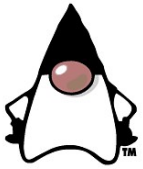
Software Foundations in JAVA (JSF)

Week 6

Advanced thread synchronization

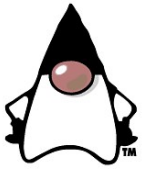
Peter Boots
p.boots@fontys.nl

Erik van der Schriek
e.vanderschriek@fontys.nl



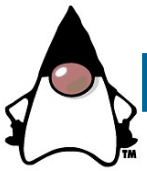
Planning

Week 1	Introduction JSF, Operating systems, Linux
Week 2	Linux shell, interoperabiliteit JAVA
Week 3	Processen en threads
Week 4	Multithreading (MT) en performance
Week 5	MT & concurrency
Week 6	Advanced MT mechanisms
Week 7	Monitors
Week 8	Roundup
Week 9	Exam
Week 10	Repair time



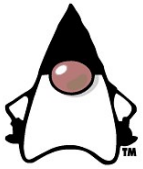
Overview

- java.util.concurrent
  - BlockingQueues
  - Thread Pools
  - Callable/Future
  - CyclicBarrier
  - CountdownLatch
  - java.util.concurrent.atomic
    - AtomicBoolean/AtomicInteger



Package `java.util.concurrent`

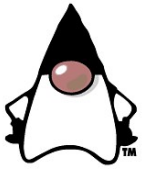
- Contains synchronization interfaces/classes
  - BlockingQueue
  - Thread Pools
  - Callable/Future
  - CyclicBarrier/CountdownLatch
  - …
- Subpackages:
  - `java.util.concurrent.atomic`
    - atomic variables
  - `java.util.concurrent.locks`
    - (reentrant) locks → week 7



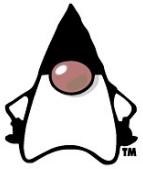
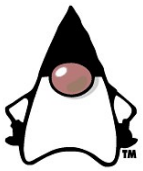
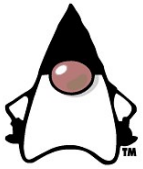
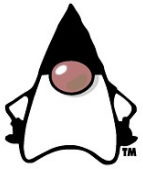
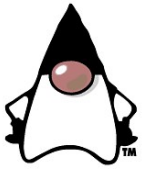
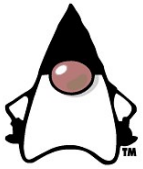
BlockingQueue

- Buffer for multiple Producers and multiple Consumers

- Interface `BlockingQueue<E>`:
  - For putting a value in the queue:
    - `void put(E o)`
      - wait if buffer is full
  - For removing a value from the queue:
    - `E take()`
      - wait if buffer is empty
- More: http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html

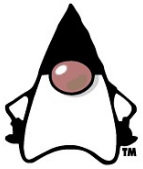


BlockingQueue

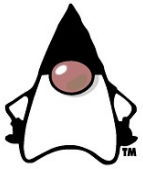
- Typical usage:
  - Creating queue:
    - ```
      BlockingQueue<int> queue =
              new ArrayBlockingQueue<int> ();
      ```
  - Producer thread:
    - ```
      int x;
      while(true) { x=...; queue.put(x); }
      ```
  - Consumer thread:
    - ```
      int y;
      while(true) { y=queue.take()); ...y...}
      ```
      ```



Thread Pool

- Contains number of threads; executing/waiting for tasks

- We can give tasks to a thread pool:
  - task will be executed by one of the threads
  - if all threads are busy, task has to wait
  - thread stays alive when task is finished
  - non-active threads are blocked (take no CPU-time)

- Advantages:
  - Re-using old thread is faster than creating new thread
  - Number of threads is limited to size of pool



Thread Pool

Java provides 5 thread pool architectures:

1. **Single thread executor** - pool of size 1.
   ExecutorService pool = Executors.newSingleThreadExecutor();
2. **Fixed thread executor** - pool of fixed size.
   ExecutorService pool = Executors.newFixedThreadPool(size);
3. **Cached thread pool** - creates new threads when necessary; deletes unused threads after 60 sec
   ExecutorService pool = Executors.newCachedThreadPool();
4. **Scheduled thread pool** – fixed size pool for scheduled or repeated execution (after certain delay)
   ScheduledExecutorService pool = Executors.newScheduledThreadPool(size);
5. **Single scheduled thread pool** – pool of size 1 for scheduled or repeated execution
   ScheduledExecutorService pool =
   Executors.newSingleThreadScheduledExecutor();



Thread Pools in Java (1)

- Creating thread pool with 10 threads
- `ExecutorService pool = Executors.newFixedThreadPool(10);`

- Giving a task to the thread pool
- `pool.execute(task);`
  - `task` must be a `Runnable`
  - `task` will be executed by one of the threads
  - unpredictable when `task` will actually start



Thread Pool and Runnable

- Defining a task:
  - ```
    class Task implements Runnable {
      public void run() {
        System.out.println("test");
    }
    ```
- Offering task to threadpool:
  - ```
    Task t = new Task();
    pool.execute(t);
    ```
- Compare:
  - Same task can also be started directly on a thread:
    - `Thread t = new Thread(t);`



Closing a Thread Pool

- We must be able to shut down a thread pool in a controlled way

- We can do this with method `shutdown`
  - `pool.shutdown();`
    - no new tasks are started anymore
    - when all tasks are finished, thread pool is deleted



Example

- A thread pool running 5 tasks:
  - ```
    public class TPExample {
      public static void main(String[] args) {
        // Create the thread pool
        ExecutorService pool =
                            Executors.newCachedThreadPool();

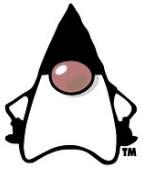
        //run each task using a thread in the pool
        for (int i=0; i<5; i++)
          pool.execute(new Task());

        // Shut down the pool;
        pool.shutdown();
      }
    }
    ```
- Problem: threads can not return a value
    ```



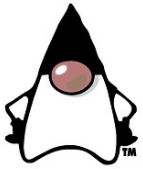
Interface `Callable`

- Threads often have to compute a result
- Problem: method `run` does not return a value
- Solution: use interface `Callable` instead of `Runnable`:
  - ```
    public interface Callable<T> {
       public T call();
    }
    ```
  - Object of class `T` is returned
    ```



Interface Future

- Use a `Future` to retrieve result of a `Callable`:
  - ```
    interface Future<T> {
      T get();
      …
    }
    ```
- Method `get` will wait until the thread is ready, and then return the result
- `Future` has some other methods (not discussed here)
    ```



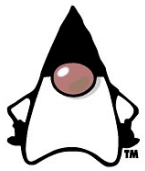
Method submit

- `Callable` and `Future` can be used on a thread pool
- In that case, use method `submit` instead of `execute`:
  - `Future<T> submit(Callable<T> task);`
- `submit` has some other variants with `Runnable`:
  - `Future<?> submit(Runnable task);`
  - `Future<T> submit(Runnable task, T result);`



Method submit

- `Callable` and `Future` can be used on a thread pool
- In that case, use method `submit` instead of `execute`:
  - `Future<T> submit(Callable<T> task);`
- `submit` has some other variants with `Runnable`:
  - `Future<?> submit(Runnable task);`
  - `Future<T> submit(Runnable task, T result);`

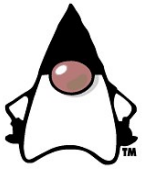


Threadpool /Callable/Future

- ```
  class MyCallable implements Callable<String> {
    public String call() {
      return “test”;
  }
  ```

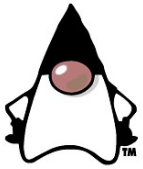
- Suppose `pool` is a thread pool:
  - ```
    MyCallable mc = new MyCallable();
    Future<String> fut = pool.submit(mc);
    String s = fut.get();
    System.out.println(s);
    ```

Wait here until the thread is finished



Shoe repair shop analogy

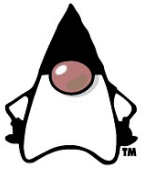
- Suppose you bring old shoes to the shoe repair shop
  - You handin your shoes and ask the repairman to fix them
    - → `submit(Callable)`
  - You get a ticket with which you can collect your shoes later
    - → `Future`
  - You leave, and do other things while your shoes are repaired
  - You go back to the shoe repair shop
    - → `get()`
      - If the shoes are finished, you get them immediately
      - If the shoes are not finished, you wait in the shop until they are done



Synchronizers

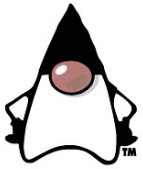
- Java offers a package with synchronisation-mechanisms: `java.util.concurrent`

- In this package we find:
  - `Semaphore` (not discussed here)
  - `CountdownLatch`
  - `CyclicBarrier`
  - …



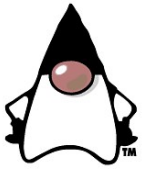
CyclicBarrier

- Let a number of threads wait for each other
  - `CyclicBarrier cb = new CyclicBarrier(5);`
  - Suppose we have 5 threads that do the following:
    - `... // do a certain calculation`
      `cb.await();`
      `... // wrapping up`
  - The first 4 threads will be blocked in `cb.await()`
  - As soon as the 5th arrives, all 5 continue



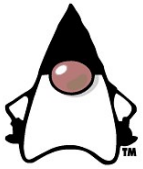
CountDownLatch

- Wait until 0 is reached
  - Is initialized to a certain value
  - Has a `countDown` method that lowers the value
  - Has an `await` method where threads are blocked
  - When the value becomes 0, all blocked threads are woken up



Example `CountDownLatch` (1)

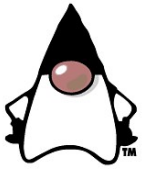
- Suppose
  - you have a thread pool,
  - N tasks for that pool,
  - main thread must wait until all N tasks are finished



Example CountDownLatch (2)

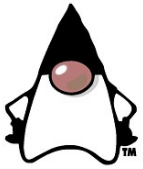
```
class WorkerRunnable implements Runnable {
  private final CountDownLatch doneSignal;

  WorkerRunnable(CountDownLatch doneSignal) {
    this.doneSignal = doneSignal;}

  public void run() {
    doWork();
    doneSignal.countDown();
  }
}
```

```
CountDownLatch doneSignal = new CountDownLatch(N);
Executor e = ...
for (int i = 0; i < N; ++i)
    e.execute(new WorkerRunnable(doneSignal, i));

try { doneSignal.await(); }
catch (InterruptedException ex) {}
```



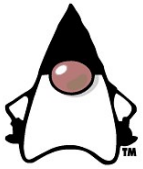
Atomic variables

- Java has some types that provide atomic operations on primitive values:
  - `AtomicInteger`
    - has operations like:
      - `public final int getAndIncrement()`
        - » This method atomically executes `i++;`
      - `public final int incrementAndGet()`
        - » This method atomically executes `++i;`



Atomic variables

- `AtomicBoolean`
  - has operations like:
    - `public final boolean compareAndSet(boolean expect, boolean update)`
      - » This method atomically executes:
      - »
```
if (current_value == expect) {
  current_value = update;
  return true;
}
else
  return false;
```



- Links:
  - http://java.dzone.com/articles/java-concurrency-%E2%80%93-part-7
  - http://tutorials.jenkov.com/java-util-concurrent/executorservice.html
- From Java Core, Volume 1 (ed 8):
  - Chapter 14 “Multithreading”, paragraphs:
    - “Blocking queues” until “Synchronizers”
    - except “Thread Safe Collections”
    - from “Synchronizers”, only CyclicBarriers and CountdownLatch