Lab 3

Java Atomic data types

The Java platform offers in the package *java.util.concurrent.atomic* several atomic data types implementations. Among these you can find:

- **`AtomicBoolean`** - a boolean value that can be updated atomically
- **`AtomicInteger`** - an integer value that can be updated atomically
- **`AtomicReference<V>`** - an object reference that can be updated atomically

Each of these classes offers a common series of methods that can be useful in thread synchronization, like:

- **`get()`** - returns the current value having the same effect considering the memory as reading a volatile variable
- **`set(newValue)`** - sets the current value to the specified parameter having the same effect considering the memory as writing a volatile variable
- **`getAndSet(newValue)`** - returns the old value and sets the new one specified as parameter; has the atomic effect of a `get()` followed by a `set()`
- **`compareAndSet(expectedValue, newValue)`** - if the current value of is the one given in the `expectedValue` parameter, it will be changed to the one given in the `newValue` parameter; the call will return true in case of changing the value, or false otherwise

One example of using the AtomicBoolean is the next Lock implementation:

```
class TASlock {
      AtomicBoolean state = new AtomicBoolean(false);

      void lock() {
            while (state.getAndSet(true)) {};
      }

      void unlock() {
            state.set(false);
      }
}
```

Peterson's mutual exclusion

One of the existing solutions for ensuring mutual exclusion for n threads is the generalized Peterson algorithm presented below. The basic idea is to pass each thread through a filter of n-1 levels up to the critical section. i can be considered to designate the thread identifier and L a stage level. The level array holds the current level for each thread, and the victim array holds the identifier of the last thread to enter one level

```
lock() {
      for (int L = 1; L < n; L++) {
            level[i] = L;
            victim[L] = i;
            while (( exists k != i with level[k] >= L ) &&
                  victim [L] == i ) {};
      }
}

unlock() {
      level[i] = 0;
}
```