Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Programação Concorrente
Teste Global da Época de Recurso, Verão de 2021/2022
Duração: 3 horas

---

1. [5] Implemente a classe **`ExecutorWithShutdown`** que adiciona a um executor, recebido no construtor, a capacidade de *shutdown*, isto é, colocar o objecto num estado em que não são aceites mais comandos.

```
class ExecutorWithShutdown(private val executor: Executor) {
   @Throws(RejectedExecutionException::class)
   fun execute(command: Runnable): Unit {...}
   fun shutdown(): Unit {...}
   @Throws(InterruptedException::class)
   fun awaitTermination(timeout: Duration): Boolean {...}
}
```

   A função **`execute`** delega a execução do comando no executor recebido no construtor. Caso esteja em modo de *shutdown*, lança a excepção **`RejectedExecutionException.`** A função **`shutdown`** coloca o executor em modo de *shutdown*, tendo como consequência que todas as chamadas de **execute** resultam no lançamento de **`RejectedExecutionException`**. A função **`awaitTermination`** bloqueia a *thread* invocante até que o executor esteja em modo de *shutdown* e que todos os comandos submetidos tenham sido executados. Esta função retorna **`true`** se a condição anterior for verdadeira, **`false`** se a condição anterior for falsa e o tempo máximo de espera definido por **`timeout`** tiver decorrido. A função **`awaitTermination`** deve reagir a interrupções da *thread* invocante, acabando a espera com o lançamento da excepção **`InterruptedException`**. Note que o comando que é passado a **`executor`** pode ser diferente do recebido na função **`execute`**.

2. [5] Implemente o sincronizador *message queue*, para suportar a comunicação entre *threads* produtoras e consumidoras através de mensagens do tipo genérico **`T`**. A comunicação deve usar o critério FIFO (*first in first out*). A interface pública deste sincronizador é a seguinte:

```
class MessageQueue<T>() {
   @Throws(InterruptedException::class)
   fun tryEnqueue(message: T, timeout: Duration): Thread? { … }
   @Throws(InterruptedException::class)
   fun tryDequeue(nOfMessages: Int, timeout: Duration): List<T> { … }
}
```

   O método **`tryEnqueue`** entrega uma mensagem à fila, retornando a referência para a *thread* que consumiu a mensagem. O método **`tryEnqueue`** fica bloqueado até que: 1) a mensagem entregue seja entregue a um consumidor, 2) o tempo **`timeout`** definido para a operação não expirar, ou 3) a *thread* não for interrompida. O método **`tryDequeue`** tenta remover **`nOfMessages`** mensagens da fila, bloqueando a *thread* invocante enquanto: essa operação não puder ser concluída com sucesso, 2) o tempo **`timeout`** definido para a operação não expirar, ou 3) a *thread* não for interrompida. A remoção pode ser realizada parcialmente, apenas caso o tempo de espera seja excedido, ou seja, a função **`tryDequeue`** pode retornar uma lista com dimensão inferior a

nOfMessages. Estas operações de remoção devem ser completadas pela ordem de chegada, independentemente dos valores de **nOfMessages**. Tenha em atenção as consequências de uma desistência, por interrupção ou *timeout*, de uma operação de **tryDequeue**.

3. [3] Considere a seguinte implementação não *thread-safe* de um contentor de objectos com contagem de utilizações, que automaticamente chama a função **close** quando essa contagem de utilizações é zero. Realize, sem utilizar *locks*, uma versão *thread-safe* desta classe.

```
class UnsafeUsageCountedHolder<T : Closeable>(value: T) {
   private var value: T? = value
   private var useCounter: Int = 1

   fun startUse(): T {
       if (useCounter == 0) throw IllegalStateException("Already closed")
       useCounter += 1
       return value ?: throw IllegalStateException("Already closed")
   }

   fun endUse() {
       if (useCounter == 0) throw IllegalStateException("Already closed")
       if (--useCounter == 0) {
           value?.close()
           value = null
       }
   }
}
```

4. [3] Realize o sincronizador **MessageBox** com a interface apresentada em seguida.

```
class MessageBox<T> {
   suspend fun waitForMessage(): T { … }
   fun sendToAll(message: T): Int { … }
}
```

A função **waitForMessage** suspende a corrotina onde foi realizada a invocação até que uma mensagem seja enviada através da função **sendToAll**. A função **sendToAll** deve retornar o número exacto de chamadas a **waitForMessage** que receberam a mensagem, podendo este valor ser zero (não existiam corrotinas à espera de mensagem), um, ou maior que um. A mensagem passada na chamada **sendToAll** não deve ficar disponível para chamadas futuras da função **waitForMessage**.

5. [4] Implemente a função com a seguinte assinatura

```
suspend fun <A,B,C> run(f0: suspend ()->A, f1: suspend ()->B, f2: suspend (A,B)->C): C
```

que retorna o valor da expressão **f2(f0(), f1())**, realizando a computação de **f0()** e **f1()** em paralelo.

Duração: 3 horas
ISEL, 20 de julho de 2022