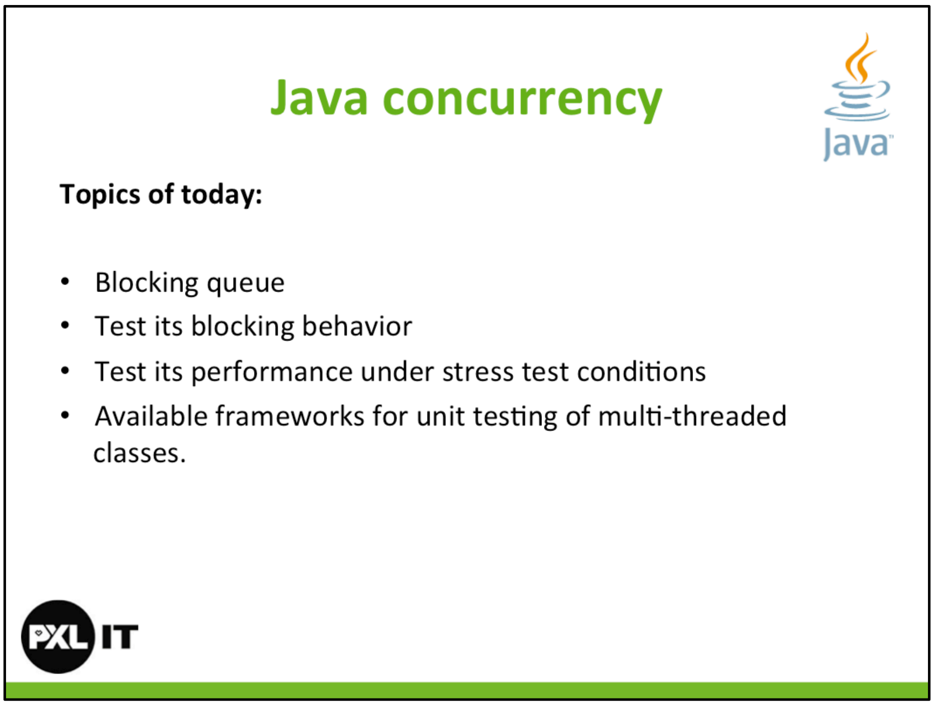
PXL IT

W4: Testing of Multi-threaded applications

.NET / Java

DE HOGESCHOOL
MET HET NETWERK

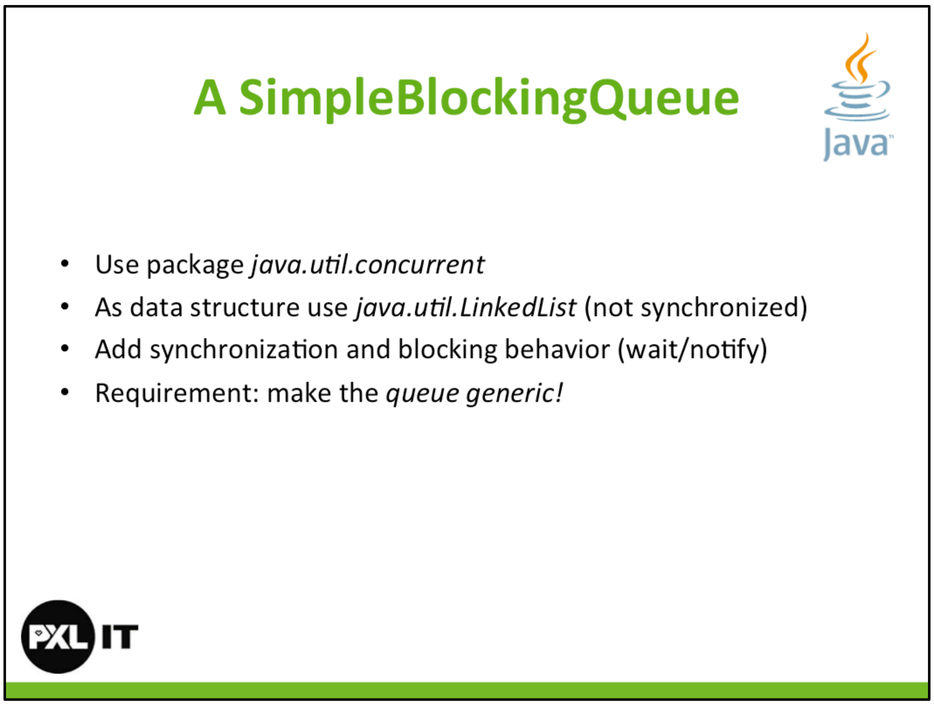
Hogeschool PXL – Dep. PXL-IT – Elfde-Liniestraat 26 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook



Java concurrency

Topics of today:

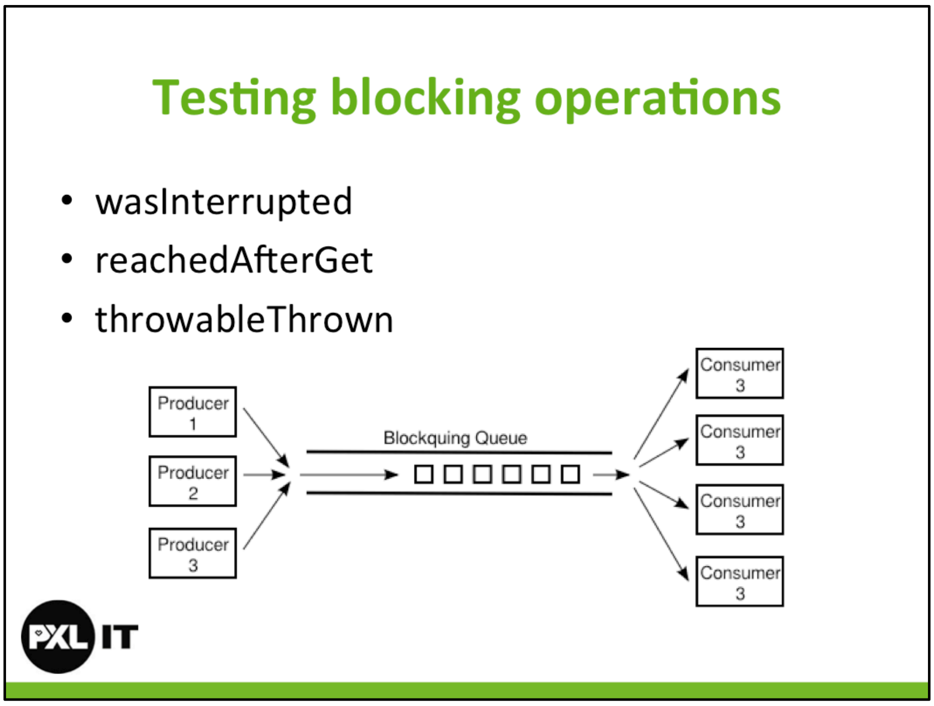
- Blocking queue
- Test its blocking behavior
- Test its performance under stress test conditions
- Available frameworks for unit testing of multi-threaded classes.



A SimpleBlockingQueue

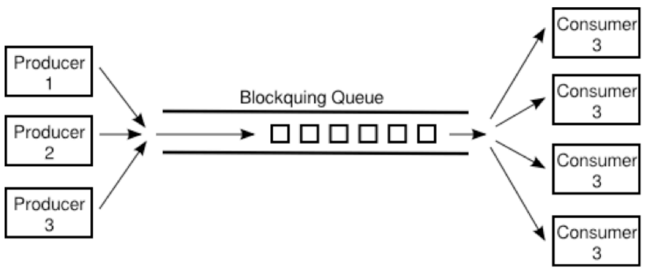
- Use package *java.util.concurrent*
- As data structure use *java.util.LinkedList* (not synchronized)
- Add synchronization and blocking behavior (wait/notify)
- Requirement: make the *queue generic!*

```
public class SimpleBlockingQueue<T> {
        private List<T> queue = new LinkedList<T>();

        public int getSize() {
                synchronized(queue) {
                        return queue.size();
                }
        }

        public void put(T obj) {
                synchronized(queue) {
                        queue.add(obj);
                        queue.notify();
                }
        }
```

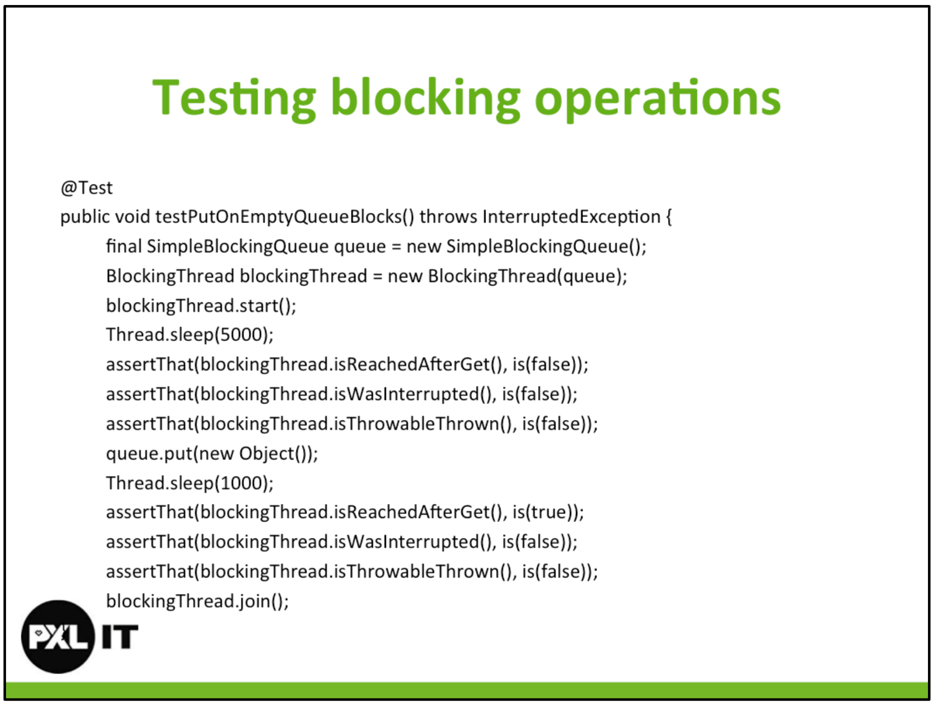


Testing blocking operations

- wasInterrupted
- reachedAfterGet
- throwableThrown

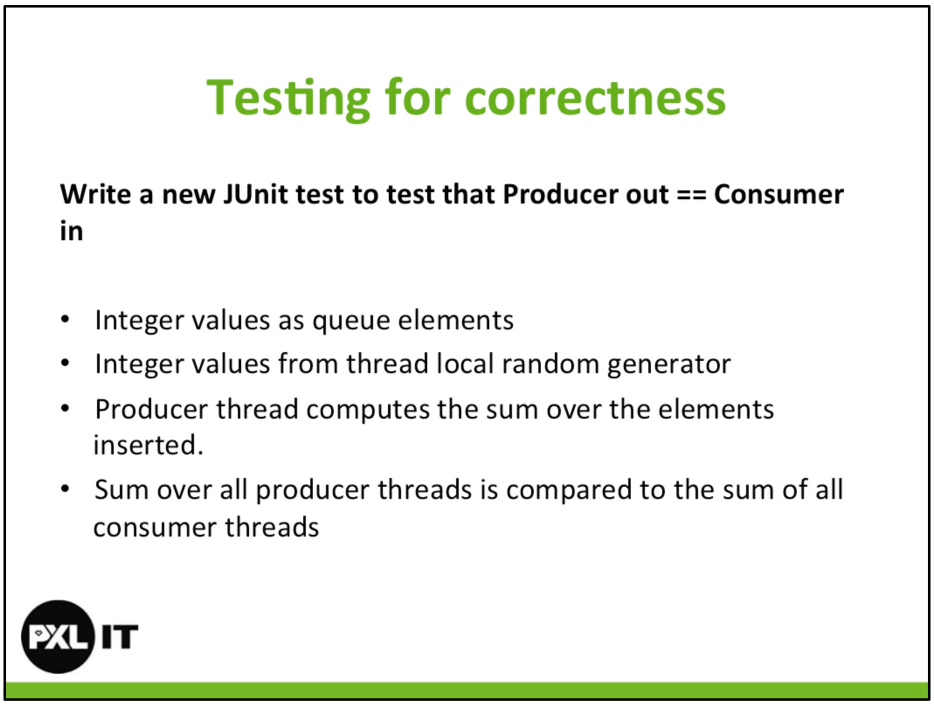


```
private static class BlockingThread extends Thread {
        private SimpleBlockingQueue queue;
        private boolean wasInterrupted = false;
        private boolean reachedAfterGet = false;
        private boolean throwableThrown;

        public BlockingThread(SimpleBlockingQueue queue) {
                this.queue = queue;
        }

        public void run() {
                try {
                        try {
                                queue.get();
                        } catch (InterruptedException e) {
                                wasInterrupted = true;
```



Testing blocking operations

```
@Test
public void testPutOnEmptyQueueBlocks() throws InterruptedException {
        final SimpleBlockingQueue queue = new SimpleBlockingQueue();
        BlockingThread blockingThread = new BlockingThread(queue);
        blockingThread.start();
        Thread.sleep(5000);
        assertThat(blockingThread.isReachedAfterGet(), is(false));
        assertThat(blockingThread.isWasInterrupted(), is(false));
        assertThat(blockingThread.isThrowableThrown(), is(false));
        queue.put(new Object());
        Thread.sleep(1000);
        assertThat(blockingThread.isReachedAfterGet(), is(true));
        assertThat(blockingThread.isWasInterrupted(), is(false));
        assertThat(blockingThread.isThrowableThrown(), is(false));
        blockingThread.join();
```

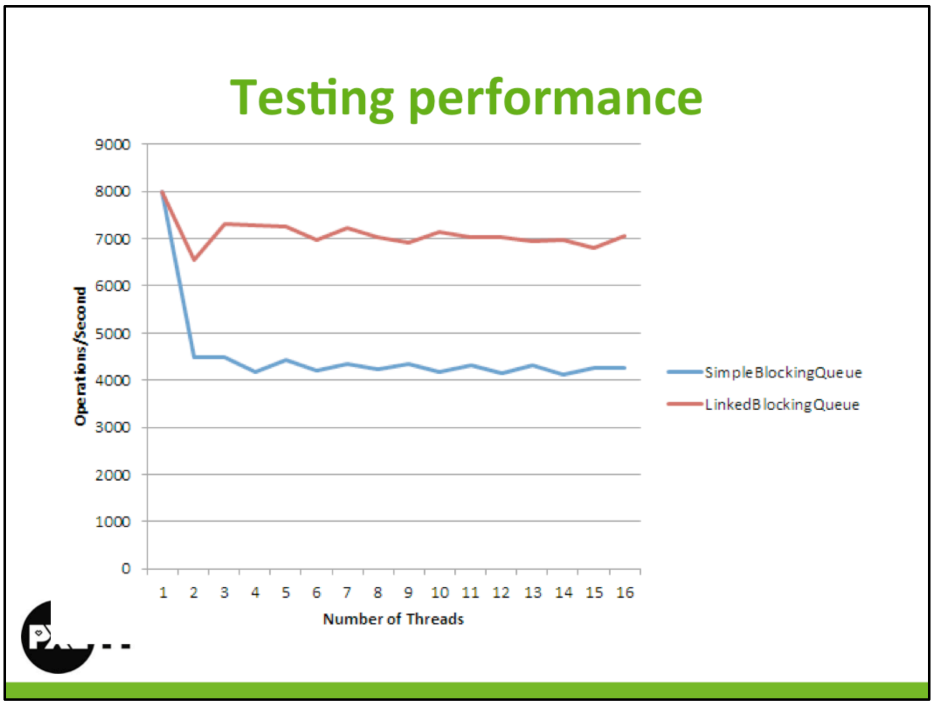


Testing for correctness

Write a new JUnit test to test that Producer out == Consumer in

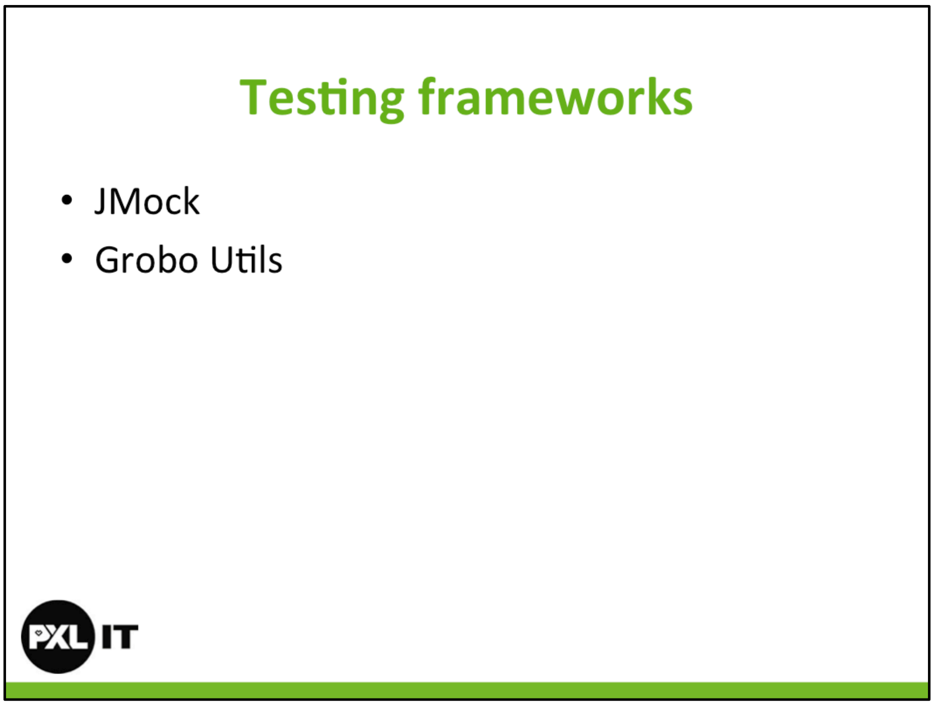
- Integer values as queue elements
- Integer values from thread local random generator
- Producer thread computes the sum over the elements inserted.
- Sum over all producer threads is compared to the sum of all consumer threads

```
@Test
public void testParallelInsertionAndConsumption() throws InterruptedException,
ExecutionException {
                final SimpleBlockingQueue<Integer> queue = new SimpleBlockingQueue<Integer>();
                ExecutorService threadPool = Executors.newFixedThreadPool(NUM_THREADS);
                final CountDownLatch latch = new CountDownLatch(NUM_THREADS);
                List<Future<Integer>> futuresPut = new ArrayList<Future<Integer>>();
                for (int i = 0; i < 3; i++) {
                                Future<Integer> submit = threadPool.submit(new Callable<Integer>() {
                                                public Integer call() {
                                                                int sum = 0;
                                                                for (int i = 0; i < 1000; i++) {
                                                                                int nextInt =
ThreadLocalRandom.current().nextInt(100);

                queue.put(nextInt);
                                                                                sum +=
nextInt;
```



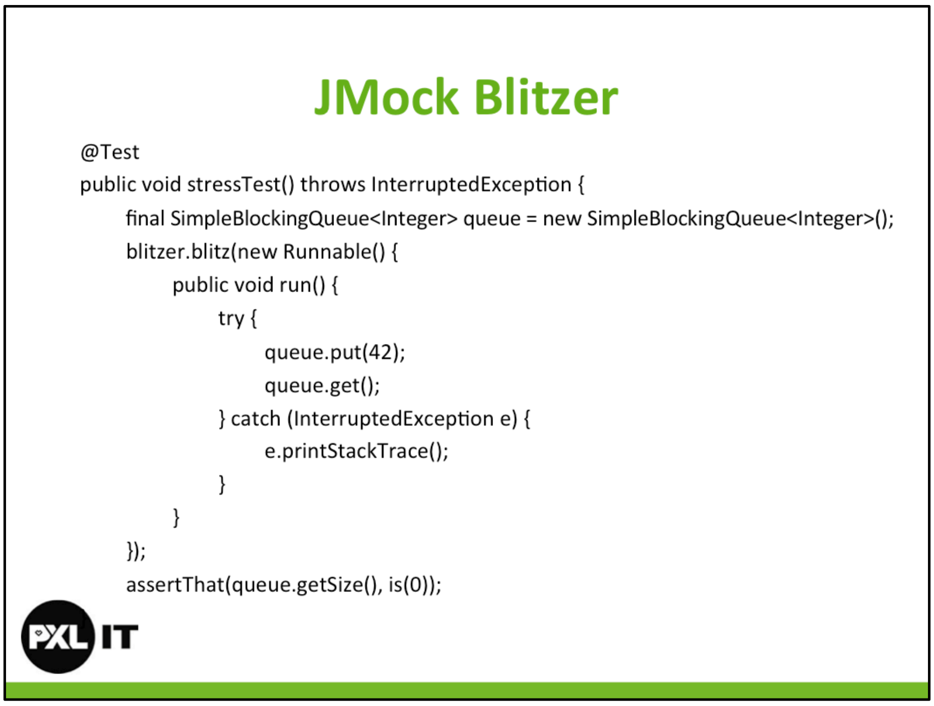
Testing performance

```
@Test
public void testPerformance() throws InterruptedException {
                for (int numThreads = 1; numThreads < THREADS_MAX; numThreads++) {
                                long startMillis = System.currentTimeMillis();
                                final SimpleBlockingQueue<Integer> queue = new
SimpleBlockingQueue<Integer>();
                                ExecutorService threadPool =
Executors.newFixedThreadPool(numThreads);
                                for (int i = 0; i < numThreads; i++) {
                                                threadPool.submit(new Runnable() {
                                                                public void run() {
                                                                                for (long i = 0;
i < ITERATIONS; i++) {

                int nextInt = ThreadLocalRandom.current().nextInt(100);

                try {
```



Testing frameworks

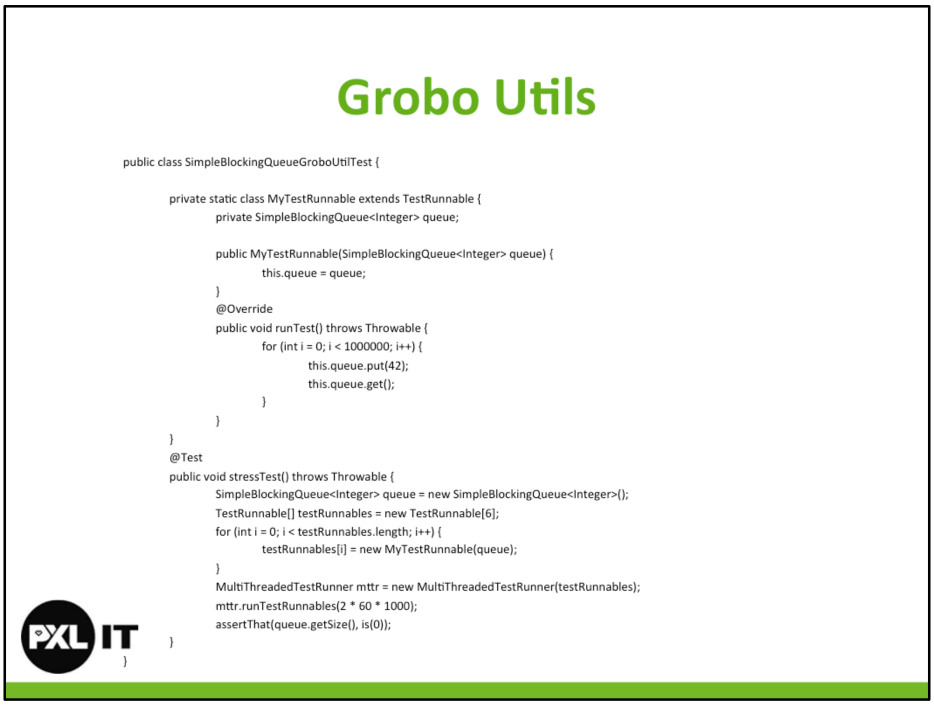
- JMock
- Grobo Utils



JMock Blitzer

```
@Test
public void stressTest() throws InterruptedException {
    final SimpleBlockingQueue<Integer> queue = new SimpleBlockingQueue<Integer>();
    blitzer.blitz(new Runnable() {
        public void run() {
            try {
                queue.put(42);
                queue.get();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    assertThat(queue.getSize(), is(0));
```

For stress testing purposes the mocking framework JMock provides the class Blitzer. This class implements functionality similar to what we have done in section "Testing for correctness" as it internally set ups a ThreadPool to which tasks are submitted that execute some specific action. You provide the number of tasks/actions to perform as well as the number of threads to the constructor.



Grobo Utils

```
public class SimpleBlockingQueueGroboUtilTest {

	private static class MyTestRunnable extends TestRunnable {
		private SimpleBlockingQueue<Integer> queue;

		public MyTestRunnable(SimpleBlockingQueue<Integer> queue) {
			this.queue = queue;
		}
		@Override
		public void runTest() throws Throwable {
			for (int i = 0; i < 1000000; i++) {
				this.queue.put(42);
				this.queue.get();
			}
		}
	}
	@Test
	public void stressTest() throws Throwable {
		SimpleBlockingQueue<Integer> queue = new SimpleBlockingQueue<Integer>();
		TestRunnable[] testRunnables = new TestRunnable[6];
		for (int i = 0; i < testRunnables.length; i++) {
			testRunnables[i] = new MyTestRunnable(queue);
		}
		MultiThreadedTestRunner mttr = new MultiThreadedTestRunner(testRunnables);
		mttr.runTestRunnables(2 * 60 * 1000);
		assertThat(queue.getSize(), is(0));
	}
}
```

Similar to the previous example we have the class MultiThreadedTestRunner that internally constructs a thread pool and executes a given number of Runnable implementations as separate threads. The Runnable instances have to implement a special interface called TestRunnable. It is worth mentioning that its only method runTest() throws an exception. This way, exceptions thrown within the threads have an influence on the test result. This is not the case when we use a normal ExecutorService. Here the tasks have to implement Runnable and its only method run() does not throw any exception. Exceptions thrown within these tasks are swallowed and don't break the test.

After having constructed the MultiThreadedTestRunner we can call its runTestRunnables() method and provide the number of milliseconds it should wait until the test should fail. Finally the assertThat() call verifies that the queue is empty again, as all test threads have removed the previously added element.