Programming in Java – Concurrency

Paolo Vercesi

Technical Program Manager

Starting a Thread

```
public static void main(String[] args) throws Exception {
    var thread = new Thread(new Runnable() {
        @Override
        public void run() {
            while (true) {
                System.out.println("Running");
                try {
                    Thread.sleep(1000);
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        }
    });
    thread.start();
    System.out.println("End of main");
}
```

```
End of main
Running
Running
Running
…
```

The Java Virtual Machine allows an application to have multiple threads of execution running concurrently, regardless the number of processors.

The Thread class is used to create and start new threads of execution.

The Thread class

```
public class Thread implements Runnable {

    public Thread()

    public Thread(Runnable target)

    public Thread(String name)

    public Thread(Runnable target, String name)

    …
}
```

```
public interface Runnable {

    public abstract void run();
}
```

The Runnable object that a thread runs can be either the Thread itself or the target thread passed to the constructor.

The Thread class implements an empty run() method.

The two constructors without the Runnable target should be used by subclasses only.

Thread instance methods

```
public void start()
@Override
public void run()
public void interrupt()
public boolean isInterrupted()
public final boolean isAlive()
public final void setName(String name)
public final String getName()
public final void join(final long millis)
public final void join(long millis, int nanos) throws InterruptedException
public final void join() throws InterruptedException
public final void setDaemon(boolean on)
public final boolean isDaemon()
```

Thread static methods

```
public static Thread currentThread()
public static void yield()
public static void sleep(long millis) throws InterruptedException
public static void sleep(long millis, int nanos) throws InterruptedException
public static boolean interrupted()
public static void dumpStack()
```

Waiting for a thread to finish

```
public static void main(String[] args) throws Exception {
    var thread = new Thread(new Runnable() {
        @Override
        public void run() {
            for (int i = 0; i < 5; i++) {
                System.out.println("Running");
                try {
                    Thread.sleep(1000);
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        }
    });
    thread.start();
    System.out.println("Start waiting for the thread to finish");
    thread.join();
    System.out.println("End of main");
}
```

```
Start waiting for the thread
to finish
Running
Running
Running
Running
Running
Running
End of main
```

Concurrent programming

In concurrent programming we must take special care of two different problems

1. *protecting shared resources from concurrent access*
2. *synchronization of concurrent threads*

Java offers solutions at different levels to solve these two problems of synchronization

Id generator 1/4

```
public class IdGenerator {

    private int id = 0;

    public int nextId() {
        id = id + 1;
        return id;
    }

}
```

```
public class IdConsumer implements Runnable {

    private final IdGenerator idGenerator;

    public IdConsumer(IdGenerator idGenerator) {
        this.idGenerator = idGenerator;
    }

    @Override
    public void run() {
        int id;
        for (int i = 0; i < 25; i++) {
            id = idGenerator.nextId();
            System.out.println(id);
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Id generator 2/4

```
public static void main(String[] args) {
    IdGenerator idGenerator = new IdGenerator();
    Thread thread1 = new Thread(new IdConsumer(idGenerator));
    Thread thread2 = new Thread(new IdConsumer(idGenerator));
    Thread thread3 = new Thread(new IdConsumer(idGenerator));
    Thread thread4 = new Thread(new IdConsumer(idGenerator));
    thread1.start();
    thread2.start();
    thread3.start();
    thread4.start();
}
```

```
56
56
58
60
59
62
61
63
63
65
67
66
64
69
68
68
68
70
72
71
70
73
```

Several ids are duplicate

The last id is expected to be 100

Id generator 3/4

```
public class IdGenerator {

    private int id = 0;

    public synchronized int nextId() {
        id = id + 1;
        return id;
    }
}
```

To avoid races, only one thread at time is allowed to enter the nextId() method

In Java synchronization is achieved using monitors. A monitor is an "object" that is used as a mutually exclusive lock.
At a given time, only one thread can own a monitor. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor.

While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance must wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

Id generator 4/4

```
public class IdGenerator {

    private final Object lock = new Object();
    private int id = 0;

    public int nextId() {
        synchronized (lock) {
            id = id + 1;
            return id;
        }
    }
}
```

Instead of synchronizing a whole method, we can use a synchronized block to synchronize just one block of code.

This fine-grained synchronization can be used to avoid to synchronize the whole IdGenerator object

Producer & Consumer 1/4

```
public class Producer implements Runnable {

    private final Buffer<Integer> buffer;

    public Producer(Buffer<Integer> buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        int counter = 0;
        while (counter < 10) {
            buffer.set(counter);
            counter++;
            Thread.sleep(1)
        }
    }
}
```

```
public class Consumer implements Runnable {

    private final Buffer<Integer> buffer;

    public Consumer(Buffer<Integer> buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        Integer object = null;
        while (!Objects.equals(object, 9)) {
            object = buffer.get();
            Thread.sleep(1);
        }
    }
}
```

Producer & Consumer 2/4

```
public class Buffer<T> {
    private volatile T object;

    public synchronized T get() {
        T result = object;
        object = null;
        return object;
    }

    public synchronized void set(T object) {
        this.object = object;
    }
}
```

```
public static void main(String[] args) {
    var buffer = new Buffer<Integer>();
    new Thread(new Producer(buffer)).start();
    new Thread(new Consumer(buffer)).start();
}
```

```
Set: 0
Got: 0
Set: 1
Got: 1
Got: null
Set: 2
Got: 2
Set: 3
Set: 4
Got: 4
Set: 5
Got: 5
Got: null
Set: 6
Set: 7
Got: 7
Set: 8
Got: 8
Set: 9
Got: 9
```

Buffer is Thread-safe

But Producer and consumer are not synchronized.

Producer & Consumer 3/4

When Consumer calls get () the implementation of get () should wait until the object is not null (or set by the Producer)

Can we avoid busy-waiting?

```
public class Buffer<T> {
    private volatile T object;

    public synchronized T get() {
        while (object == null) {
            // wait…
        }
        T result = object;
        object = null;
        return object;
    }

    public synchronized void set(T object) {
        this.object = object;
    }
}
```

```
public class Buffer<T> {
    private volatile T object;

    public synchronized T get() {
        while (object == null) {
            wait();
        }
        T result = object;
        object = null;
        return object;
    }

    public synchronized void set(T object) {
        this.object = object;
        notify();
    }
}
```

Producer & Consumer 4/4

When Producer calls set () the implementation of set () should wait until the object is null (or got by the Consumer)

```
public class SynchronizedBuffer<T> {
    private volatile T object;

    public synchronized T get() {
        while (object == null) {
            wait();
        }
        T result = object;
        object = null;
        notify();
        return object;
    }

    public synchronized void set(T object) {
        while (object != null) {
            wait();
        }
        this.object = object;
        notify();
    }
}
```

Monitors 1/2

```
synchronized (monitor) {
    while (condition-for-wait) {
        try {
            monitor.wait();
        } catch (InterruptedException e) {
            …
        }
    }
    // do processing

    monitor.notify();
}
```

wait(),
notify() and
notifyAll()
are final methods defined in Object.

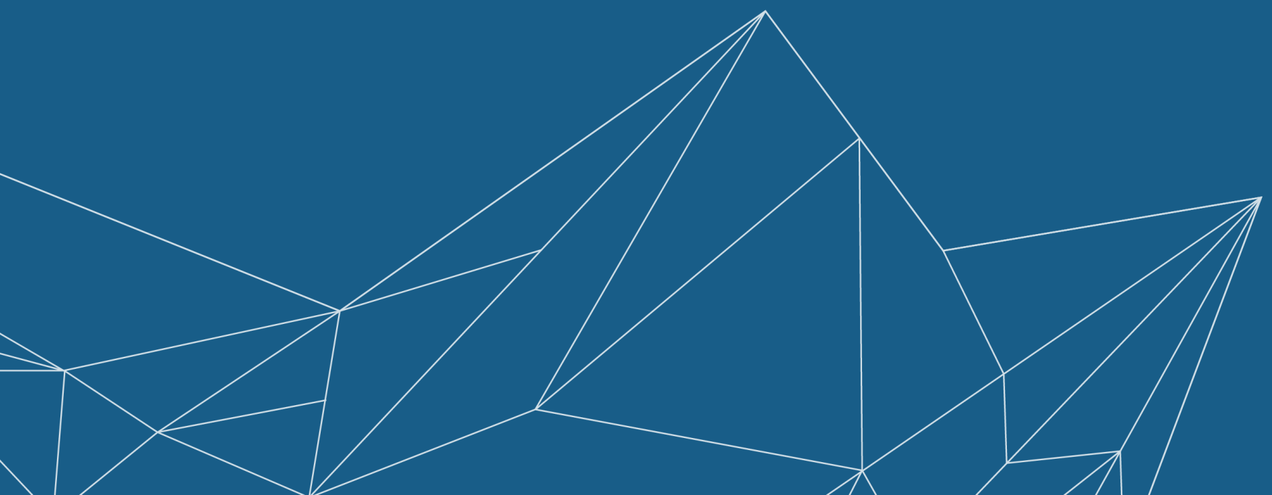
Monitors 2/2

```
synchronized (monitor) {
    while (condition-for-wait) {
        try {
            monitor.wait();
        } catch (InterruptedException e) {
            …
        }
    }
    try {
        // do processing
    } finally {
        monitor.notify();
    }
}
```

Assignment

Allow the SynchronizedBuffer to accept null values.

Concurrent API



java.util.concurrent

- Synchronizers
- Executors
- Concurrent collections
- Locks
- Atomics

Synchronizers

- Semaphore
- CountDownLatch
- CyclicBarrier
- Exchanger
- Phaser (a CyclicBarrier with multiple phases)

Semaphore

```
public Semaphore(int permits)
public Semaphore(int permits, boolean fair)

public void acquire() throws InterruptedException
public void acquire(int permits) throws InterruptedException
public void acquireUninterruptibly()
public void acquireUninterruptibly(int permits)

public boolean tryAcquire()
public boolean tryAcquire(long timeout, TimeUnit unit)
public boolean tryAcquire(int permits)
public boolean tryAcquire(int permits, long timeout, TimeUnit unit)

public void release()
public void release(int permits)

public int availablePermits()
public int drainPermits()
public boolean isFair()
public final boolean hasQueuedThreads()
public final int getQueueLength()
```

A semaphore maintains a set of permits and it is used to restrict the number of threads than can access some resource.

Each acquire() blocks if necessary until a permit is available, and then takes it.

Each release() adds a permit, potentially releasing a blocking acquirer.

The release() can be invoked by a different thread to allow Deadlock resolution.

Consistency between acquire and release from the same thread is not required.

Semaphore “trivial” example

```
public class SemaphoreIdGenerator {

    private int id = 0;
    private Semaphore semaphore = new Semaphore(1);

    public int nextId() throws InterruptedException {
        semaphore.acquire();
        try {
            id = id + 1;
            return id;
        } finally {
            semaphore.release();
        }
    }
}
```

CountDownLatch

```
public CountDownLatch(int count)

public void await() throws InterruptedException

public boolean await(long timeout, TimeUnit unit)

public void countDown()

public long getCount()
```

> *A CountDownLatch is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.*

CountDownLatch example

```
class Worker implements Runnable {

    private final CountDownLatch latch;
    private final List<String> data;

    private Worker(CountDownLatch latch, List<String> data) {
        this.latch = latch;
        this.data = data;
    }

    @Override
    public void run() {
        //process data
        latch.countDown();
    }
}
```

```
public static void main(String[] args) throws InterruptedException {
    int numberOfWorkers = 4;

    var latch = new CountDownLatch(numberOfWorkers);
    List<String> data = List.of("Some", "data", "to", "process");
    int size = data.size() / numberOfWorkers;

    for (int i = 0; i < 4; i++) {
        var workerData = data.subList(i * size, (i + 1) * size);
        new Thread(new Worker(latch, workerData)).start();
    }

    latch.await();

    System.out.println("All workers finished");
}
```

CyclicBarrier

```
public CyclicBarrier(int parties, Runnable barrierAction)
public CyclicBarrier(int parties)

public int getParties()

public int await()throws InterruptedException, BrokenBarrierException
public int await(long timeout, TimeUnit unit)
        throws InterruptedException, BrokenBarrierException, TimeoutException

public boolean isBroken()
public void reset()
public int getNumberWaiting()
```

A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.

CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called cyclic because it can be re-used after the waiting threads are released.

Exchanger

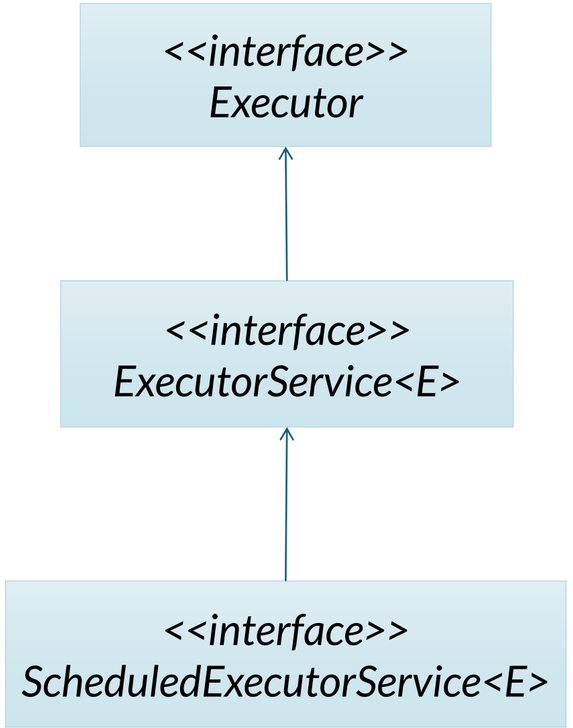
```
public class Exchanger<V> {
    public Exchanger() {
    public V exchange(V x) throws InterruptedException {
    public V exchange(V x, long timeout, TimeUnit unit)
}
```

A synchronization point at which threads can pair and swap elements within pairs. Each thread presents some object on entry to the exchange method, matches with a partner thread, and receives its partner's object on return.

Executor

An executor initiates and controls the execution of threads

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



ExecutorService

```
public interface ExecutorService extends Executor {
    <T> Future<T> submit(Callable<T> task)
    <T> Future<T> submit(Runnable task, T result)
    Future<?> submit(Runnable task)
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) throws InterruptedException;
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)
            throws InterruptedException;
    <T> T invokeAny(Collection<? extends Callable<T>> tasks) throws InterruptedException, ExecutionException;
    <T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)
            throws InterruptedException, ExecutionException, TimeoutException;
    boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException
    void shutdown()
    boolean isShutdown()
    boolean isTerminated()
}
```

Callable & Future

```
public interface Callable<V> {

    V call() throws Exception;

}
```

An «improved» Runnable, returns a value and can throw checked exceptions

The result of an asynchronous computation.

```
public interface Future<V> {

    V get() throws InterruptedException, ExecutionException;

    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;

    boolean cancel(boolean mayInterruptIfRunning);

    boolean isCancelled();

    boolean isDone();
}
```

ScheduledExecutorService

```
public interface ScheduledExecutorService extends ExecutorService {

    public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);

    public <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit);

    public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                                  long initialDelay,
                                                  long period,
                                                  TimeUnit unit);

    public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                                     long initialDelay,
                                                     long delay,
                                                     TimeUnit unit);

}
```

ScheduledFuture

```
public interface Delayed extends Comparable<Delayed> {

    /**
     * Returns the remaining delay associated with this object, in the
     * given time unit.
     */
    long getDelay(TimeUnit unit);
}
```

```
public interface ScheduledFuture<V> extends Delayed, Future<V> {

}
```

Creating ExecutorServices with the Executors class

```
static ExecutorService newCachedThreadPool()
static ExecutorService newFixedThreadPool(int nThreads)
static ExecutorService newSingleThreadExecutor()
static ScheduledExecutorService newSingleThreadScheduledExecutor()
static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
static ExecutorService newWorkStealingPool()
static ExecutorService newWorkStealingPool(int parallelism)
static ExecutorService unconfigurableExecutorService(ExecutorService executor)
static ScheduledExecutorService unconfigurableScheduledExecutorService(ScheduledExecutorService executor)
```

Concurrent collections

- *ArrayBlockingQueue*
- *ConcurrentHashMap*
- *ConcurrentLinkedQueue ConcurrentLinkedDeque*
- *ConcurrentSkipListMap*
- *ConcurrentSkipListSet*
- *CopyOnWriteArrayList*
- *CopyOnWriteArraySet*
- *DelayQueue*
- *LinkedBlockingQueue LinkedBlockingDeque*
- *LinkedTransferQueue PriorityBlockingQueue*
- *SynchronousQueue*

Provide thread-safety and various forms of synchronization

LinkedBlockingQueue

```
public class SynchronizedBuffer<T> {
    private volatile T object;

    public synchronized T get() {
        while (object == null) {
            wait();
        }
        T result = object;
        object = null;
        notify();
        return object;
    }

    public synchronized void set(T object) {
        while (object != null) {
            wait();
        }
        this.object = object;
        notify();
    }
}
```

It's already implemented by the LinkedBlockingQueue or the ArrayBlockingQueue

Locks 1/3

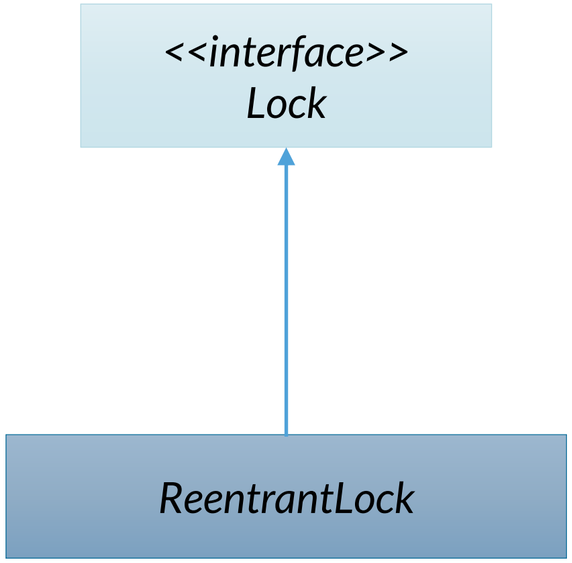
```
public interface Lock {

    void lock();

    void lockInterruptibly() throws InterruptedException;

    boolean tryLock();

    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;

    void unlock();

    Condition newCondition();
}
```

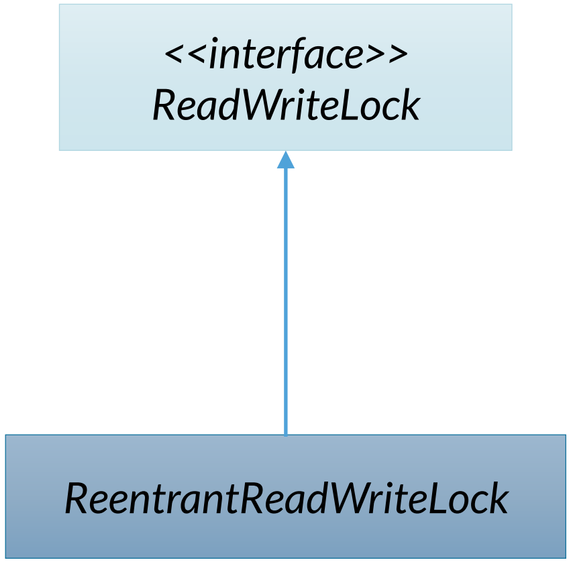
Locks are like synchronized blocks, but the added flexibility that you can lock and unlock from different scopes or methods.

Conditions provide the same functionalities of wait and notify, with the difference that you can have multiple wait sets for the same object.

```
public interface Condition {

   void await() throws InterruptedException;

   void awaitUninterruptibly();

   long awaitNanos(long nanosTimeout) throws InterruptedException;

   boolean await(long time, TimeUnit unit) throws InterruptedException;

   boolean awaitUntil(Date deadline) throws InterruptedException;

   void signal();

   void signalAll();
}
```

Locks 2/3





Locks 3/3

```
class BoundedBuffer<E> {
  private final Lock lock = new ReentrantLock();
  private final Condition isNull  = lock.newCondition();
  private final Condition valueSet = lock.newCondition();
  private E item;

  public void set(E x) throws InterruptedException {
    lock.lock();
    try {
      while (item != null)
        isNull.await();
      item = x;
      valueSet.signal();
    } finally {
      lock.unlock();
    }
  }
```

```
  public E get() throws InterruptedException {
    lock.lock();
    try {
      while (item == null)
        valueSet.await();
      E x = item;
      item = null;
      isNull.signal();
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```

Atomics

AtomicBoolean	A boolean value that may be updated atomically.
AtomicInteger	An int value that may be updated atomically.
AtomicIntegerArray	An int array in which elements may be updated atomically.
AtomicIntegerFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.
AtomicLong	A long value that may be updated atomically.
AtomicLongArray	A long array in which elements may be updated atomically.
AtomicLongFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes.
AtomicMarkableReference<V>	An AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may be updated atomically.
AtomicReferenceFieldUpdater<T,V>	A reflection-based utility that enables atomic updates to designated volatile reference fields of designated classes.
AtomicStampedReference<V>	An AtomicStampedReference maintains an object reference along with an integer "stamp", that can be updated atomically.
DoubleAccumulator	One or more variables that together maintain a running double value updated using a supplied function.
DoubleAdder	One or more variables that together maintain an initially zero double sum.
LongAccumulator	One or more variables that together maintain a running long value updated using a supplied function.
LongAdder	One or more variables that together maintain an initially zero long sum.

AtomicInteger

```
public class AtomicInteger extends Number {
    public AtomicInteger(int initialValue)
    public AtomicInteger()
    public final int get()
    public final void set(int newValue)
    public final int getAndSet(int newValue)
    public final boolean compareAndSet(int expectedValue, int newValue)
    public final int getAndIncrement()
    public final int getAndDecrement()
    public final int getAndAdd(int delta)
    public final int incrementAndGet()
    public final int decrementAndGet()
    public final int addAndGet(int delta)
    public final int getAndUpdate(IntUnaryOperator updateFunction)
    public final int updateAndGet(IntUnaryOperator updateFunction)
    public final int getAndAccumulate(int x, IntBinaryOperator accumulatorFunction)
    public final int accumulateAndGet(int x, IntBinaryOperator accumulatorFunction)
    …
}
```

AtomicInteger example

```
public class IdGenerator {

    private int id = 0;

    public synchronized int nextId() {
        id = id + 1;
        return id;
    }
}
```

```
public class IdGenerator {

    private AtomicInteger id = new AtomicInteger();

    public int nextId() {
       return id.incrementAndGet();
    }
}
```

Thank you!

esteco.com