Programming in Java – Basics of concurrency

Paolo Vercesi

Technical Program Manager

© 2019 ESTECO SpA

Concurrency utilities

© 2019 ESTECO SpA

Multithreaded programming

© 2019 ESTECO SpA

Concurrency

"more than one task running simultaneously on a system"

Writing correct programs is hard;
writing correct concurrent programs is harder.

© 2019 ESTECO SpA

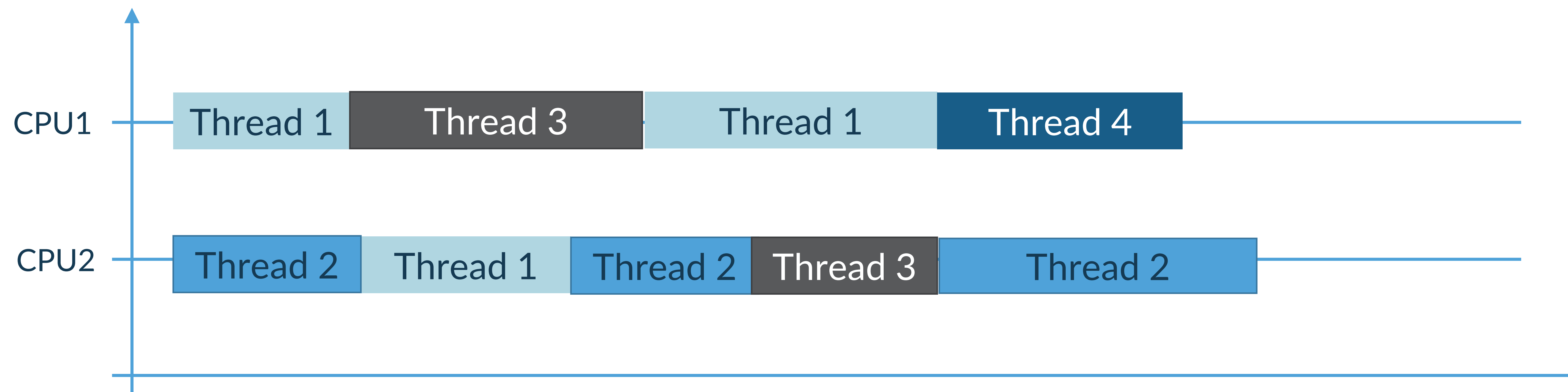
Processes and threads

A process is an executing program

A thread of execution, or simply a thread, is the smallest unit of execution to which a scheduler allocates a CPU

A Java process contains several concurrent threads executing in a shared memory environment

Java threads are scheduled (started, interrupted, and resumed) by the scheduler of the underlying operating system



The execution of a thread is assigned to a CPU until the scheduler decides to interrupt the thread execution to schedule another thread

© 2019 ESTECO SpA

Threads

A thread consists of a stack,
a program counter, and an id

In Java, threads are instances of the
`java.lang.Thread` class

```
"RMI TCP Accept-0" #24 daemon prio=6 os_prio=0 cpu=0.00ms elapsed=325.06s tid=0x000002116749700 nid=0x253c runnable
[0x0000004e2e0fe000]
   java.lang.Thread.State: RUNNABLE
        at java.net.PlainSocketImpl.accept0(java.base@11.0.15/Native Method)
        at java.net.PlainSocketImpl.socketAccept(java.base@11.0.15/PlainSocketImpl.java:159)
        at java.net.AbstractPlainSocketImpl.accept(java.base@11.0.15/AbstractPlainSocketImpl.java:474)
        at java.net.ServerSocket.implAccept(java.base@11.0.15/ServerSocket.java:565)
        at java.net.ServerSocket.accept(java.base@11.0.15/ServerSocket.java:533)
        at it.esteco.rmi.ssl.SslRMIServerSocketFactory$1.accept(SslRMIServerSocketFactory.java:24)
        at sun.rmi.transport.tcp.TCPTransport$AcceptLoop.executeAcceptLoop(java.rmi@11.0.15/TCPTransport.java:394)
        at sun.rmi.transport.tcp.TCPTransport$AcceptLoop.run(java.rmi@11.0.15/TCPTransport.java:366)
        at java.lang.Thread.run(java.base@11.0.15/Thread.java:829)
```

A thread is executing the code of a single method, namely the current method for that thread and the program counter contains the address of the instruction currently being executed

© 2019 ESTECO SpA

Starting a Thread

```java
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

© 2019 ESTECO SpA

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

To use a thread, you must either pass a Runnable in the constructor or to override the run() method

© 2019 ESTECO SpA

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()
```

A thread does not return a value nor throw any exception

© 2019 ESTECO SpA

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()
```

© 2019 ESTECO SpA

Waiting for a thread to finish

```java
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
End of main
```

© 2019 ESTECO SpA

Issues in concurrent programming

Data access synchronization

- Critical sections
- Mutual exclusion

Process synchronization

- Asynchronous programming
- Wait/notify

© 2019 ESTECO SpA

Evolution of concurrent programming in Java

Level of abstraction

parallel programming

synchronization and messages

```
synchronized
wait
notify
```

Threads and monitors

Java (1996)

Synchronizers
Executors
Concurrent collections
Locks
Atomics

Concurrency utilities

Java 5 (2004)

ForkJoinPool
ForkJoinTask

Fork/Join Framework

Java 7 (2011)

Stream

Parallel streams

Java 8 (2014)

Flow

Reactive streams

Java 9 (2017)

Virtual threads

Java ? (????)

© 2019 ESTECO SpA

Concurrency utilities

© 2019 ESTECO SpA

Concurrency utilities

Part of the java.base module

- package java.concurrency
  - synchronizers
  - executors
  - concurrent collections
- package java.concurrency.atomic
- package java.concurrency.locks

© 2019 ESTECO SpA

Id generator 1/2

```
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();
            }
        }
    }
}
```

© 2019 ESTECO SpA

Id generator 2/2

```
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
```

© 2019 ESTECO SpA

Critical sections

```java
public class IdGenerator {

    private int id = 0;

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

sum 1 and id
assign the result to id

Critical section

A critical section is a piece of code that can be executed by only one thread at a time

Unprotected critical sections make your code non thread-safe

Many parts of Java are not thread-safe, as stated in their documentation, e.g., all the mutable collections

In general, in multi-threading environments we serialize the access to critical sections

© 2019 ESTECO SpA

Synchronized method

```
public class SynchronizedIdGenerator implements IdGenerator {

    private int id = 0;

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

While a thread is inside a synchronized method, all other threads cannot enter the same method or any other synchronized method of the same instance, so they go into a wait state

When a thread returns from the synchronized method, the control of the instance is given to the next waiting thread

© 2019 ESTECO SpA

Synchronized block

```
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

© 2019 ESTECO SpA

Synchronization with ReentrantLock

```java
import java.util.concurrent.locks.ReentrantLock;

public class LockIdGenerator implements IdGenerator {

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

    @Override
    public int nextId() {
        lock.lock();
        id = id + 1;
        lock.unlock();
        return id;
    }
}
```

© 2019 ESTECO SpA

The Lock interface

```
public interface Lock {

    void lock();

    void lockInterruptibly() throws InterruptedException;

    boolean tryLock();

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

    void unlock();

    Condition newCondition();
}
```

The thread acquires the lock or waits until the lock is available

If the interrupted flag is set it throws an InterruptedException

Try to acquire the lock otherwise returns false

Try to acquire the lock for the specified amount of time

Release the lock

© 2019 ESTECO SpA

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

© 2019 ESTECO SpA

Semaphore “trivial” example

```
public class SemaphoreIdGenerator {

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

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

© 2019 ESTECO SpA

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.

© 2019 ESTECO SpA

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");
}
```

© 2019 ESTECO SpA

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.

© 2019 ESTECO SpA

Exchanger

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

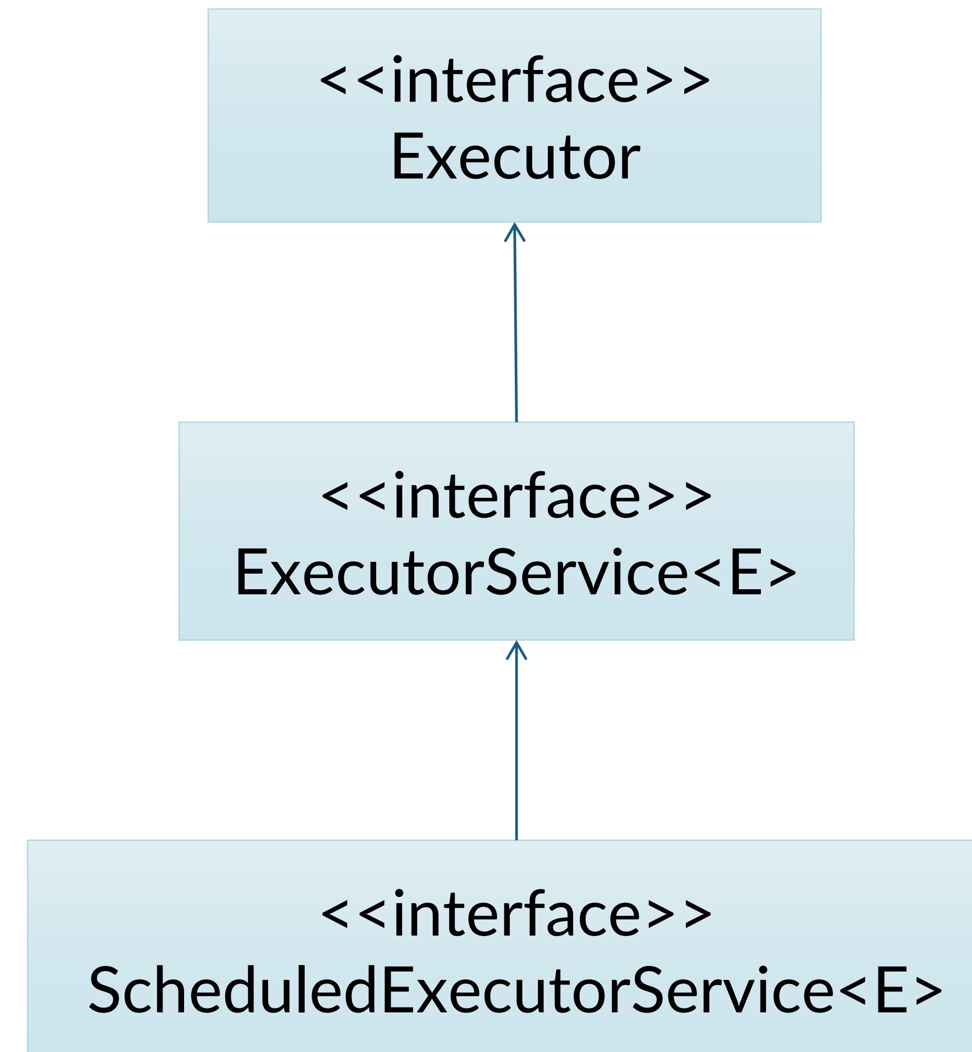
A synchronization point at which threads can pair and swap elements within pairs. Each thread presents some object on entry to the exchange method, matches with a partner thread, and receives its partner's object on return.

© 2019 ESTECO SpA

Executor

An executor initiates and controls the execution of threads

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



© 2019 ESTECO SpA

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();
}
```

© 2019 ESTECO SpA

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 …;
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit) throws …;
    <T> T invokeAny(Collection<? extends Callable<T>> tasks) throws …;
    <T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit) throws …;
    boolean awaitTermination(long timeout, TimeUnit unit) throws …;
    boolean isTerminated();
    void shutdown();
    boolean isShutdown();
}
```

© 2019 ESTECO SpA

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)
```

© 2019 ESTECO SpA

Digression – Termination of Java programs

- A Java program continues until there are alive/running non-daemon threads
- A Java program terminates when all alive/running threads are daemon threads
- The Java main thread is a non-daemon thread
- In an ExecutorService can be convenient to create daemon threads only or to invoke shutdown

© 2019 ESTECO SpA

Concurrent collections

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

Provide thread-safety and various forms of synchronization

© 2019 ESTECO SpA

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.

© 2019 ESTECO SpA

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)
    …
}
```

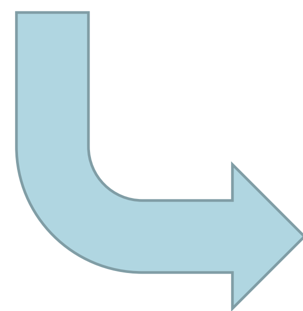
© 2019 ESTECO SpA

AtomicInteger example

```java
public class IdGenerator {

    private int id = 0;

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



```java
public class IdGenerator {

    private AtomicInteger id = new AtomicInteger();

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

© 2019 ESTECO SpA

Thank you!

esteco.com

Read the ESTECO Copyright Policy © 2019 ESTECO SpA