# Laboratorul 10: Corutine - suport nativ pentru paralelism în Kotlin

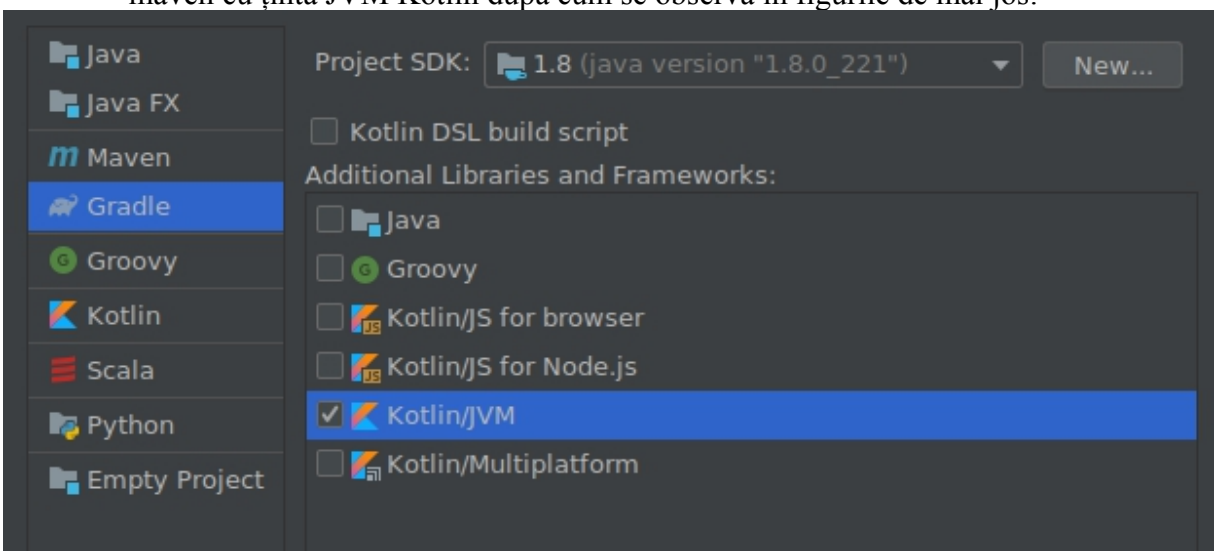
## Introducere

Pentru o imagine de ansamblu asupra corutinelor, vezi:

- **„Hands-on design patterns with Kotlin”** - Alexey Soshin
- **„Functional Kotlin”** - Mario Arias, Rivu Chakraborty
- Cursul 10 de la disciplina *Paradigme de programare*
- https://github.com/Kotlin/kotlinx.coroutines/blob/master/docs/topics/coroutines-guide.md

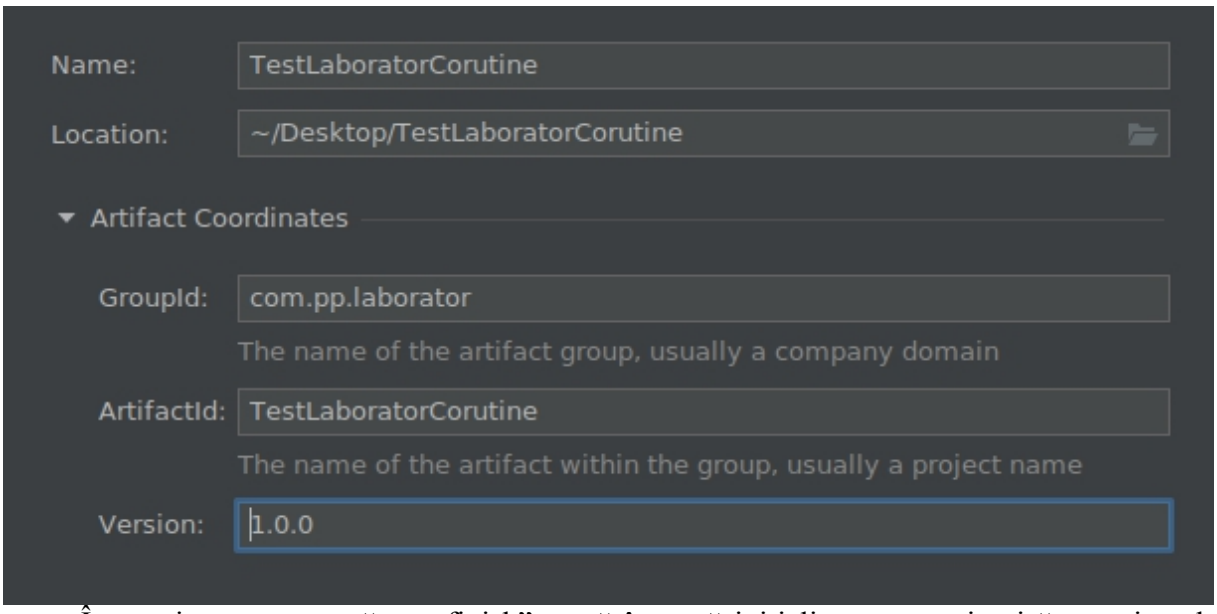
Înainte de a trece la realizarea unei aplicații utilizând corutinele în Kotlin, este bine să se realizeze următorii pași:

1. se lansează procedura de update a mediului de dezvoltare până când nu mai sunt actualizări de aplicat
2. Crearea unui proiect gol pentru a utiliza corutinele. După cum a fost discutat la curs, Kotlin folosește o bibliotecă separată pentrui a avea abilități specifice calculului paralel. Pentru aceasta, ea trebuie introdusă explicit în configurarea unui proiect. Se va verifica ca Internetul să fie conectat. Se va porni de la un proiect nou de gradle / maven cu țintă JVM Kotlin după cum se observă în figurile de mai jos:

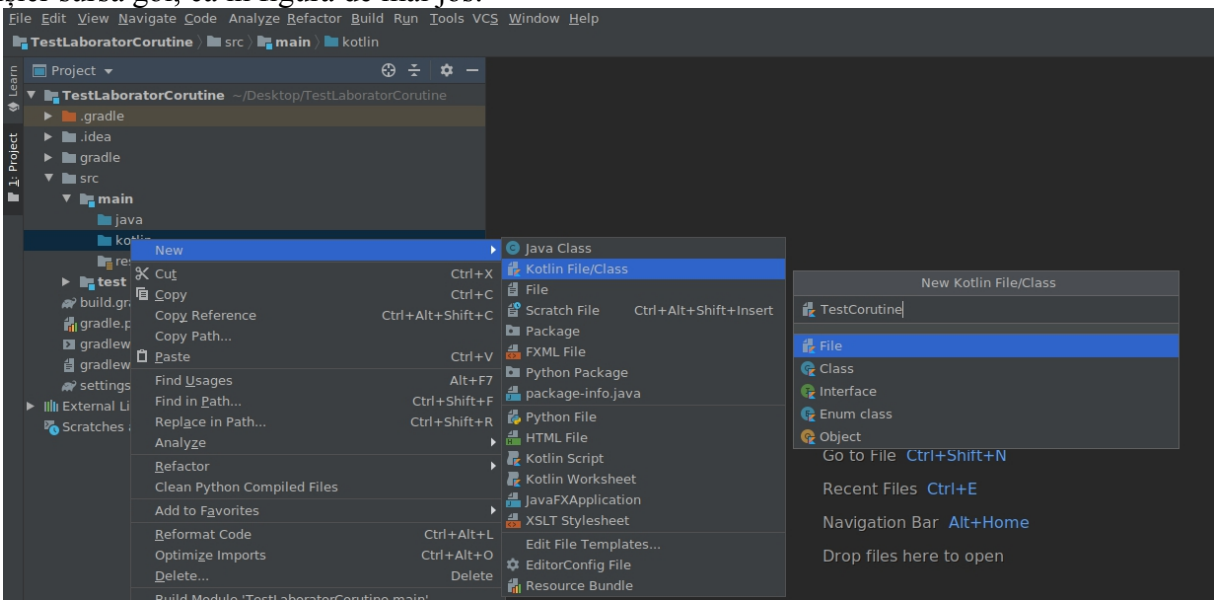



Creare proiect Gradle

apoi se specifică numele acestuia (spre exemplu: *TestLaboratorCorutine*)




În continuare se apasă pe „finish” ca să înceapă inițializarea propriu-zisă a proiectului. Mediul de dezvoltare va efectua la început (dacă este cazul) o serie de descărcări de pachete, precum și configurări automate ale acestora, deci în funcție de performanțele sistemului, acest pas poate avea o durată de timp variabilă. După ce acest proces s-a terminat, trebuie creat un fișier sursă gol, ca în figura de mai jos:

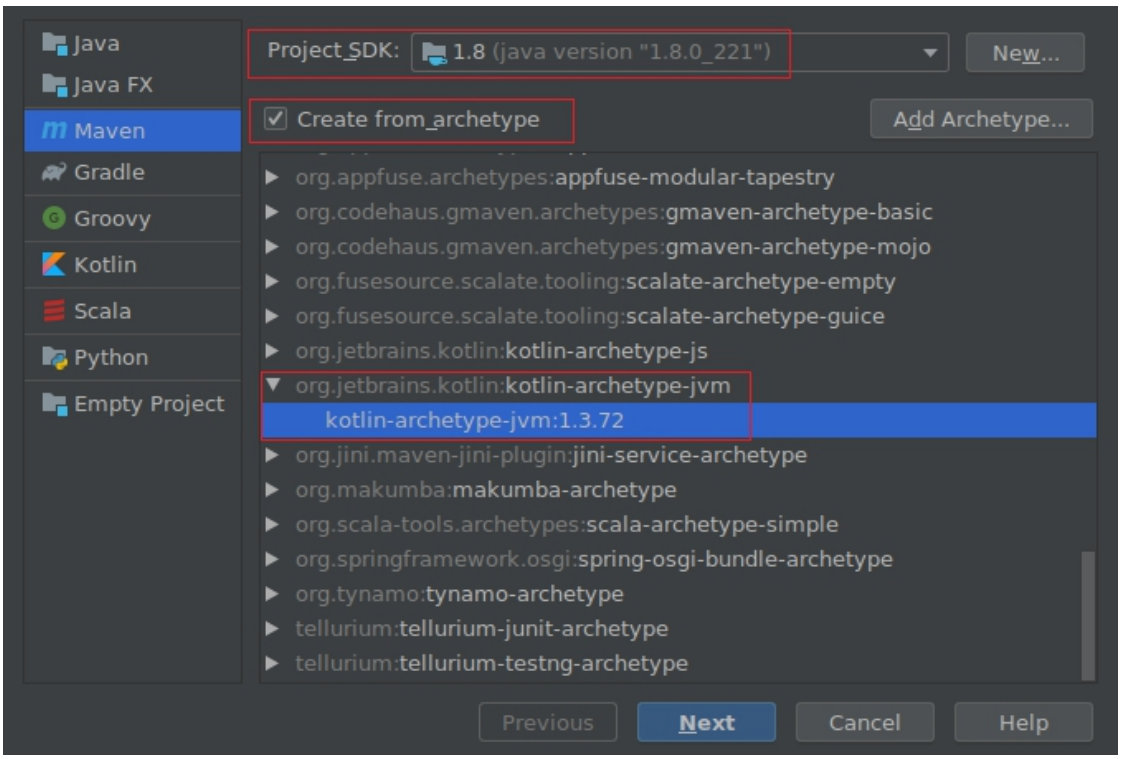



Crearea unui fișier sursă Kotlin într-un proiect Gradle

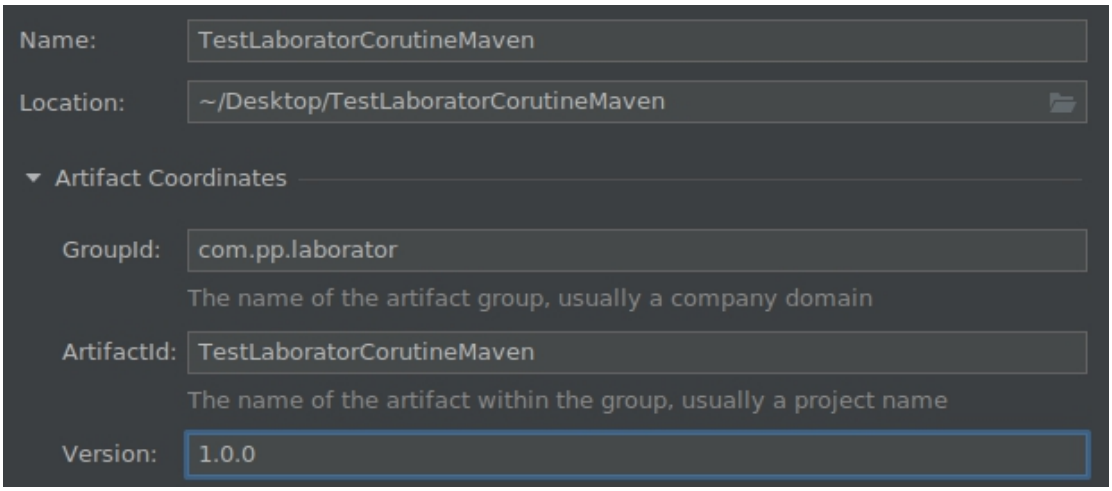
Pentru un proiect **Gradle**, se vor adăuga în fișierul *build.gradle* următoarea configurare:

```
dependencies {
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-
core:1.6.0'
}
```

**ALTERNATIV, se poate crea un proiect Maven:**

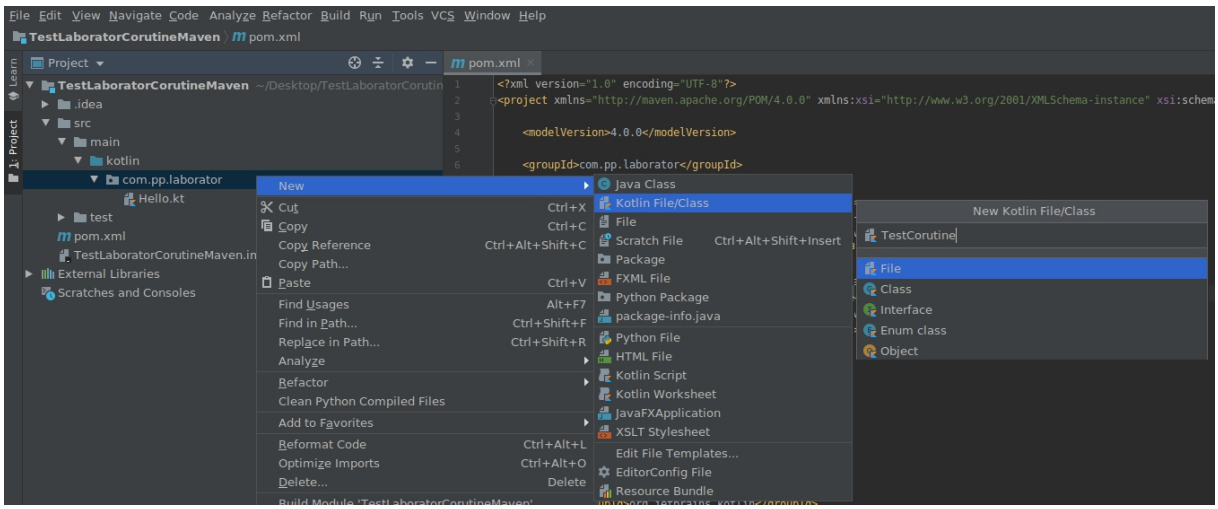



Creare proiect Maven




Denumirea proiectului Maven

La final, se apasă în colțul dreapta jos pe Import Changes.




Crearea unui fișier sursă Kotlin într-un proiect Maven

Pentru un proiect **Maven**, în fișierul *pom.xml* se va adăuga urmatoarea dependență (ca subelement al tag-ului <dependencies>):

```
<dependency>
   <groupId>org.jetbrains.kotlinx</groupId>
   <artifactId>kotlinx-coroutines-core</artifactId>
   <version>1.6.0</version>
   <type>pom</type>
</dependency>
```

## Funcții recursive

Kotlin suportă un stil de programare funcțională cunoscut ca *recursivitatea coadă* (tail recursion). Aceasta permite unor algoritmi care în mod normal ar fi fost scrișI cu bucle să fie scris cu o funcție recursivă, dar fără riscul de „stack overflow”.

Spre deosebire de recursivitatea normală în care toate apelurile recursive sunt efecutate la început și apoi se calculează rezultatul din valorile returnate la urmă, în *recursivitatea coadă* calculele sunt execcutate primele, apoi apelurile recursive (apelul recursiv trimite rezultatul pasului curent către următorul apel recursiv).

Când o funcție este marcată cu modificatorul ***tailrec*** și are forma corespunzătoare, compilatorul optimizează recursivitatea, rezultând o versiune bazată pe o buclă rapidă și eficientă în loc:

```
val eps = 1E-10 // suficient, dar ar putea fi si 10^(-15)

tailrec fun findFixPoint(x: Double = 1.0): Double = if (Math.abs(x -
Math.cos(x)) < eps) x else findFixPoint(Math.cos(x))
```

Exemplul de mai sus calculează punctul de referință al cosinusului (o constantă matematică). Apelează în mod repetat Math.cos începând cu 1.0 până când rezultatul nu se mai schimbă, făcând yield la valoarea 0.7390851332151611.

Codul de mai sus este echivalent cu forma tradițională de mai jos:

```
val eps = 1E-10 // suficient, dar ar putea fi si 10^(-15)

private fun findFixPoint(): Double {
    var x = 1.0
```

```
    while (true) {
        val y = Math.cos(x)
        if (Math.abs(x - y) < eps) return x
        x = Math.cos(x)
    }
}
```

### *Corutine recursive*

Pentru ca o funcție recursivă *tailrec* să poată fi utilizată într-o corutină, trebuie utilizat cuvântul cheie *suspend:*

```
tailrec suspend fun fibonacci(n: Int, a: Long, b: Long): Long {
    return if (n == 0) a else fibonacci(n-1, b, a+b)
}
```

## Debug pe corutine

Pentru proiect Gradle, se va adăuga următoarea dependență în *build.gradle*:

```
dependencies {
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-
debug:1.6.0'
}
```

Pentru proiect Maven, se va adăuga dependența de mai jos în *pom.xml*:

```
<dependency>
   <groupId>org.jetbrains.kotlinx</groupId>
   <artifactId>kotlinx-coroutines-debug</artifactId>
   <version>1.6.0</version>
</dependency>
```

### *Exemplu de debug*

```
package com.pp.laborator

import kotlinx.coroutines.*
import kotlinx.coroutines.debug.*

suspend fun computeValue(): String = coroutineScope {
    val one = async { computeOne() }
    val two = async { computeTwo() }
    combineResults(one, two)
}

suspend fun combineResults(one: Deferred<String>, two:
Deferred<String>): String =
    one.await() + two.await()

suspend fun computeOne(): String {
    delay(5000)
    return "4"
}
```

```
suspend fun computeTwo(): String {
    delay(5000)
    return "2"
}

fun main() = runBlocking {
    DebugProbes.install()
    val deferred = async { computeValue() }
    // Delay for some time
    delay(1000)
    // Dump running coroutines
    DebugProbes.dumpCoroutines()
    println("\nDumping only deferred")
    DebugProbes.printJob(deferred)
}
```

### *Exemplu de logging*

```
package com.pp.laborator

import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main() = runBlocking<Unit> {
    val a = async {
        log("I'm computing a piece of the answer")
        6
    }
    val b = async {
        log("I'm computing another piece of the answer")
        7
    }
    log("The answer is ${a.await() * b.await()}")
}
```

### *Exemplu de logging cu corutine denumite*

```
package com.pp.laborator

import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main() = runBlocking<Unit> {
    log("Started main coroutine")
// run two background value computations
    val v1 = async(CoroutineName("v1coroutine")) {
        delay(500)
        log("Computing v1")
        256
    }
    val v2 = async(CoroutineName("v2coroutine")) {
        delay(1000)
        log("Computing v2")
```

```
        8
    }
    log("The answer for v1 / v2 = ${v1.await() / v2.await()}")
}
```

Pentru mai multe detalii, vezi:

- https://kotlinlang.org/docs/reference/coroutines/coroutine-context-and-dispatchers.html#debugging-coroutines-and-threads
- https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-debug/kotlinx.coroutines.debug/-debug-probes/
- https://github.com/Kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-debug/README.md

## Exemple

### *Exemplul 1*

Acesta este un exemplu din curs și pune mai clar în evidență problemele specifice lipsei măsurilor necesare pentru asigurarea coerenței datelor.

```
package com.pp.laborator

import kotlinx.coroutines.*
import kotlin.system.*
suspend fun CoroutineScope.massiveRun(action: suspend () -> Unit) {
    val n = 100
    val k = 1000
    val time = measureTimeMillis {
        val jobs = List(n)
        {
            launch { repeat(k) { action() } }
        }
        jobs.forEach { it.join() }
    }
    println("S-au efectuat ${n * k} operatii in $time ms")
}

val mtContext = newFixedThreadPoolContext(2, "mtPool")
var counter = 0
fun main() = runBlocking<Unit> {
    CoroutineScope(mtContext).massiveRun {
        counter++ //variabila comuna unde vor aparea erori
    }
    println("Numarator = $counter")
}
```

### *Exemplul 2*

Se pornește de la un exemplu cu actori (discutat la curs) care reprezintă entitatea creată prin combinarea unei corutine, o stare care este izolată în interiorul acestei corutine și un canal de comunicație cu alte subrutine care este prezentat mai jos:

```
package com.pp.laborator
```

```
import kotlinx.coroutines.*
import kotlin.system.*
import kotlinx.coroutines.channels.*

sealed class ContorMsg

object IncContor : ContorMsg()

class GetContor(val response: CompletableDeferred<Int>) : ContorMsg()

fun CoroutineScope.counterActor() = actor<ContorMsg> {
    var contor = 0
    for (msg in channel) {
        when (msg) {
            is IncContor -> contor++
            is GetContor -> msg.response.complete(contor)
        }
    }
}

suspend fun CoroutineScope.massiveRun(action: suspend () -> Unit) {
    val n = 100
    val k = 1000
    val timp = measureTimeMillis {
        val jobs = List(n) {
            launch {
                repeat(k) { action() }
            }
        }
        jobs.forEach { it.join() }
    }
    println("Am terminat atatea ${n * k} actiuni in $timp ms")
}

fun main() = runBlocking<Unit> {
    val contor = counterActor()
    GlobalScope.massiveRun()
    {
        contor.send(IncContor)
        println(contor.onSend)
    }
    val raspuns = CompletableDeferred<Int>()
    contor.send(GetContor(raspuns))
    println("Contor = ${raspuns.await()}")
    contor.close()
}
```

### *Exemplul 3*

```
package com.pp.laborator

fun main(args: Array<String>){
    object : Thread() {
     override fun run() {
      println("Sunt in thread-ul singleton ${Thread.currentThread()}")
     }
    }.start()
    val t1=SimpleThread()
    t1.run()
    val t2=SimpleRunnable()
    t2.run()
    val thread = Thread {
        println("Thread lambda ${Thread.currentThread()} s-a
executat.")
    }
    thread.start()
}
class SimpleThread: Thread() {
    public override fun run() {
        println("Instanta clasei derivate din Thread
${Thread.currentThread()} s-a executat.")
    }
}
class SimpleRunnable: Runnable {
    public override fun run() {
        println("Instanta clasei care implementeaza Runnable
${Thread.currentThread()} s-a executat.")
    }
}
```

## Aplicații și teme

**Aplicații de laborator:**

- Pornind de la exemplul 1, va trebui să căutați o manieră de rezolvare/evitare a apariției erorilor în codul respectiv. De asemenea, problema va trebui modificată astfel încât să se genereze un set de valori care să fie depus întâi într-un ADT și apoi să fie scris **în mod concurent** într-un fișier pe disc.

- Reluați problema de la exemplul 1 utilizând abordarea cu actori.

- Pornind de la exemplul 3 să se completeze/modifice codul de mai jos care utilizează modelul Singleton pentru a implementa explicit un semafor. Apoi, se va utiliza pentru a executa un acces thread safe multiplu la un fișier de date. Apoi, să se reia implementarea cu corutine. Trebuie menționat că acest exemplu are doar o utilitate didactică, fiind redundant, deoarece atât thread-urile cât și corutinele au mecanisme specifice mult mai eficiente pentru gestionarea problemei.

```
package com.pp.laborator

import java.io.File

class Log private constructor() {
    companion object {
        val instance = Log()
        val fname = "Semafor.txt"
    }
    fun Write(line : String) {
        File(fname).appendText(line)
    }
    fun Reset(){
        File(fname).delete()
    }
}

class Semafor private constructor() {
    fun Enter() : Boolean {

    }
    fun Exit() {

    }
}
```

**Tema pe acasă:**

**1)** Să se proiecteze și să se implementeze un lanț de responsabilități dublu (similar unei liste dublu înlănțuite). Scopul duplicării lanțului este de a trimite un răspuns către handler-ul ierarhic superior la finalul procesării unei cereri, pentru a-l anunța că sarcina a fost încheiată cu succes (sau nu). Pentru instanțierea Handler-elor, se va utiliza modelul fabrică abstractă, care va crea două fabrici:

- EliteFactory ce permite crearea CEOHandler, ExecutiveHandler și ManagerHandler;
- HappyWorkerFactory ce permite crearea HappyWorkerHandler;

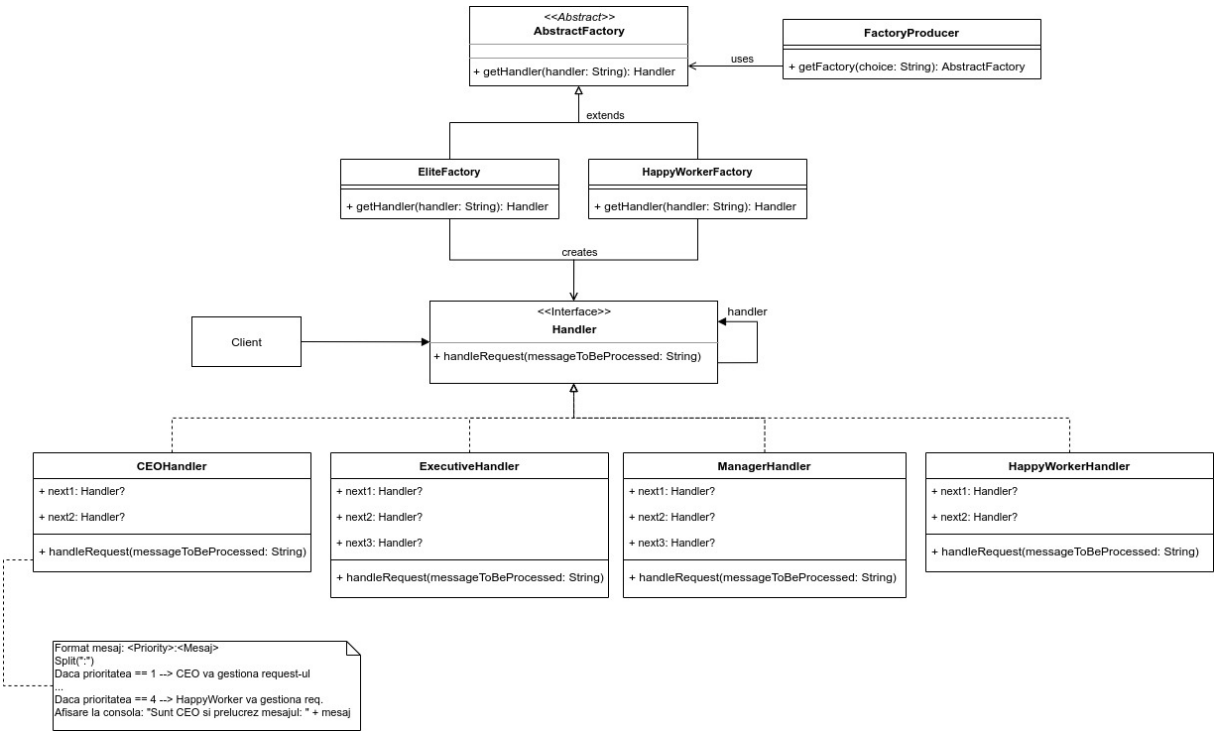



Diagrama de clase

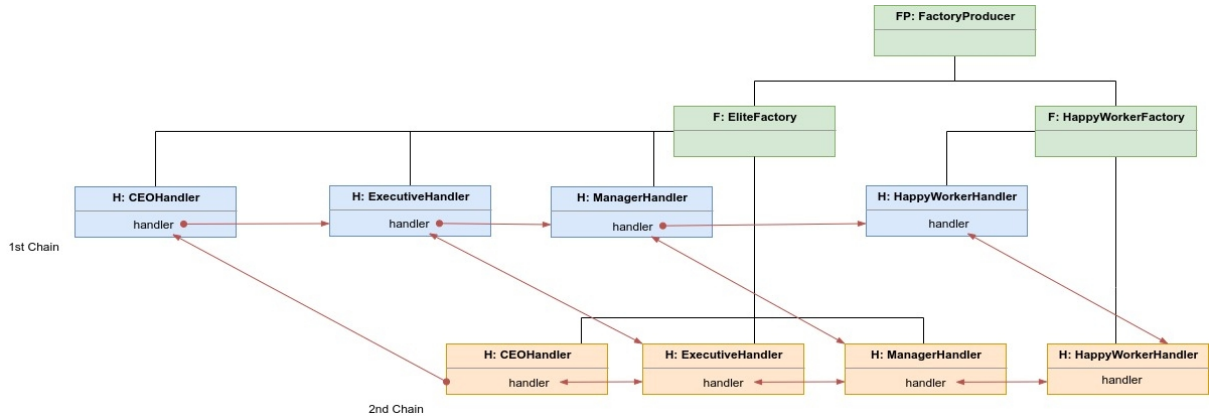



Diagrama de obiecte

Spre deosebire de laboratorul 8, funcțiile *handleRequest* din Handler-e vor fi de tip *suspend,* și fiecare va crea o nouă corutină de tratare a cererii sosită în următoarea manieră:

- dacă cererea sosită este menită handler-ului curent, aceasta va fi prelucrată într-o corutină pe handler-ul curent, iar răspunsul va fi trimis printr-un request pe lanțul de jos, care îl va trimite la superiorul ierarhic de pe acest lanț, ajungând în final la superiorul ierarhic de pe lanțul de sus.
- dacă cererea **NU** trebuie prelucrată de handler-ul curent, se va crea o corutină de tratare ce va apela funcția *handleRequest* a următorului handler
- se va utiliza funcția *delay* cu perioadă variabilă, pentru a simula procesările de durată.

Observație: Se poate reproiecta aplicația astfel încât handler-ul să primească corutina care gestionează request-ul ca parametru.

Structura mesajului:

- pentru un mesaj trimis ca cerere:

```
Request - <mesaj>
```

- pentru un mesaj trimis ca răspuns:

```
Response - <mesaj>
```

Exemplu de flux de execuție: se creează o cerere care trebuie prelucrată de ExecutiveHandler. Aceasta va ajunge întâi la CEOHandler pe lanțul de sus, care îl va trimite pe același lanț la ExecutiveHandler. Acesta va prelucra cererea și va trimite răspunsul pe lanțul de jos (tot la ExecutiveHandler), care îl trimite la CEOHandler (tot lanțul de jos), iar la final, ajunge înapoi pe primul lanț.

**Următoarele teme pe acasă se vor implementa atât cu corutine cât și cu apeluri de thread-uri Java:**

2. Să se realizeze o procesare după modelul pipeline a unui ADT întreg. Primul thread din pipe înmulțește toate elementele din vectorul V cu o constantă alpha, următorul thread din pipe va ordona mulțimea, iar ultimul thread o va afișa.
3. Să se realizeze calculul $\sum_{0}^{n} i$ simultan pentru patru valori diferite ale lui *n* luate dintr-o coadă de către patru corutine diferite.
4. Să se modifice algoritmul din exemplul 3 astfel încât să poată procesa mai multe fișiere simultan (se pot folosi pool și actori dacă se dorește)

**[BONUS]:** Utilizând canale și eventual actori, să se implementeze pipeline-ul din figura de mai jos:

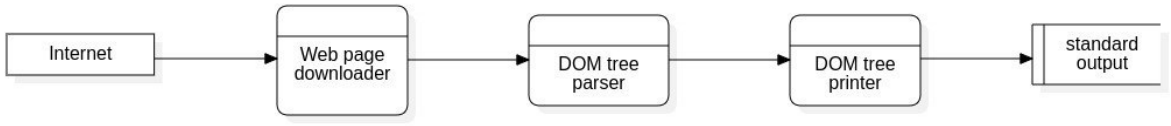



Diagrama fluxului de date

Primul procesor din pipeline va descărca o pagină WEB oarecare, al doilea va citi arborele DOM, iar al treilea îl va afișa la consolă sub formă arborescentă (se poate utiliza și forma indentată - vezi comanda *tree* din linux).