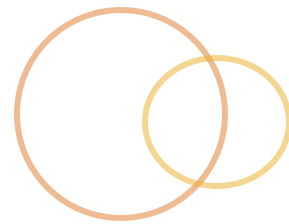


Advanced Concurrent Programming



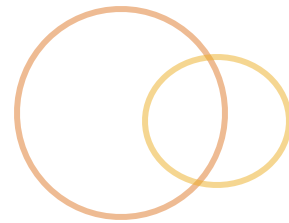
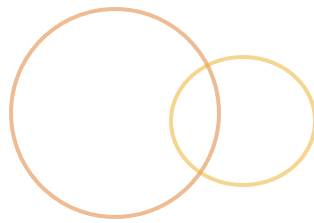
Presentation Topics



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



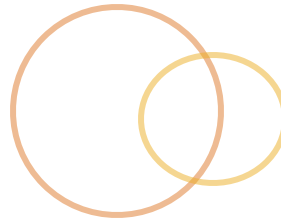
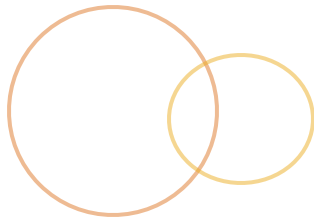
- ◎ Address limitations of Java's thread model
- ◎ Standardize and simplify common concurrency mechanisms
 - ◎ Lower complexity in development concurrent programs
 - ◎ Increase maintainability of concurrent code
 - ◎ Lessen common “concurrency” issues
- ◎ Provide robust, efficient, and high-performance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

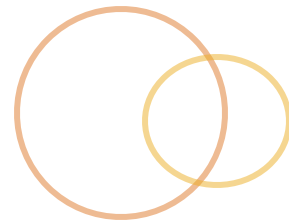
- Removal:

- `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

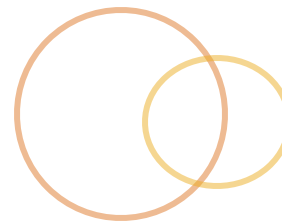
- ☉ `SynchronousQueue`
- ☉ Take waits for put / put waits for take
- ☉ No “internal” capacity

Using LinkedBlockingQueue

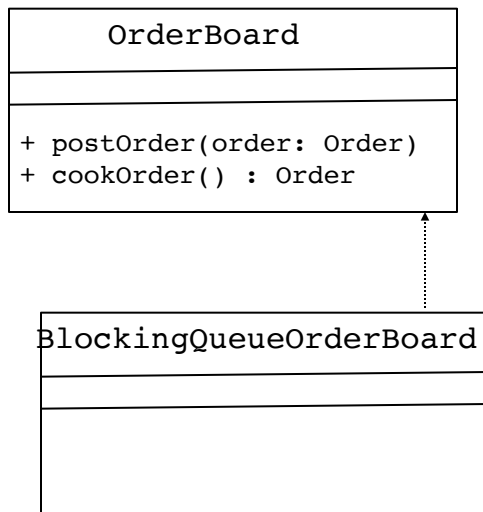


- Can be used to simplify producer - consumer problem
- Current solution uses OrderBoard
 - OrderBoard manages synchronization
 - Obtains list object lock before modifying list
 - Release list object lock after modifying list
 - OrderBoard manages availability
 - Determines whether insert operation is 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



```
1  package examples.concurrent.advanced;
2
3  import java.util.concurrent.BlockingQueue;
4  import java.util.concurrent.LinkedBlockingQueue;
5
6  /** ... */
13 public class BlockingQueueOrderBoard implements OrderBoard {
14
15     BlockingQueue<Order> orders;
16
17     public BlockingQueueOrderBoard() {
18         orders = new LinkedBlockingQueue<Order>(5);
19     }
20
21     public void postOrder(Order toBeProcessed) {
22         try {
23             orders.put(toBeProcessed);
24         } catch (InterruptedException e) {
25             e.printStackTrace();
26         }
27     }
28 }
```

BlockingQueue Example [cont.]



```
29 public Order cookOrder() {  
30     Order returnValue = null;  
31     try {  
32         returnValue = orders.take();  
33     } catch (InterruptedException e) {  
34         e.printStackTrace();  
35     }  
36  
37     return returnValue;  
38 }  
39 }  
40
```

Lab: Rewrite Order Board



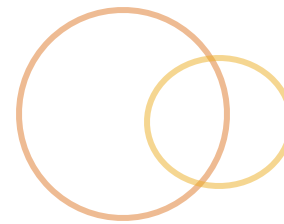
- ◎ **GOAL:** Refactor the order board to use a blocking queue from the concurrency library. The resulting code should be less complex, less lines of code. Yet the functionality will be exactly the same.
- ◎ **NOTE:** Keep an old copy of the order board around. It will be used in other upcoming labs.
- ◎ **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

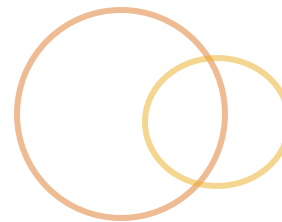
1. Thread as an execution framework

- ⦿ `Runnable` becomes task
- ⦿ Thread governs when `run` is executed
- ⦿ No support for canceling, scheduled execution, etc.

2. `java.util.Timer` as execution framework

- ⦿ Introduced in 1.3
 - ⦿ Task represented as `TimerTask`
 - ⦿ Supports canceling, fixed rate scheduling, date-based scheduling
1. 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

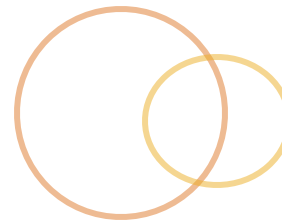
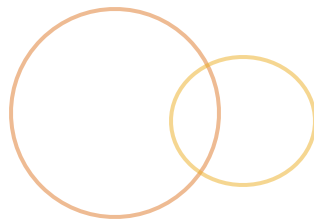
1. `java.lang.Runnable`

- ◎ Standard Runnable
- ◎ Implement `run` method
- ◎ Don't worry about Threading semantics

2. `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

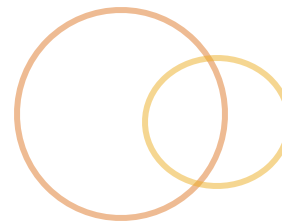


Entities that execute tasks

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`

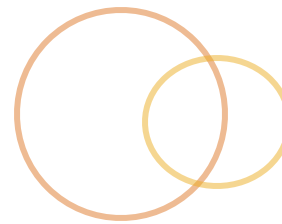
Types of Execution Services [cont.]



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



```
1  package examples.concurrent.executer;
2
3  +import ...
8
9  +/**...*/
21 public class TimedPing {
22
23  - public static void main(String[] args) throws Exception {
24
25      URL url = new URL(args[0]);
26      HttpPinger pinger = new HttpPinger(url);
27
28      //create a scheduled execution service
29      //only need one thread to perform ping functionality
30      ScheduledExecutorService pingService =
31          Executors.newSingleThreadScheduledExecutor();
32
33      //schedule the HttpPinger to ping every ping
34      ScheduledFuture future =
35          pingService.scheduleAtFixedRate(pinger, 30L,
36                                          60L, TimeUnit.SECONDS);
37
38      //schedule a task to cancel the pinger after 5 minutes
39      //task should also notify the service to shutdown
40      pingService.schedule(new CancelPinger(future, pingService),
41                          60*5, TimeUnit.SECONDS);
42  }
43
```

Exec. Framework Example [cont.]



```
1  package examples.concurrent.executer;
2
3  +import ...
6
7  +/** ... */
11 public class HttpPinger implements Runnable {
12
13     private boolean keepTesting = true;
14     private URL theHostToTest;
15     private ScheduledFuture scheduledFuture;
16
17     - public HttpPinger(URL url) {
18         theHostToTest = url;
19     }
20 }
```

Exec. Framework Example [cont.]



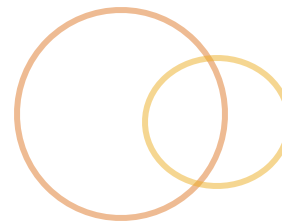
```
21 public void run() {
22     try {
23         HttpURLConnection connection =
24             (HttpURLConnection) theHostToTest.openConnection();
25
26         //just see if we can access it
27         connection.setRequestMethod("HEAD");
28         connection.connect();
29
30         //the HTTP response code
31         int responseCode = connection.getResponseCode();
32
33         if (responseCode != HttpURLConnection.HTTP_OK) {
34             System.out.println("Failed attempt");
35         } else {
36             System.out.println("Connected ok: "+System.currentTimeMillis());
37         }
38         connection.disconnect();
39     } catch (Exception e) {
40         e.printStackTrace();
41     }
42 }
43 }
```

Exec. Framework Example [cont.]



```
1  package examples.concurrent.executer;
2
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.ScheduledFuture;
5
6  /** ... */
10 public class CancelPinger implements Runnable {
11
12     private ScheduledFuture future;
13     private ExecutorService service;
14
15     public CancelPinger(ScheduledFuture f, ExecutorService pingService) {
16         future = f;
17         service = pingService;
18     }
19
20     public void run() {
21         future.cancel(false);
22         service.shutdown();
23     }
24 }
```

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 ever 30 seconds to every minute.

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

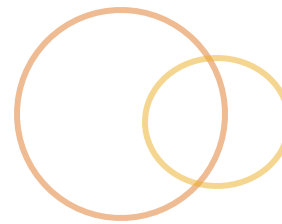
- ◎ **DURATION:** 45 minutes
 - ◎ 30 minutes - development
 - ◎ 15 minutes - group code review.

Working with Synchronizers

The Great Barrier Reef



Synchronizers



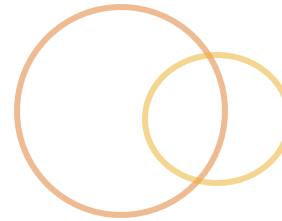
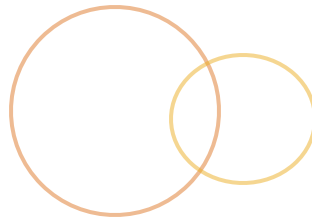
Utility classes

- 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:

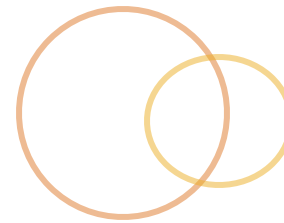
- Latches
- Barriers
- Semaphores

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



```
1 package examples.concurrent.advanced;
2
3 import java.util.concurrent.CountDownLatch;
4
5 /**...*/
12 public class LatchWaiter extends Thread {
13
14     private CountDownLatch latch;
15
16     public LatchWaiter(CountDownLatch latch) {
17         this.latch = latch;
18     }
19
20     public void run() {
21         try {
22             latch.await();
23         } catch (InterruptedException e) { }
24         System.out.println("All threads completed, waiter is going to work");
25     }
26 }
```

Latch Example : BusBoy(s)



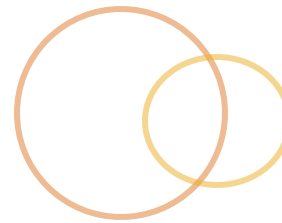
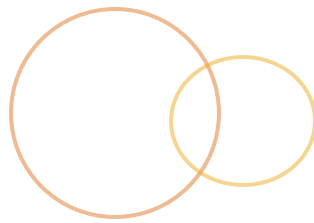
```
1 package examples.concurrent.advanced;
2
3 import java.util.Random;
4 import java.util.concurrent.CountDownLatch;
5
6 /**...*/
13 public class BusBoy extends Thread {
14
15     private static Random randomGenerator = new Random();
16
17     private CountDownLatch latch;
18
19     BusBoy(CountDownLatch latch) {
20         this.latch = latch;
21     }
22
23     public void run() {
24         try {
25             latch.countDown();
26             System.out.println("BusBoy cleaning table " + latch.getCount());
27             int sleepTime = Math.abs(randomGenerator.nextInt());
28             Thread.sleep(sleepTime);
29
30         } catch (InterruptedException ie) {
31             System.out.println(ie);
32         }
33     }
34 }
```

Latch Example : SmokeBreak



```
1  package examples.concurrent.advanced;
2
3  import java.util.concurrent.CountDownLatch;
4
5  /**...*/
12 public class SmokeBreak {
13
14     public static void main(String[] args) {
15         CountDownLatch latch = new CountDownLatch(5);
16
17         LatchWaiter waiter = new LatchWaiter(latch);
18         waiter.start();
19
20         for(int i=0;i<5;i++) {
21             new BusBoy(latch).start();
22         }
23     }
24 }
25
```

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

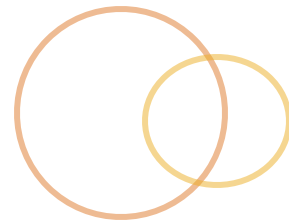


```
1 package examples.concurrent.advanced;
2
3 +import ...
4
5
6
7 +/**...*/
8
14 public class BusBoyBarrier extends Thread {
15
16     private static Random randomGenerator = new Random();
17
18     private CyclicBarrier barrier;
19
20     BusBoyBarrier(CyclicBarrier barrier) {
21         this.barrier = barrier;
22     }
23
24 Ⓢ↑ public void run() {
25     try {
26         System.out.println("BusBoy cleaning table ");
27         int sleepTime = Math.abs(randomGenerator.nextInt());
28         Thread.sleep(1000);
29         System.out.println("BusBoys waiting: " + barrier.getNumberWaiting());
30         barrier.await();
31
32     } catch (InterruptedException ie) {
33         System.out.println(ie);
34     } catch (BrokenBarrierException e) {
35         System.out.println(e);
36     }
37 }
38 }
```

CyclicBarrier Example : SmokeBreak

```
1 package examples.concurrent.advanced;
2
3 import java.util.concurrent.CyclicBarrier;
4
5 /** ... */
12 public class SmokeBreak {
13
14     public static void main(String[] args) {
15         CyclicBarrier barrier = new CyclicBarrier(5, new Runnable() {
16             public void run() {
17                 System.out.println("BusBoy Smoke Break");
18             }
19         });
20
21         for(int i=0; i<5; i++) {
22             new BusBoyBarrier(barrier).start();
23         }
24     }
25 }
26 }
```

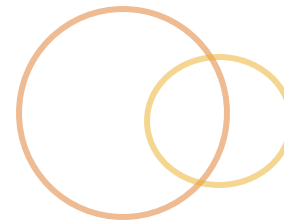
Exchanger



- Barrier with data passing semantics

- Used with two threads

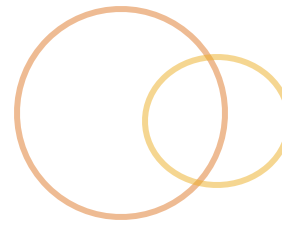
- Meet at rendezvous point
- Once there, exchange data
- Continue on processing



Formalization of counting semaphore

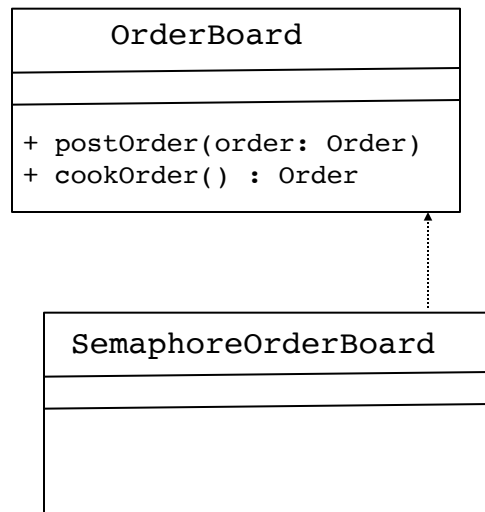
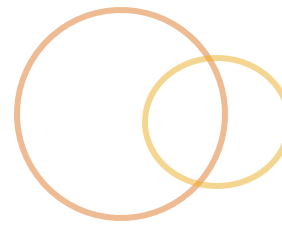
- ⦿ Counting associated with set number of permits
- ⦿ Semaphore with one permit is considered a mutex
- ⦿ Removes counting semantics found in many synchronization techniques

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



```
1  package examples.concurrent.advanced;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.Collections;
6  import java.util.concurrent.Semaphore;
7
8  /** ... */
16 public class SemaphoreOrderBoard implements OrderBoard {
17
18     private List<Order> orders;
19     private Semaphore fullSem, emptySem;
20
21     /** ... */
26     public SemaphoreOrderBoard() {
27         orders = Collections.synchronizedList(new ArrayList<Order>());
28         fullSem = new Semaphore(5);
29         emptySem = new Semaphore(0);
30     }
```


Semaphore Example [cont.]



```
32  /**...*/
36  public void postOrder(Order toBeProcessed) {
37      try {
38          fullSem.acquire(); //decrease permits by one
39          orders.add(toBeProcessed);
40      } catch (Exception e) {
41          e.printStackTrace();
42      } finally {
43          emptySem.release(); //increase permits by one
44      }
45  }
46
```

Semaphore Example [cont.]



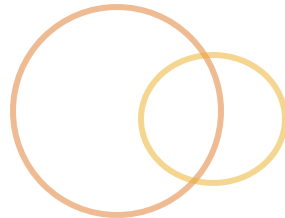
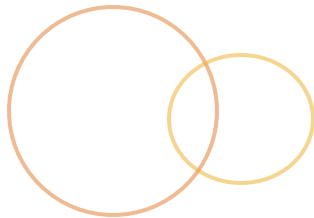
```
47  /**...*/
53  public Order cookOrder() {
54      Order tmpOrder = null;
55      try {
56          emptySem.acquire(); //decrease permits by one
57          tmpOrder = orders.remove(0);
58      } catch (Exception e) {
59          e.printStackTrace();
60      } finally {
61          if(orders.size() < 3)
62              fullSem.release(); //increae permits by one
63      }
64
65      return tmpOrder;
66  }
67 }
```

Lab: Rewrite Order Board



- ◎ **GOAL:** Refactor the order board to use a Semaphore 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.

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

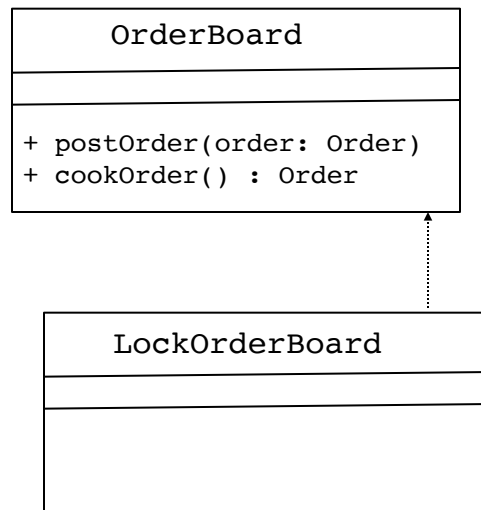
⦿ Used to synchronize access to orders list

⦿ Access controlled by two conditions

⦿ full

⦿ empty

⦿ Queue empty / full management



Lock Example



```
1 package examples.concurrent.advanced;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.concurrent.locks.Condition;
6 import java.util.concurrent.locks.Lock;
7 import java.util.concurrent.locks.ReentrantLock;
8
9 /** ... */
13 public class LockOrderBoard implements OrderBoard {
14
15     List<Order> orders;
16
17     Lock fullLock = new ReentrantLock();
18     Condition full = fullLock.newCondition();
19     Condition empty = fullLock.newCondition();
20
21
22     public LockOrderBoard() {
23         orders = new ArrayList<Order>();
24     }
25 }
```

Lock Example [cont.]



```
26  public void postOrder(Order toBeProcessed) {  
27      try {  
28          fullLock.lock();  
29          while(orders.size() == 5) {  
30              full.await();  
31          }  
32          orders.add(toBeProcessed);  
33          empty.signalAll();  
34      } catch(Exception e) {  
35          e.printStackTrace();  
36      }  
37      fullLock.unlock();  
38  }  
39  }
```


Lock Example [cont.]

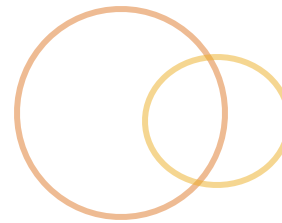
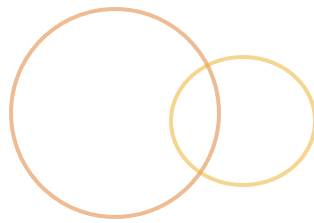


```
41  public Order cookOrder() {  
42      Order returnValue = null;  
43      try {  
44          fullLock.lock();  
45          while(orders.size() == 0) {  
46              empty.await();  
47          }  
48          returnValue = orders.remove(0);  
49          full.signalAll();  
50      } catch(Exception e) {  
51          e.printStackTrace();  
52      }  
53      fullLock.unlock();  
54      return returnValue;  
55  }  
56 }
```

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

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

Advanced Concurrency Lab



🎯 **GOAL:** Create a basic site map creation utility.

The site map creation utility should generate a text file containing all of the domain-specific URLs found on a given website. Associated with each domain, should be a indicator denoting whether the URL was accessible. The site map utility does not need to track images, only `<a href>` tags.

Use multiple threads to make the utility efficient. There should be one thread that finds `<a href>`s, and one thread that processes `<a href>`s.

When all of the `<a href>`s are processed, the application should generate a text file and then shut-down gracefully.

🎯 **HINTS:** A map could be used to represent the “indexed” site. A queue could be used as the shared resource representing the unindexed / untested elements of the site.

Advanced Concurrency Lab



- ◎ **DURATION:** 120 minutes
 - ◎ 100 minutes – development (consider pair programming)
 - ◎ 20 minutes – group code review