  - Typically retrieval is not synchronized
- Contains:
  - List
  - Map
  - Set
  - Queue

Concurrent Collections : Iterators

- Standard ***`java.util.Iterator`***
  - Fail-fast implementation
  - If underlying collection changes, ***`Iterator`*** throws ***`ConcurrentModificationException`***
- Concurrency Collections
  - Weakly-consistent implementation
  - Support concurrent modifications
  - May reflect underlying changes while iterating

Concurrent Collections : Lists

- Add concurrency support to lists
  - Alternative to ***Collections.synchronizedList***
  - Uses Concurrency APIs for thread-safety

- ***CopyOnWriteArrayList***
  - Modifications on the list cause are performed on a copy of an array
  - Efficient because no locking on traversal
  - Not-efficient because of memory copy

Concurrent Collections : Maps

- Two interfaces:
  - ***ConcurrentMap***
  - ***ConcurrentNavigableMap***
- Provide atomic operations for Map
  - ***putIfAbsent***
  - ***remove***
  - ***replace***
- Implementations include:
  - ***ConcurrentHashMap***
  - ***ConcurrentSkipListMap***

Concurrent Collections : Set

- Implementations include:
  - ***CopyOnWriteArraySet***
  - ***ConcurrentSkipListSet***
- More fine grained access control

Collection Framework: Queues

New ***`Queue`*** interface added to ***`java.util`***

- Represents some form of waiting list
- Implementations have different ordering algorithms
  - First-in-first-out (FIFO)
  - Last-in-first-out (LIFO)
  - Natural ordering
  - Priority
- Defined in terms of
  - Head (start of queue)
  - Tail (end of queue)

Collection Framework: Qs [cont.]

- Support normal-collection behaviors
  - ***`java.util.Collection`***
  - ***`java.util.Iterable`***

- Support new behaviors
  - Insertion:
    - ***`offer`*** - inserts element into queue; if space available
  - Removal:
    - ***`remove`*** - removes head of queue or throws ***`NoSuchElementException`***
    - ***`poll`*** - removes head of queue or ***`null`***
  - Viewing
    - ***`element`*** - retrieves head or throws ***`NoSuchElementException`***
    - ***`peek`*** - retrieves head or ***`null`***

Concurrent Queues

- Concurrency libraries provide concurrent implementations of **`Queue`** interface
  - ***`BlockingQueue`***
    - Adds waiting functionality to queue
    - ***`put`*** - adds to queue or waits for space
    - ***`take`*** - removes from queue or waits for availability
  - ***`BlockingDeque`***

Concurrent Queues [cont.]

Sample implementations:

- Bounded Implementations
  - ***ArrayBlockingQueue***
  - ***LinkedBlockingQueue***
  - ***LinkedBlockingDeque***

- Unbounded Implementations
  - ***PriorityBlockingQueue***
  - ***DelayQueue***

- Synchronous Implementation
  - ***SynchronousQueue***
  - Take waits for put / put waits for take
  - No “internal” capacity

Using LinkedBlockingQueue

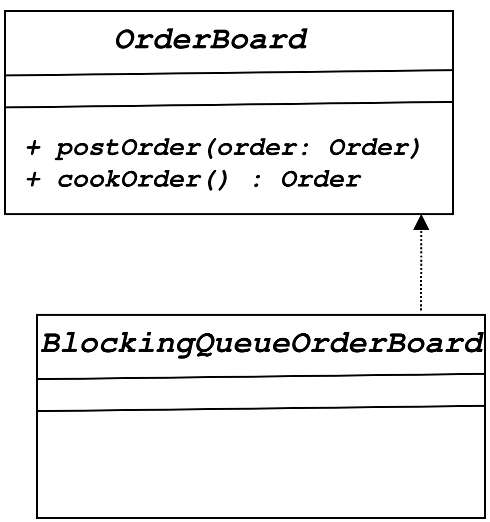
- Can be used to simplify producer - consumer problem

- Current solution uses ***`OrderBoard`***
  - ***`OrderBoard`*** manages synchronization
    - Obtains list object lock before modifying list
    - Release list object lock after modifying list
  - ***`OrderBoard`*** manages availability
    - Determines whether insert operation is valid
    - Determines whether remove operation is valid
    - Synchronizes threads appropriately

- ***`LinkedBlockingQueue`*** alternative
  - Manages synchronization of access
  - Manages availability of access

OrderBoard Redesign

- **`OrderBoard`** has been redesigned
  - Extracted interface
  - Enables us to create different implementations
  - Don’t need to modify cook or waiter



- **`BlockingQueueOrderBoard`**
  - Implementation of **`OrderBoard`**
  - Uses a bounded **`BlockingQueue`**
  - No synchronization
  - No queue empty / full management
  - Simplifies original **`OrderBoard`**

BlockingQueue Example

```java
package examples.concurrent.advanced;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

/**...*/
public class BlockingQueueOrderBoard implements OrderBoard {

  BlockingQueue<Order> orders;

  public BlockingQueueOrderBoard() {
    orders = new LinkedBlockingQueue<Order>(5);
  }

  public void postOrder(Order toBeProcessed) {
    try {
      orders.put(toBeProcessed);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

```

BlockingQueue Example [cont.]

```java
    public Order cookOrder() {
        Order returnValue = null;
        try {
            returnValue = orders.take();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        return returnValue;
    }
}
```

Lab: Rewrite Order Board

- **GOAL:** Refactor the order board to use a blocking queue from the concurrency library. The resulting code should be less complex, less lines of code. Yet the functionality will be exactly the same.

- **NOTE:** Keep an old copy of the order board around. It will be used in other upcoming labs.

- **DURATION:** 45 minutes
  - 30 minutes - development
  - 15 minutes - group code review.

Execution Framework

Delegate, delegate, delegate

Task Execution

Two task execution frameworks built into Java

1. ***`Thread`*** as an execution framework
   - ***`Runnable`*** becomes task
   - Thread governs when ***`run`*** is executed
   - No support for canceling, scheduled execution, etc.

1. ***`java.util.Timer`*** as execution framework
   - Introduced in 1.3
   1. Task represented as ***`TimerTask`***
   2. Supports canceling, fixed rate scheduling, date-based scheduling
   3. No real-time timing guarantees - relies on wait mechanism

Task Execution [cont.]

- Both task execution frameworks are “functional”, but somewhat incomplete
- Generally you need a more robust execution framework that provides:
  - Thread reuse and pooling
  - Task scheduling
  - Task canceling
  - Decoupling of task registration from execution

Concurrency Execution Framework

- Part of ***`java.util.concurrent`*** package
  - Decouples task execution from ***`Thread`*** dependency
  - Supports more robust task handling
  - Implemented using Factory and command-pattern

- Built around three key concepts:
  - Tasks
  - Executors
  - Execution services

Execution Framework Tasks

Two "tasks" in concurrency execution framework

1. ***java.lang.Runnable***
   - Standard ***Runnable***
   - Implement ***run*** method
   - Don't worry about Threading semantics

1. ***java.util.concurrent.Callable***
   - Similar to a ***Runnable***, in concept
   - Single method to implement
     - ***public V call()***

     1. Can return value
     2. Can throw checked exceptions

Executors

- Entities that execute tasks

- Represented by ***`java.util.concurrent.Executor`***
  - Decouples task submission from execution
  - Does not define how ***`Runnable`*** will be executed
  - ***`Executor`*** implementation could be:
    - Dedicated single-thread based
    - Thread-pool based
    - Current-thread based
  - Does not define when ***`Runnable`*** will be executed
  - Single task submission method

    `public void execute(Runnable cmd)`

Execution Services

- Entities responsible for execution and management of tasks
- Two types:
  - ***java.util.concurrent.ExecutorService***
    - Interface extensions of ***Executor***
    - Adds management capabilities to ***Executor***
      - Supports blocking - ***awaitTermination***
      - Shutdown - ***shutdown***
      - Service monitoring - ***isShutdown*** / ***isTerminated***
    - Enhances task handling
      - Submission – supports ***Callable and Runnable***
      - Management – returns ***Future***

Types of Execution Services [cont.]

- Two types (cont):
  - ***java.util.concurrent.ScheduledExecutorService***
    - Interface extensions of ***ExecutorService***
    - Adds scheduling capabilities to ***ExecutorService***
      - Supports Callable and Runnable
      - Single-schedule execution
      - Fixed-rate scheduled execution
      - Fixed-delay scheduled execution
      - No “date-based” scheduled execution

Task Management

- Every scheduled task has an associated “handle”
- Handle used for task cancellation and monitoring
- Handles are decoupled from:
  - Service - no way to get execution service reference from handle
  - Task - no way to get task reference from handle
- Two types of “handles”
  - ***`java.util.concurrent.Future`***
  - ***`java.util.concurrent.ScheduledFuture`***

NOTE: Handle type dependent on scheduling mechanism

Executor Implementations

- Create your own ***`Executor`*** implementation
  - Easiest way is to define a ***`ThreadFactory`***
  - Associate it with ***`ThreadPoolExecutor`***

- Or utilize the built-in implementations
  - ***`Executors`*** class is a factory
    - Can be used to create ***`ExecutorServices`***
      - Single thread
      - Cached thread pool
      - Fixed thread pool
      - Scheduled thread pool
    - Can be used to create ***`Callable`*** objects out of ***`Runnable`*** objects

Execution Framework Example

A Simplistic example

- Intended to illustrate use of an execution service
- Built around scheduled execution of a task
- Task performs HTTP ping-like functionality to determine availability of web server

Execution Framework Example

```java
package examples.concurrent.executer;

import ...

/**...*/
public class TimedPing {

    public static void main(String[] args) throws Exception {

        URL url = new URL(args[0]);
        HttpPinger pinger = new HttpPinger(url);

        //create a scheduled execution service
        //only need one thread to perform ping functionality
        ScheduledExecutorService pingService =
                Executors.newSingleThreadScheduledExecutor();

        //schedule the HttpPinger to ping every ping
        ScheduledFuture future =
                pingService.scheduleAtFixedRate(pinger, 30L,
                                                60L, TimeUnit.SECONDS);

        //schedule a task to cancel the pinger after 5 minutes
        //task should also notify the service to shutdown
        pingService.schedule(new CancelPinger(future, pingService),
                                                60*5, TimeUnit.SECONDS);
    }
}
```

Exec. Framework Example [cont.]

```java
package examples.concurrent.executer;

import ...

/**...*/
public class HttpPinger implements Runnable {

    private boolean keepTesting = true;
    private URL theHostToTest;
    private ScheduledFuture scheduledFuture;

    public HttpPinger(URL url) {
        theHostToTest = url;
    }
```

Exec. Framework Example [cont.]

```java
  public void run() {
    try {
      HttpURLConnection connection =
              (HttpURLConnection) theHostToTest.openConnection();

      //just see if we can access it
      connection.setRequestMethod("HEAD");
      connection.connect();

      //the HTTP response code
      int responseCode = connection.getResponseCode();

      if (responseCode != HttpURLConnection.HTTP_OK) {
        System.out.println("Failed attempt");
      } else {
        System.out.println("Connected ok: "+System.currentTimeMillis());
      }
      connection.disconnect();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Exec. Framework Example [cont.]

```java
package examples.concurrent.executer;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.ScheduledFuture;

/**...*/
public class CancelPinger implements Runnable {

   private ScheduledFuture future;
   private ExecutorService service;

   public CancelPinger(ScheduledFuture f, ExecutorService pingService) {
      future = f;
      service = pingService;
   }

   public void run() {
      future.cancel(false);
      service.shutdown();
   }
}
```

Lab: Task Execution

- **GOAL:** Implement the example. Allow more than one “pinger” to exist and “ping” the health of a website. After a website has been determined to be up (10 successful pings), change the frequency (the delay) from every 30 seconds to every minute.

- **HINT:** You will need more than a single threaded executor.

- **DURATION:** 45 minutes
  - 30 minutes - development
  - 15 minutes - group code review.

Working with Synchronizers

The Great Barrier Reef

Synchronizers

- Utility classes
  - Used to help coordinate control flow of Threads
  - Have their own state to determine “go” or “wait”
  - Potential replacements for synchronization blocks

- Three broad types:
  - Latches
  - Barriers
  - Semaphores

Latches

- Synchronizer that delays progress of threads
  - Threads are delayed until a terminal state is reached
  - Once reached, all threads can proceed

- Function like a gate:
  - When gate is closed, can't go through
  - When gate is open, can go through
  - Once gate is open, stays open

- Useful when trying to synchronize:
  - Resource initialization
  - Service startup
  - Application shutdown

CountDownLatch

- Implementation of a latch
  - Forces threads to wait until a predefined number of “events” occur
  - Utilizes a counter
  - As events occur, counter decrements
  - When **`count`** becomes 0, latch is released

- Can not be reused

Latch Example : LatchWaiter

```java
package examples.concurrent.advanced;

import java.util.concurrent.CountDownLatch;

/**...*/
public class LatchWaiter extends Thread {

  private CountDownLatch latch;

  public LatchWaiter(CountDownLatch latch) {
    this.latch = latch;
  }

  public void run() {
    try {
      latch.await();
    } catch (InterruptedException e) { }
    System.out.println("All threads completed, waiter is going to work");
  }
}
```

Latch Example : BusBoy(s)

```
package examples.concurrent.advanced;

import java.util.Random;
import java.util.concurrent.CountDownLatch;

/**...*/
public class BusBoy extends Thread {

  private static Random randomGenerator = new Random();

  private CountDownLatch latch;

  BusBoy(CountDownLatch latch) {
    this.latch = latch;
  }

  public void run() {
    try {
      latch.countDown();
      System.out.println("BusBoy cleaning table " + latch.getCount());
      int sleepTime = Math.abs(randomGenerator.nextInt());
      Thread.sleep(sleepTime);

    } catch(InterruptedException ie) {
      System.out.println(ie);
    }
  }
}
```

Latch Example : SmokeBreak

```java
package examples.concurrent.advanced;

import java.util.concurrent.CountDownLatch;

/**...*/
public class SmokeBreak {

  public static void main(String[] args) {
    CountDownLatch latch = new CountDownLatch(5);

    LatchWaiter waiter = new LatchWaiter(latch);
    waiter.start();

    for(int i=0;i<5;i++) {
      new BusBoy(latch).start();
    }
  }
}
```

Barriers

- Block threads until some “event” occurs
  - Used to “join” groups of threads
  - All threads must reach rendezvous point at same time
  - Once all threads reach barrier, then proceed
- Threads don’t die when they reach barrier
  - Different than ***`Thread.join()`***
  - They can continue processing
- Could be implemented using ***`wait`*** / ***`notify`*** mechanics
  - But might be messy
  - And potentially error prone

Barriers: `CyclicBarrier`

- Provides "blocking" point for threads
  - Constructed with number of threads in party
  - Each thread calls ***`await`*** when it gets to point
  - ***`CyclicBarrier`*** blocks ***`await`*** thread until all members of party arrive
  - Once all arrive, releases threads

- Can be reused - ***`reset`*** the barrier

CyclicBarrier Example : BusBoyBarrier

```java
package examples.concurrent.advanced;

import ...

/**...*/
public class BusBoyBarrier extends Thread {

    private static Random randomGenerator = new Random();

    private CyclicBarrier barrier;

    BusBoyBarrier(CyclicBarrier barrier) {
        this.barrier = barrier;
    }

    public void run() {
        try {
            System.out.println("BusBoy cleaning table ");
            int sleepTime = Math.abs(randomGenerator.nextInt());
            Thread.sleep(1000);
            System.out.println("BusBoys waiting: " + barrier.getNumberWaiting());
            barrier.await();

        } catch(InterruptedException ie) {
            System.out.println(ie);
        } catch (BrokenBarrierException e) {
            System.out.println(e);
        }
    }
}
```

CyclicBarrier Example : SmokeBreak

```java
package examples.concurrent.advanced;

import java.util.concurrent.CyclicBarrier;

/**...*/
public class SmokeBreak {

  public static void main(String[] args) {
    CyclicBarrier barrier = new CyclicBarrier(5, new Runnable() {
      public void run() {
        System.out.println("BusBoy Smoke Break");
      }
    });

    for(int i=0;i<5;i++) {
      new BusBoyBarrier(barrier).start();
    }

  }
}
```

Exchanger

- Barrier with data passing semantics

- Used with two threads
  - Meet at rendezvous point
  - Once there, **`exchange`** data
  - Continue on processing

Semaphore

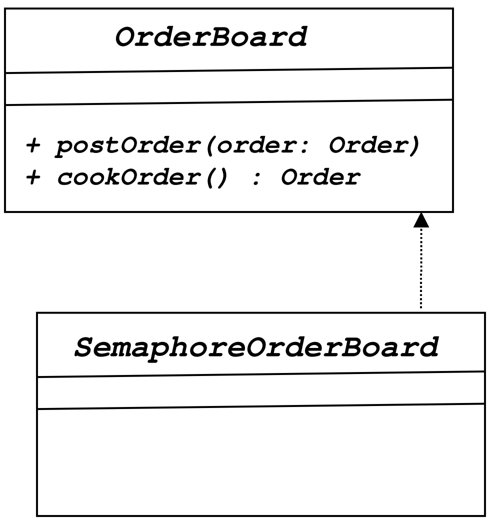
Formalization of counting semaphore

- Counting associated with set number of permits
- `Semaphore` with one permit is considered a mutex
- Removes counting semantics found in many synchronization techniques

Semaphore [cont.]

- Permits provide access control
  - Initialized to the number of resources it controls
  - Two key methods:
    - **`acquire`**
      - Decreases the number of available permits
      - Will wait if no permits available
    - **`release`** - increases number of available permits
  - Thread can hold more than one permit
  - Permit can be released by non-holding thread

OrderBoard Redesign



- ***SemaphoreOrderBoard***
  - Implementation of ***OrderBoard***
  - Uses two Semaphores
    - ***fullSem*** - initialized with 5 permits
    - ***emptySem*** - initialized with 0 permits
  - No synchronization - uses synchronized list
  - No queue empty / full management
  - Simplifies original ***OrderBoard***

Semaphore Example

```java
package examples.concurrent.advanced;

import java.util.ArrayList;
import java.util.List;
import java.util.Collections;
import java.util.concurrent.Semaphore;

/**...*/
public class SemaphoreOrderBoard implements OrderBoard {

  private List<Order> orders;
  private Semaphore fullSem, emptySem;


  /**...*/
  public SemaphoreOrderBoard() {
    orders = Collections.synchronizedList(new ArrayList<Order>());
    fullSem = new Semaphore(5);
    emptySem = new Semaphore(0);
  }
```

Semaphore Example [cont.]

```java
/**...*/
public void postOrder(Order toBeProcessed) {
  try {
    fullSem.acquire(); //decrease permits by one
    orders.add(toBeProcessed);
  } catch (Exception e) {
    e.printStackTrace();
  } finally {
    emptySem.release(); //increase permits by one
  }
}
```

Semaphore Example [cont.]

```java
    /**...*/
    public Order cookOrder() {
        Order tmpOrder = null;
        try {
            emptySem.acquire(); //decrease permits by one
            tmpOrder = orders.remove(0);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if(orders.size() < 3)
                fullSem.release(); //increae permits by one
        }

        return tmpOrder;
    }
}
```

Lab: Rewrite Order Board

- **GOAL:** Refactor **TheBurgetJoint** to create 3 waiter threads. Write code to ensure that, on start up, the three waiter threads wait for each other to initialize, and start work "at the same time". Hint – use a **CountDownLatch.**

- **NOTE:** Keep an old copy of the order board around. It will be used in other upcoming labs.

- **DURATION:** 45 minutes
  - 30 minutes - development
  - 15 minutes - group code review.

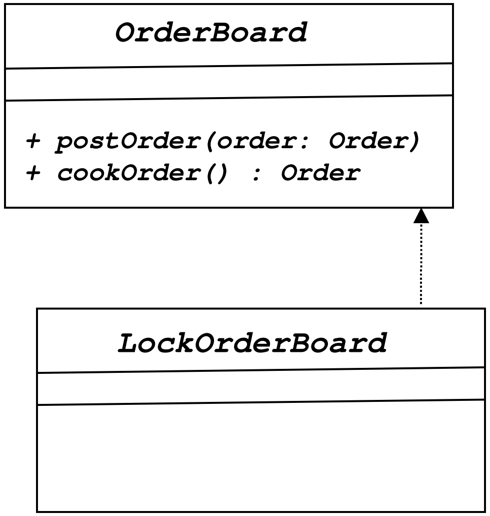
Other Concurrency Packages

`java.util.concurrent.locks`

- Locking and waiting condition framework
  - Alternative to monitor lock mechanism
  - Provides greater flexibility
    - Locks do not require synchronized blocks
    - Locks support re-entrance and fairness policies
    - Locks have multiple conditions
    - Waiting based on condition not “object lock”

- Key components:
  - `java.util.concurrent.locks.Lock`
  - `java.util.concurrent.locks.Condition`
  - `java.util.concurrent.locks.ReentrantLock`

OrderBoard Redesign



- ***`LockOrderBoard`***
  - Implementation of ***`OrderBoard`***
  - Uses one Lock
    - ***`ReentrantLock`***
    - Used to synchronize access to orders list
  - Access controlled by two conditions
    - ***`full`***
    - ***`empty`***
  - Queue empty / full management

Lock Example

```java
package examples.concurrent.advanced;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**...*/
public class LockOrderBoard implements OrderBoard {

    List<Order> orders;

    Lock fullLock = new ReentrantLock();
    Condition full = fullLock.newCondition();
    Condition empty = fullLock.newCondition();


    public LockOrderBoard() {
        orders = new ArrayList<Order>();
    }
```

Lock Example [cont.]

```
public void postOrder(Order toBeProcessed) {
    try {
        fullLock.lock();
        while(orders.size() == 5) {
            full.await();
        }
        orders.add(toBeProcessed);
        empty.signalAll();
    } catch(Exception e) {
        e.printStackTrace();
    }
    fullLock.unlock();

}
```

Lock Example [cont.]

```java
public Order cookOrder() {
    Order returnValue = null;
    try {
        fullLock.lock();
        while(orders.size() == 0) {
            empty.await();
        }
        returnValue = orders.remove(0);
        full.signalAll();
    } catch(Exception e) {
        e.printStackTrace();
    }
    fullLock.unlock();
    return returnValue;
}
}
```

java.util.concurrent.atomic

- Toolkit of classes
  - Provide atomic manipulation of variables
  - Uses lock-free thread-safe implementation

- Extends ***volatile*** using ***compareAndSet*** functionality
  - Relies on enhancements made to JVM
  - Take advantage of compare-and-swap or load-linked/store-condition hardware based operations
  - Foundational for entire concurrency package

- May or may not be used at application development level

java.util.concurrent.atomic

- AtomicInteger, AtomicLong, AtomicBoolean etc.
- Atomic update operations:
  - getAndDecrement() → like number--
  - incrementAndGet() → like ++number
  - compareAndSet(int expectedValue, int newValue) → sets the newValue if current value == expectedValue

Lab: Rewrite Order Board

- **GOAL:** Refactor **TheBurgetJoint** to create 3 waiter threads. Write code to ensure that, on start up, the three waiter threads wait for each other to initialize, and start work "at the same time". (Hint – use a **CountDownLatch.**)

You will also have to change the counter that is used in the **Order** class to make it thread safe. (AtomicInteger would be useful here)

- **NOTE:** Keep an old copy of the order board around. It will be used in other upcoming labs.

- **DURATION:** 45 minutes

Summary

- Java SE 5.0 exponentially expands concurrent programming in Java
- Concurrency libraries provide standardization to common concurrent problems
- Concurrency libraries provide:
  - Concurrent collections
  - Synchronizers
  - Locks
  - Atomic wrappers