# Lecture 03

## Concurrent Programming in Java

### Conditional synchronization, native library, and implementation of elementary synchronization mechanisms

*Object-Oriented Concurrent Programming, 2019-2020*

*v2.3, 02-10-2019*

Miguel Oliveira e Silva
DETI, Universidade de Aveiro

# Conditional synchronization

# Conditional synchronization

- In the communication between different threads sometimes it is necessary to ensure a certain condition before executing a given code (for example, a thread for printing documents could be scheduled to wait for a non-empty queue);
- This communication requirement is called conditional synchronization, and its implementation can be done in two ways;
- It could *actively* be waiting for the condition:

```
while (!condition)
   Thread.yield();
```

- However, if the wait is long, the processor will be actively working on anything relevant (instead of doing useful work elsewhere);

# Conditional synchronization (2)

- Alternatively, a cooperative mechanism could be used, in which some threads voluntarily wait (without CPU consumption), and threads that might change the condition explicitly notify such modifications, possibly awakening the former;
- This mechanism is called condition variables;
- In `Java`, this behavior is implemented by methods defined in the `Object` class: `wait`, `notify`, and `notifyAll`.

```
package java.lang;

public class Object {
   public void notify();
   public void notifyAll();
   public void wait() throws InterruptedException;
   public void wait(long timeout) throws InterruptedException;
   public void wait(long timeout, int nanos)
      throws InterruptedException;
   ...
}
```

# Conditional synchronization (3)

- There is a very close relationship between `synchronized` regions and condition variables;
- A conditional wait (`wait`) **has** to be done inside a `synchronized` block (or function) of the **same** object (otherwise, a race condition might occur).
- Although condition variables are based on a cooperative wait/notify mechanism, the mechanism allows waiting threads to be awaken without an explicit notification (*spurious wakeups*);

# Conditional synchronization (4)

- Thus a correct use of this mechanism **requires** that the invocation of service `wait` is inside a cycle checking the waiting condition:

```
while (!condition)
   wait();
```

- This requirement also eases the reuse of the same condition variable for several conditions;
- In this case, however: (1) either it is necessary to use the `notifyAll` service (which is the safest and simpler option); (2) or use `notify`, and whenever the wrong thread is awoken, re-notify other possibly waiting threads:

```
while (!condition) {
   wait();
   if (!condition)
      notify();
}
```

# Conditional synchronization: the problem of checked exceptions

- One of the practical problems posed by native Java's conditional synchronization interface, is the requirement to explicitly deal with `InterruptedException` exceptions;
- If in theory this group of exceptions (named *checked*) seems to make sense, in practice it has shown to be a real menace to program quality (arising several correctness and robustness problems);
- In fact, program developers usually do not know what to do to handle exceptions (or, simply, don't want to "waste time" on that, delaying that problem to a later time) leaving empty `catch` blocks (or only with a log registry, which is logically the same thing);
- The possibility to declare within the method's signature the list of exceptions it may launch (an obligation when uncatched checked exceptions are involved), is also a very questionable alternative because, not only may require a lot of bookkeeping code, but also it may break algorithmic abstraction (which, let us not forget, is the essence of the methods);

# Conditional synchronization: the problem of checked exceptions

- All these problems can be avoided by implementing waiting methods in which such exceptions are transformed into a proper *unchecked exception*. Such as: `ThreadInterruptedException` (such as the one defined in `pt.ua.concurrent` library);
- We can also create a new class to be used in place of the `Object` class where we abstract all this functionality:

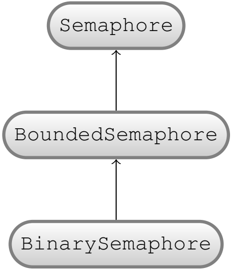
```
public class SimplerObject {
   protected SimplerObject() { super(); }

   public void await() {
      try { wait(); }
      catch(InterruptedException e) {
         throw new ThreadInterruptedException(e);
      }
   }
   ...
}
```

- The problem with this option is that it is only applicable to classes that do not need to extend other classes. That is, classes that extend it directly or indirectly.

# Basic synchronization mechanisms

# Semaphores

- A semaphore is a special shared non-negative integer variable, to which the following operations are applicable:
  - acquire (lock, wait, P, decrement, down)
  - release (unlock, signal, V, increment, up)
- The acquire operation ensures a non-negative result, thus it may block until such outcome is possible.
- In general, a maximum counter value is attached to a semaphore (maximum number of concurrent accesses to the resource).
- A semaphore is said to be *binary* if this value is one.
- We can establish a typing relation between different semaphore types:



# Semaphores: Interface

```
public class Semaphore {
   /** Creates an unbounded semaphore */
   public Semaphore();

   /** Creates an unbounded semaphore with
    * counter initialized to initialCount.
    *  <P><B>requires</B>: {@code initialCount >= 0}
    */
   public Semaphore(int initialCount);

   /** Decrements counter (waits if counter equals zero) */
   public synchronized void acquire();

   /** Increments counter */
   public synchronized void release();
}
```

# Semaphores: Interface (2)

```java
public class BoundedSemaphore extends Semaphore {
   /** Creates a semaphore with maxCount upper limit.
    *  <P><B>requires</B>: {@code maxCount > 0}
    */
  public BoundedSemaphore(int maxCount);

   /** Creates a semaphore with maxCount upper limit
    *  and counter initialized to initialCount.
    *  <P><B>requires</B>: {@code maxCount > 0}
    *  <P><B>requires</B>: {@code initialCount >= 0 &&
    *                             initialCount <= maxCount}
    */
  public BoundedSemaphore(int maxCount, int initialCount);
}

public class BinarySemaphore extends BoundedSemaphore {
   /** Creates a binary semaphore */
   public BinarySemaphore();

   /** Creates a binary semaphore initialized to initialCount.
    *  <P><B>requires</B>: {@code initialCount >= 0 &&
    *                             initialCount <= maxCount}
    */
   public BinarySemaphore(int initialCount);
}
```

# Mutex

- Synchronization mechanism that gives a single thread the exclusive access to a shared resource.
- It is characterized for having (at least) the following operations:
  - `lock`
  - `unlock`
- Unlike a binary semaphore, it is also characterized by having lock ownership. Thus, an `unlock` operation can only be executed by the owner thread (i.e. the current lock thread).
- A mutex could be defined to be recursive (i.e. the mutex owner thread might perform recursive locks).

# Mutex: Interface

```
public class Mutex {
   /** Creates a non-recursive Mutex */
   public Mutex();

   /** Creates a Mutex */
   public Mutex(boolean recursive);

   /** Locks mutex.
    *  <P><B>requires</B>: {@code !lockIsMine() || recursive()}
    *  <P><B>ensures</B>: {@code lockIsMine()}
    */
   public synchronized void lock();

   /** Unlocks mutex.
    *  <P><B>requires</B>: {@code lockIsMine()}
    */
   public synchronized void unlock();

   /** Is mutex locked by current thread? */
   public synchronized boolean lockIsMine();

   /** Is mutex recursive? */
   public synchronized boolean recursive();
}
```

# Condition variables

- Synchronization mechanism used to safely pass indicative information on the state of the shared resource.
- Is is characterized by the following operations:
    - `await` (`wait`)
    - `signal` (`notify`)
    - `broadcast` (`notifyAll`)
- A change in the state attached to a condition variable, should be followed by the execution of a `signal` or `broadcast` operation.
- An exclusive access to the resource that does not finds the required (waiting) state should be followed by the execution of an `await` operation that atomically waits and releases the resource (otherwise a race condition could occur).

# Condition variables: Interface

```
public class CVMutex {
   /** Creates a condition variable attached to mutex mtx
    *  <P><B>requires</B>: {@code mtx != null}
    */
   CVMutex(Mutex mtx);

   /** Mutex attached to condition variable */
   public final Mutex mtx;

   /** Waits signaling.
    *  <P><B>requires</B>: {@code mtx.lockIsMine() &&
    *                              mtx.lockCount() == 1}
    */
   public synchronized void await();

   /** Signal one waiting thread */
   public synchronized void signal();

   /** Signal all waiting thread */
   public synchronized void broadcast();
}
```

# Readers-writer exclusion





- Synchronization mechanism that gives exclusive access to a single *writer* (a thread executing a method that might change the resource state) at each time, but allows the concurrent execution to multiple readers (threads execution side-effect free query methods).
- Is is characterized by the following operations:
  - `lockReader`
  - `unlockReader`
  - `lockWriter`
  - `unlockWriter`

# Readers-writer exclusion: Interface

```
public class RWEx {
   public enum Priority {WRITER, READERS};
   /** Creates a non-recursive RWEx */
   public RWEx(Priority priority);
   /** Is current lock mine? */
   public synchronized boolean lockIsMine();
   /** Is current lock mine as a reader? */
   public synchronized boolean readerLockIsMine();
   /** Is current lock mine as a writer? */
   public synchronized boolean writerLockIsMine();
   /** Locks RWEx as a reader.
    * <P><B>requires</B>: {@code !lockIsMine()}
    * <P><B>ensures</B>: {@code readerLockIsMine()} */
   public synchronized void lockReader();
   /** Locks RWEx as a writer.
    * <P><B>requires</B>: {@code !lockIsMine()}
    * <P><B>ensures</B>: {@code writerLockIsMine()} */
   public synchronized void lockWriter();
   /** Unlocks reader RWEx lock.
    * <P><B>requires</B>: {@code readerLockIsMine()}
    * <P><B>ensures</B>: {@code !lockIsMine()} */
   public synchronized void unlockReader();
   /** Unlocks writer RWEx lock.
    * <P><B>requires</B>: {@code writerLockIsMine()}
    * <P><B>ensures</B>: {@code !lockIsMine()} */
   public synchronized void unlockWriter();
}
```

# Mutual exclusion with Semaphores

- The condition variable mechanism is a programming pattern that can easily be applied to any use of condition synchronization.
- On the other hand, although semaphores are not equivalent to a conditional variable, they are a basic synchronization mechanism implemented for all types of concurrent systems (UNIX processes, for instance, have semaphores, but not condition variables).
- Mutual exclusion is easily implemented with a single binary semaphore.

```java
Semaphore mtx = new BinarySemaphore(1);

// lock:
mtx.acquire();

// unlock:
mtx.release();
```

# Conditional synchronization with Semaphores

- Semaphores can also be applied to implement general condition variables. Lets see how.
- The first thing to be noted is that a condition variable is required to be attached with a mutual exclusion mechanism. Hence, an exclusion binary semaphore is also required.
- The second characteristic of condition variables is that a group of threads may be required to wait for proper signaling. Hence, an integer waiting counter is also necessary.
- Finally, this mechanism requires an atomic behavior joining signaling and mutex acquisition.
- Considering all these requirements, a general solution is the following:

# Conditional synchronization with Semaphores

Mutex code:

```
// variables:
Mutex mtx = new Mutex();

MutexCV cvar = mtx.newCV();



// wait:
mtx.lock();
while (!condition)
   cvar.wait();






...
mtx.unlock();

// broadcast:
mtx.lock();
...
cvar.broadcast();




mtx.unlock();
```

Semaphore code:

```
// variables:
Semaphore mtx =
   new BinarySemaphore(1);
int waitingCounter = 0;
Semaphore waitingCondition =
   new Semaphore(0);

// wait:
mtx.acquire();
while (!condition)
{
   waitingCounter++;
   mtx.release();
   waitingCondition.acquire();
   mtx.acquire();
}
...
mtx.release();

// broadcast:
mtx.acquire();
...
while(waitingCounter > 0)
{
   waitingCondition.release();
   waitingCounter--;
}
mtx.release();
```

# Concurrency Library: Lock

# Explicit synchronization

- The Java native library contains classes that allow the use of explicit locks: `Lock` (and `ReentrantLock`) interfaces;
- This classes surpass some limitations of the `synchronized` blocks:
  - Do not require in association to a structured block;
  - Allow interruption (method `lockInterruptibly`);
  - Provide `tryLock` service;
  - There are different implementations (adapted to each use).
- Usage pattern:

```
lock.lock();
try {
   // action
}
finally {
   lock.unlock();
}
```

# Lock interface

- Interface:

```
package java.util.concurrent.locks;

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

- (See also ReentrantLock)

# Concurrency Library: Condition

# `Condition` interface

- Interface:

```java
package java.util.concurrent.locks;

interface Condition {
   void await() throws InterruptedException;
   void awaitUninterruptibly();
   boolean await(long time, TimeUnit unit)
      throws InterruptedException;
   long awaitNanos(long nanosTimeout)
      throws InterruptedException;
   boolean awaitUntil(Date deadline)
      throws InterruptedException;
   void signal();
   void signalAll();
}
```

- Allows more than one condition variable in each object.

# Concurrency Library: Tasks

# Tasks, methods, and threads

- A concurrent program organizes itself around the execution of *threads*;
- The creation an execution of threads is clearly based on the message passing communication model, but in which the communicating active entities do so by the `Runnable` interface;
- In this model, the asynchronous execution of method is desired, in which `Thread` class binds a method (`run`) to a thread;
- This direct one-to-one connection between tasks and threads may, sometimes, create problems:
  - The overhead involved in the creation/destruction of threads;
  - Resource consumption, *memory* in particular;
  - Scalability: there is a practical limit to the number of threads that may coexist.
- Concurrency abstractions based on the separation between “task” and “thread” provides a controllable solution to this limitations.

# Concurrency Library: Executor

# Tasks



- A scalable alternative that avoids these problems is the usage of a task library;
- The package `java.util.concurrent` provides such a group of functionalities.
- The topmost abstraction defined in this package toward such a goal is the `Executor`:

```
package java.util.concurrent;

public interface Executor {
   void execute(Runnable task);
}
```

- This abstraction is a direct replacement of the thread's `start` method (or `run` method of `Runnable` class).

# Tasks

- “Under” this interface different thread control schemes can be built:
- Sequential:

```
public class SeqExecutor implements Executor {
   public void execute(Runnable task) {
      task.run();
   }
}
```

- One task per *thread*:

```
public class ThreadExecutor implements Executor {
   public void execute(Runnable task) {
      new Thread(task).start();
   }
}
```

- Or any other.

# Concurrency Library: Thread-pools

# Creating task managers

- The `java.util.concurrent` library provide a rich set of `Executor` modules that covers the more common needs.
- `Executors` class provides static methods that give a simple instantiation:

```
package java.util.concurrent;

public class Executors {
   static ExecutorService newSingleThreadExecutor();
   static ExecutorService newFixedThreadPool(int nThreads);
   static ExecutorService newCachedThreadPool();
   static ScheduledExecutorService newScheduledThreadPool(int corePoolSize);
   ...
}
```

## `Executors`

- `newSingleThreadExecutor`
  - Creates a single thread that is reused by the submitted tasks (stored internally in an unbounded queue).
- `newFixedThreadPool`
  - Creates a fixed predefined number of threads that are reused by the submitted tasks.
- `newCachedThreadPool`
  - Automatically increases or reduces the number of threads that are reused by the submitted (the number is adapted by the number of submitted tasks).
- `newScheduledThreadPool`
  - Allows the execution of tasks with a pre-defined time schedule. We can launch tasks to be performed after a time delay, or launch tasks to be performed periodically (like timers). This type of thread-pool replaces with many advantages the `Timer` class.

# ExecutorService



- The abstraction `ExecutorService` allows the management of the `Executor` lifecycle:

```
package java.util.concurrent;

public interface ExecutorService extends Executor {
    boolean isShutdown();
    boolean isTerminated();
    void shutdown();
    <T> Future<T> submit(Callable<T> task);
    Future<?> submit(Runnable task);
    <T> Future<T> submit(Runnable task, T result);
    ...
}
```

# Future, and Callable

```java
package java.util.concurrent;

public interface Future<V> {
   boolean cancel(boolean mayInterruptIfRunning);
   V get();
   boolean isCancelled();
   boolean isDone();
   ...
}
```

```java
package java.util.concurrent;

public interface Callable<V> {
   V call();
}
```

- (Later on we will see this type of interfaces in more detail.)

# Concurrency Library: Atomics

# Atomics

- Non-blocking transaction like shared data structures.
- These modules allow a lock-free (i.e. with non-blocking behavior) handling of shared variables.
- Package: `java.util.concurrent.atomic`
- `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, `AtomicIntegerArray`, `AtomicLongArray`, `AtomicReference`, `AtomicReferenceArray`.
- These modules represent a better and safer alternative to volatile variables.

# Concurrency Library:

# Concurrent data structures

# Blocking Queue



- `java.util.concurrent.BlockingQueue`
- A shared queue in which its normal semantics might be ensured by blocking (add/remove element from a full/empty queue).
- Selected interface:

```
void put(E e)
E take()
```

- Existing implementations: `ArrayBlockingQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`, `LinkedBlockingDeque`, `SynchronousQueue`, `DelayQueue`

# Non-Blocking Queue

- `java.util.concurrent.ConcurrentLinkedQueue`
- Shared lock-free queue.
- Defensive approach to queue operations, issuing immediately a failure if an operation is not possible
- Selected interface:

```
boolean add(E e)
E peek()
E poll()
Iterator<E> iterator()
```

# Concurrent associative array

- `java.util.concurrent.ConcurrentHashMap`
- Shared (mostly) lock-free associative array.
- Defensive approach, issuing immediately a failure if an operation is not possible
- `Class ConcurrentHashMap<K,V>` selected interface:

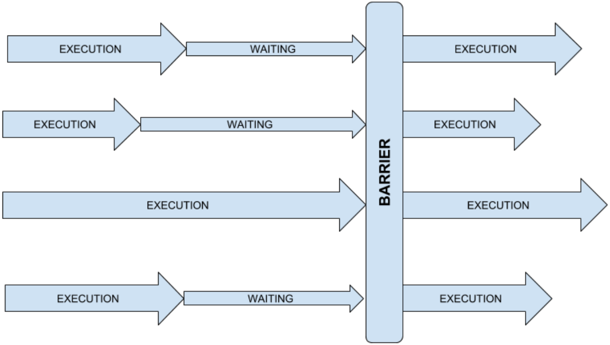
```
V get(Object key)
V put(K key, V value)              // might be a race condition
V putIfAbsent(K key, V value)      // safe
boolean containsKey(Object key)
boolean replace(K key, V oldValue, V newValue)
Enumeration<K> keys()
```

# Concurrency Library: Barriers

# Barriers



- Synchronizing the execution of multiple threads to a single point:



# Barriers (2)

- Class `java.util.concurrent.atomic.CountDownLatch`
- Selected interface:

```
CountDownLatch(int count)
void await()
boolean await(long timeout, TimeUnit unit)
void countDown()
long getCount()
```

- Service `await` waits for `getCount` invocations of `countDown` method.

# Concurrency Library: Fork/join executor

# Fork/join executor

- Optimized executor for fork/join parallel execution (new in Java 7).
- Create the pool:

```
ForkJoinPool forkJoinPool = new ForkJoinPool();
```

- Execute with `invoke` or `invokeAll`
- Possible to define `ForkJoinTask` (`RecursiveAction`, `RecursiveTask`)

# Fork/join executor: example

```
import static java.lang.System.*;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class Fibonacci {
   public static void main(String[] args) {
      int n = Integer.parseInt(args[0]); // error checking missing!
      FibonacciTask fn = new FibonacciTask(n);
      ForkJoinPool forkJoinPool = new ForkJoinPool();
      forkJoinPool.invoke(fn);
      out.println("f("+n+")="+fn.result());
   }
   static class FibonacciTask extends RecursiveAction {
      final int n;
      int result = 1;
      FibonacciTask(int n) {
         this.n = n;
      }
      int result() {
         return result;
      }
      protected void compute() {
         if (n > 2) {
            FibonacciTask t1 = new FibonacciTask(n-1);
            FibonacciTask t2 = new FibonacciTask(n-2);
            invokeAll(t1, t2);
            result = t1.result + t2.result;
         }
      }
   }
}
```

# Concurrency Library: Other useful classes

# Other useful classes





- `ThreadLocal`: variable in which each threads register its own data (a kind of thread local variable accessible through the same object);
- *ReadWriteLock* (to be presented later on);
- *Synchronizers*: `Semaphore`, `FairSemaphore`, `Exchanger`, ...
- ...
- None of these classes uses Design by Contract!
- A library is being developed (since 2010) that solve this serious flaw:

  `http://sweet.ua.pt/mos/pt.ua.concurrent/index.xhtml`