# Concurrent Programming - First Work Assignment

## Introduction

This document contains the requirements for the first work assignment of the Concurrent Programming (*Programação Concorrente*) course. The solution to this work assignment must be delivered until the end of 2025-03-30, via the creation (and push) of the `0.1.0` Git tag. For each requirement:

- Include technical documentation with the solution design, e.g. as a KDoc to the complete class (e.g. example).
- Include automated tests, including both basic functional tests and *stress tests*.

This work assignment should be done in groups of 3 students.

## Requirement 1 - `BoundedStream<T>` class

Develop the `BoundedStream<T>` class implementing a thread-safe bounded multi-producer multi-consumer sequence of items, where each item is associated to an monotonically increasing index, starting at index zero.

Each instance is constructed with a capacity. Producers write items of type `T` into the end of the stream. If the stream is already at full capacity, then the oldest item (i.e. the item with lowest index) is discarded. Indexes are not reused, i.e., no two written items can have the same index.

Consumers read items starting at a given index: if this index is lower or equal that the stream's highest index, then the read request is completed immediately; otherwise the read will block until the stream has elements with index greater or equal to the requested one. The read operation does not remove the items of the stream. Each consumer is responsible for maintaining its own reading index independently of the `BoundedStream<T>` instance.

The stream should also support a close operation, after which all read or write operations should terminate immediately with an indication of the stream closed state.

The blocking operations, namely the `read` function, must support cancellation via timeout and must also implement the JVM interruption protocol.

*The `BoundedStream` class*

```
class BoundedStream<T>(capacity: Int) : Closeable {

    fun write(item: T): WriteResult {...}

    @Throws(InterruptedException::class)
    fun read(startIndex: Int, timeout: Duration): ReadResult {...}
    override fun close() {...}

    sealed interface WriteResult {
        // Write was done successfully
        data object Success : WriteResult

        // Write cannot be done because stream is closed
        data object Closed : WriteResult
    }

    sealed interface ReadResult<out T> {
        // Read cannot be done because stream is closed
        data object Closed: ReadResult<Nothing>

        // Read cannot be done because the timeout was exceeded
        data object Timeout: ReadResult<Nothing>

        // Read was done successfully and items are returned
        data class Success<T>(
            // Read items
            val items: List<T>,
            // Index of the first read item
            val startIndex: Long,
        ): ReadResult<T>
    }
}
```

## Requirement 2 - `ThreadScope` class.

Implement the `ThreadScope` class, responsible for holding a set of related threads. Each `ThreadScope` instance is constructed with a *name* (of type `String`) and a *thread builder* (of type `Thread.Builder`), and supports the following operations:

- `startThread`` - starts a new thread managed by the scope, given a `Runnable`.
- `newChildScope` - creates a new child thread scope, given a name.
- `close` - closes the scope, disallowing the creation of any further thread or child scope. The `close` operation is idempotent.
- `cancel` - interrupts all threads in the scope and cancels all child scopes. A `cancel` operation performs an implicit close.
- `join` - waits for a scope to be completed, which is defined by all started threads being terminated and all child scopes being completed.

*The* `ThreadScope` *class*

```
class ThreadScope(...) : Closeable {

    // Creates a new thread in the scope, if the scope is not closed.
    fun startThread(runnable: Runnable): Thread? {...}

    // Creates a new child scope, if the current scope is not closed.
    fun newChildScope(name: String): ThreadScope? {...}

    // Closes the current scope, disallowing the creation of any further thread
    // or child scope.
    override fun close() {...}

    // Waits until all threads and child scopes have completed
    @Throws(InterruptedException::class)
    fun join(timeout: Duration): Boolean {...}

    // Interrupts all threads in the scope and cancels all child scopes.
    fun cancel() {...}
}
```

## Requirement 3 - Broadcast TCP server

Implement a TCP server, based on the examples provided in the lectures, where a line sent via a connection is broadcasted to all other active connections. Use the `BoundedStream<T>` class to hold the last N lines and to support the communication between connections.

Each connection should use two threads: a *reader* thread that reads from the socket and writes into a shared `BoundedStream<T>`; a *writer* thread that reads from the shared `BoundedStream<T>` and writes into the socket. Use the `ThreadScope` to help correctly manage the created threads, namely to ensure:

- When a connection is closed, both associated threads are terminated.
- When a server is closed, all associated threads are terminated. Namely, when waiting for a server to terminate, this wait should consider all threads created in the context of that server.

This TCP server should also support the following additional functionality:

- Use JVM's *shutdown hooks* to correctly terminate a server.
- Support listening for incoming connections in more than one port.
- Send a farewell message to to each active connection when the server is shuting down.
- Print a message into *standard output* when all created threads are completed.
- Each connection should have an unique identifier, based on a monotonically increasing integer. This unique identifier should be written to the connection in a welcome message. This unique identifier should also be used to identify all messages (i.e. lines) communicated through the server.

## Requirement 4 - `CyclicCountDownLatch` class

Implement the `CyclicCountDownLatch` synchronizer class, with the following public interface

*The `CyclicCountDownLatch` class*

```
class CyclicCountDownLatch(val initialCount: Int) {
   init { require(initialCount > 0) }

   fun countDown(): Int { ⋯ }

   @Throws(InterruptedException::class)
   fun await(timeout: Duration): Boolean { ⋯ }
}
```

The countDown function decrements the counter, which is initialized with `initialCount`. If the counter reaches zero, it must be immediately reset to `initialCount`, and all threads that are currently waiting in the `await` function must successfully complete their wait. The return value to a `countDown` function call must return the number of threads that successfully completed their wait due to that call.

The `await` function must passively wait for a call to `countDown` to decrement the counter to zero, while allowing cancellation due to a timeout or thread interruption. The await function returns `true` if the wait completes successfully, or `false` if it ends due to a timeout.